Towards Evidence-Based Architectural Design for Safety-Critical Software Applications

Weihang Wu and Tim Kelly

Department of Computer Science, The University of York, York YO10 5DD
{Weihang.Wu,Tim.Kelly}@cs.york.ac.uk

Abstract. Robust software and system architectures have been increasingly recognised as one of the keys to improving dependability. However, most modern design methods and explanations of underlying design principles still remain ad hoc. The communication between design and safety assessment in practice is often characterised as an "over-the-wall" process. The problems are exacerbated by the uncertainty problem in the early development lifecycle. In this paper, we propose a Triple Peaks process framework, from which a system model, deviation model, mitigation model are proposed and linked together. The application of this framework is supported by the use of Bayesian Belief Networks and collation of relevant evidence. We elaborate the linkage between the three models by means of a case study. The central tenet in this paper is to address safety concerns based upon evidence available at an architectural level.

Keywords: software architecture, design decisions, software safety evidence.

1 Introduction

1.1 Motivation

For many years there has been an objective to improve software and system safety. Testing and inspection late in the system development lifecycle should no longer be relied upon as the primary line of defence for engineering software systems of significant size and complexity. Empirical experience shows that problems identified in the late lifecycle are often costly to fix and may introduce unexpected new problems [17]. Robust software and system architectures have been increasingly recognised as one of the keys to improving safety.

However, most modern architectural design methods and explanations of underlying design principles remain ad hoc. Architects or designers, who could claim in their defence that they adopted a specific design pattern or followed an industry standard, rarely articulate their design rationale and analyse the impact of their decisions along with design alternatives in a precise and sound manner. The communication between design and safety assessment in practice is often characterised as an "over-the-wall" process [19]. The problems are exacerbated by the presence of a high degree of uncertainty in the design detail that is available early in the system development lifecycle.

R. de Lemos et al. (Eds.): Architecting Dependable Systems IV, LNCS 4615, pp. 383–408, 2007.

1.2 Software Safety Evidence

The development of software safety evidence is increasingly advocated in the safety community [38] to explicitly evaluate the safety of software, as opposed to relying on process prescription through safety standards such as IEC 61508 [3] and DO178B [5]. The tenet of using software safety evidence is straightforward: evidence shall be provided for assessors to demonstrate sufficient mitigation of risks associated with the use of software in safety-critical systems. The term "sufficiency" has been defined and deployed in a variety of risk acceptance regimes in the domain of risk management Risk mitigation has been generalised in terms of the following activities: hazard elimination, hazard reduction, hazard detection and control [35]. In principle, like other system components, software can only contribute to hazards in the system context by means of deviations from its intended behaviour [35]. Thus, it is possible to bring together the notion of "deviation", "mitigation", and "risk acceptance" with the aid of "evidence". Here we define an item of software safety evidence to be an object encapsulating knowledge about potential deviations, plausible mitigation options and estimated risk reduction, along with reference to partial specification knowledge about a system and its environment.

Very often, safety evidence is produced after design completion. The need for incremental construction of safety evidence and corresponding safety arguments (a.k.a., safety cases) has been increasingly recognised. By utilising structured safety evidence explicitly from the very beginning of the system development lifecycle, the key issues such as loss of safety rationale and late discovery of safety flaws may be addressed. Figure 1 shows an evidence-oriented development process proposed by the Australian software safety standard DefAust 5679 [2], in which consideration of safety case development starts from the earliest stage of system development.

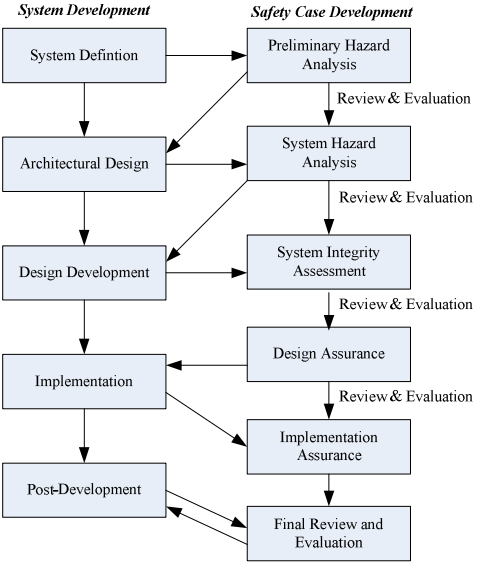


Fig. 1. The integrated development process (adapted from [2])

However, the linkage between the two processes (e.g., moving from preliminary hazard analysis to architectural design) still remains undefined, especially for software. Existing system safety approaches such as those advocated by ARP 4754 focus on the hazard analysis of purely functional requirements (i.e. Functional Failure Analysis – FFA), from which quantitative failure targets are defined and allocated, thereby driving the development of system and software architectures. Experience in application of FFA to engine controller development has revealed this technique is particularly vulnerable, as there is lack of rigorous techniques to identify and estimate controller-related failures with respect to levels of design detail [9]. The SEI (Software Engineering Institute) at Carnegie Mellon University has established a design method termed Attribute-Driven Design (ADD) [11] to emphasise the active role of quality attributes in architectural design. Yet there is little practical guidance on how to address safety concerns using ADD. Furthermore, the nature and amount of evidence changes as the design process progresses. Systematic techniques should be provided to handle these evolution issues within a structured evidence framework.

1.3 Scope of Paper

In attempting to integrate architecture design and safety evidence development processes in an effective manner, the interactions between the two must be elaborated – i.e. how requirements are evaluated by risk assessment, and design choices justified and design decisions driven by safety tradeoffs. In this paper, we introduce a Triple Peaks model as a framework for architectural design for safety. The process we present intertwines partial system specification, potential deviation concerns and plausible mitigation mechanisms incrementally and iteratively, through which incremental construction of safety arguments is facilitated and exploited. While risks may be communicated in a qualitative manner, they must be evaluated quantitatively, even in the early system development stages. We adopt Bayesian Belief Networks (BBNs) as the flexible medium for risk-based reasoning. The causal nature of BBNs allows reasoning about the propagation of software deviations and the effect of mitigation chosen. We illustrate the process by means of an aircraft wheel brake system (WBS) controller example extracted from ARP 4761 [1] and evaluate our process by comparison with the conventional ARP approach. We argue that systematic treatment of software safety evidence – guided by the Triple-Peaks model – holds the key to gaining confidence in safety-related architectural decision making.

This paper is organised in the following six sections. Section 2 reviews related work. Section 3 describes the process model within the evidence framework. Sections 4 and 5 elaborate the linkages between the three models by means of the WBS example. Finally, section 6 discusses the findings based upon the case studies conducted and section 7 presents the summary and future work.

2 Related Work

In order to inform architectural decisions, the first essential step for an architect is to interpret system and software requirements so that they can be understood. Goal-oriented methods, pioneered by van Lamsweerde [30, 31], proposed the use of goal

modelling languages such as Knowledge Acquisition in Automated Specification (KAOS) [31] to guide requirements elicitation and refinement. Alternatively, scenario-based approaches have been proposed in the form of use cases [24], Use Case Maps (UCMs) [14] and sequence diagrams [6] to elaborate requirements over a known system structure. Goals and scenarios are complementary to each other and can be combined in the design process [7]. Arguably, requirements to be addressed at an architectural level include both functional and quality requirements (e.g., timing, accuracy or reliability targets). Chung et al's Non-Functional Requirements (NFR) framework [40] and SEI's quality-attribute scenario framework [11] have been proposed for the purpose of formulating the quality requirements.

From the perspective of architecting dependable software, it is equally important to address all possible negative requirements. In contrast with (positive) requirements, negative requirements describe the system characteristics that are not allowed or desired. Nevertheless, the formulation of negative requirements and the relationship to positive requirements had not been explored until a decade ago. Potts et al informally proposed the notion of obstacles that might challenge the achievement of requirements within the Inquiry Cycle framework [45]. van Lamsweerde & Letier extended the KAOS language to incorporate the notion of obstacles as goal violation and provided heuristics on obstacle analysis over goals and resolution [32]. van Lamsweerde elaborated the obstacle framework further through anti-goals and anti-models in the context of security [29]. Rather than dealing with negative requirements at a goal level, complementary approaches have extended the notion of scenarios for the same purpose. Previous work at York [9] developed a method for deriving functional hazards from use cases. Alexander later proposed a unified view of deviation analysis over use cases in the form of misuse cases [8]. The safety community also turned their attention to extending hazard analysis at the requirements level to identify safety-related requirements errors. de Lemos et al [33] proposed an integrated framework that facilitates requirements analysis and hazard analysis iteratively and incrementally. Leveson et al [36] proposed to combine a set of hazard analysis techniques into an integrated safety analysis for checking safety-related requirements errors. All the approaches are defined solely in the context of requirements without considerations of architectural characteristics.

Given the positive and negative requirements formulated, the plausible design space must be elicited in order to capture the appropriate architectural choices. In the early 90s, Lane [48] proposed a multidimensional design space, each dimension representing relevant design choices to achieve a specific usability property. The SEI later developed a tree-form design space in terms of architectural tactics with respect to six common quality attributes [11]. The linkage between quality attributes and design space was also elaborated by SEI through the notion of quality-attribute reasoning frameworks [10]. A quality-attribute reasoning framework encapsulates knowledge about relevant analytic models for a quality attribute. For example, a performance reasoning framework imposes constraints on the relevant parameters for various performance measures such as a hard deadline. The collection of these frameworks thus offer an effective means of predicting system qualities and rationalising the selection of tactics. However, no specific techniques are provided for addressing the uncertainty and levels of design detail available in the early lifecycle.

From the viewpoint of safety, the reasoning framework should be built upon risk assessment. NASA have developed a probabilistic risk assessment (PRA) scheme [49] for more than two decades. The PRA approach is based upon the combination of fault tree analysis (FTA) [51] and event tree analysis (ETA) [35] and mandates a generic risk quantification and mitigation process that can be tailored to all phases of a project lifecycle. We believe that the burden of combining FTA and ETA can be relieved by the use of BBNs as a unified model. At the Jet Propulsion laboratory, a lightweight approach to risk assessment was developed, namely Defects Detection and Protection (DDP) [18]. The DDP scheme mandates the quantification of requirements, failure modes and mitigation by means of expert judgement. The underlying formal model of DDP is less justified, however.

Early work on argument-based design rationale (e.g., [16] and [46]) developed a set of generic models of design processes in terms of three common elements: issues/questions, positions/options, and arguments/criteria. The three elements are consistent with our design scheme as described in the section 1.2. The issues represent knowledge about deviations, the positions capture knowledge about possible mitigations, and arguments feature knowledge about reasoning. While the proposed design rationale models capture most common situations of design, they are too general to be configured for software architecture design problems. We believe there is a need to elaborate further the linkage of the three elements: i.e., how to generate issues, how to move from issues to positions, and then from positions to arguments.

The relationship between requirements engineering and architectures has recently been studied. Brandozzi and Perry [13] proposed the use of "architectural prescriptions" to describe the mappings between goals and architectural structures. Jackson et al [23] extended the problem frames approach to allow architectural decisions to be considered in requirements models in terms of "architectural frames". Both approaches have explored the achievement of system functionality rather than system qualities. Nuseibeh [41] proposed a Twin Peaks model which explicitly features the challenges raised during the parallel development of requirements and architectures. Leveson [34] proposed the intent specification approach to deriving software safety requirements. An intent specification comprises two main dimensions: intent and part-whole dimensions. The two dimensions dictate the requirements and safety-related design decisions made respectively in the design process. Nevertheless, there is lack of practical guidance on how to generate intent specifications.

The notion of BBNs was developed by combining probability theory and graph theory [44]. A BBN represents a directed acyclic graph together with associated conditional probability distributions based upon explicit independence assumptions, thereby saving space of probabilistic computation [44]. In practice, BBNs are often interpreted as causal models [44], in which the directed edges are captured by knowledge about causal relations. Several tools such as Netica [4] are also available for public evaluation. BBNs have already been applied to solving software engineering problems. Fenton et al [50] developed generic BBN patterns to quantify software safety risks. Sutcliffe et al proposed a method of constructing generic BBNs to evaluate usability [20] and later developed an automated tool to evaluate reliability and performance through different configurations of BBNs [21]. Bosch and Gurp [22] proposed a generic BBN model as a software architecture evaluation framework.

However, most of the BBN models developed are generic and thus need to be tailored for a specific system domain.

At York, there has been a long-term objective to integrate software design and safety analysis. A decade ago Fenelon et al [19] proposed a prototype of compositional failure modelling language – Failure Propagation and Transformation Notation (FPTN). In our previous work, we developed a collection of safety tactics [55] as primitive building blocks for software safety design. In order to identify safety concerns, we have also proposed a method for deviation analysis over UCMs [52]. We further elaborated it by developing a negative scenario framework and mitigation action model [54] to help generate design options for the safety concerns formulated. We also examined the application of Communication Sequential Processes (CSP) as the implementation of FPTN for the purpose of architectural feedback [53]. Although CSP can capture nondeterminism in both a qualitative and quantitative manner [39], as a behaviour modelling language it is inadequate for capturing and evaluating evidence for the purpose of risk assessment. The work outlined in this paper is intended to offer a unified view of our previous work and address the need for risk-based quantitative reasoning through the application of BBNs.

3 Evidence-Oriented Method Construction

We treat design as an iterative and incremental process of producing evidence in which a system-to-be and its domain are *better* understood, *credible* deviation concerns are *exhaustively* identified and *sufficiently* mitigated by design options chosen, as Figure 2 suggests. Consequently, the design process comprises a number of design stages, each representing a cycle of moving from system modelling to deviation modelling and then mitigation modelling. The system model characterises system behaviours in terms of goals and scenarios and system structures with respect to viewpoints. The deviation model features the negative counterparts in terms of anti-goals and negative scenarios. The mitigation model captures possible design space in terms of mitigation actions to help inform decision-making. The proposed Triple-Peaks model is based upon Nuseibeh's Twin Peaks model and elaborates further the interactions between the requirements and architecture models by means of the deviation and mitigation models. From the viewpoint of evolutionary design, co-existent nature of the requirements and architecture models makes it possible to merge them into a single system model, thereby forming the Triple Peaks model.

At every single stage, appropriate evidence should be provided to justify the 'state' of the design progress – how safe the system-to-be would be given current knowledge and evidence. At York we have developed Goal Structuring Notation (GSN) [27] for communicating safety arguments. The items of evidence and their relationships to safety claims are described in terms of goal structures. Figure 3 shows the principal symbols of GSN. Goals can be refined by the aid of specific strategies. The goal refinement process stops when the goals can be satisfied by evidence available. Modular construction of GSN models is facilitated by the notation of 'Away Goal' and 'Module References' [26]: an away goal is a goal that is not defined (and supported) within the module where it is presented but is instead defined (and supported) in another module; a module reference is simply a goal structure packaged

in a module form. The development of the goal structures should proceed in parallel with the design process and reflect the progress of design. In other words, it is possible to use the goal structures to guide the development process. In GSN terminology, generalised goal structures can be captured by GSN patterns [27].

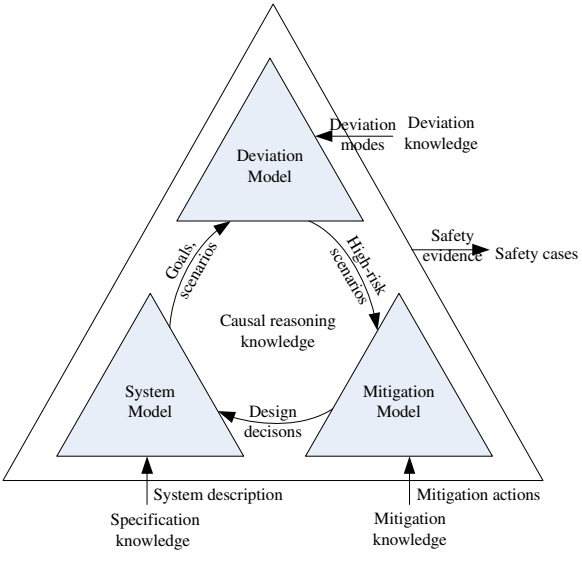


Fig. 2. The Triple-Peaks model

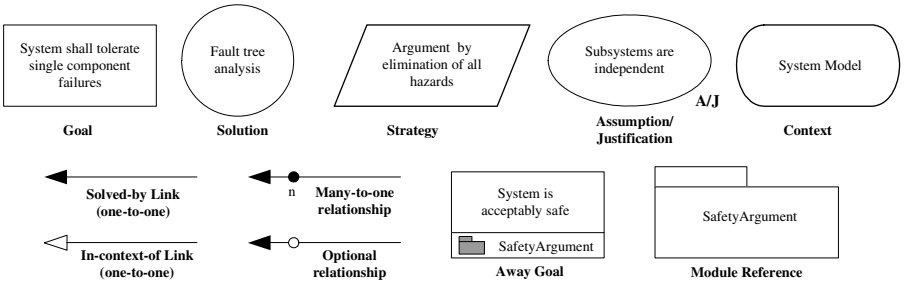


Fig. 3. The principal symbols of GSN

Figure 4 illustrates a sample GSN pattern for describing a system-independent design process when the architect is required to identify all anti-goals and relevant negative scenarios (as described in section 4.4 and 4.5) derived from a single system goal, and choose appropriate avoidance actions to mitigate them. The claims that the negative scenarios could not occur are satisfied by the analysis results of current development process. Justification of the completeness of the anti-goals is based upon the breadth of considerations of deviation modes. Credibility of the negative scenarios identified is evaluated through BBN modelling. By developing GSN patterns tailored

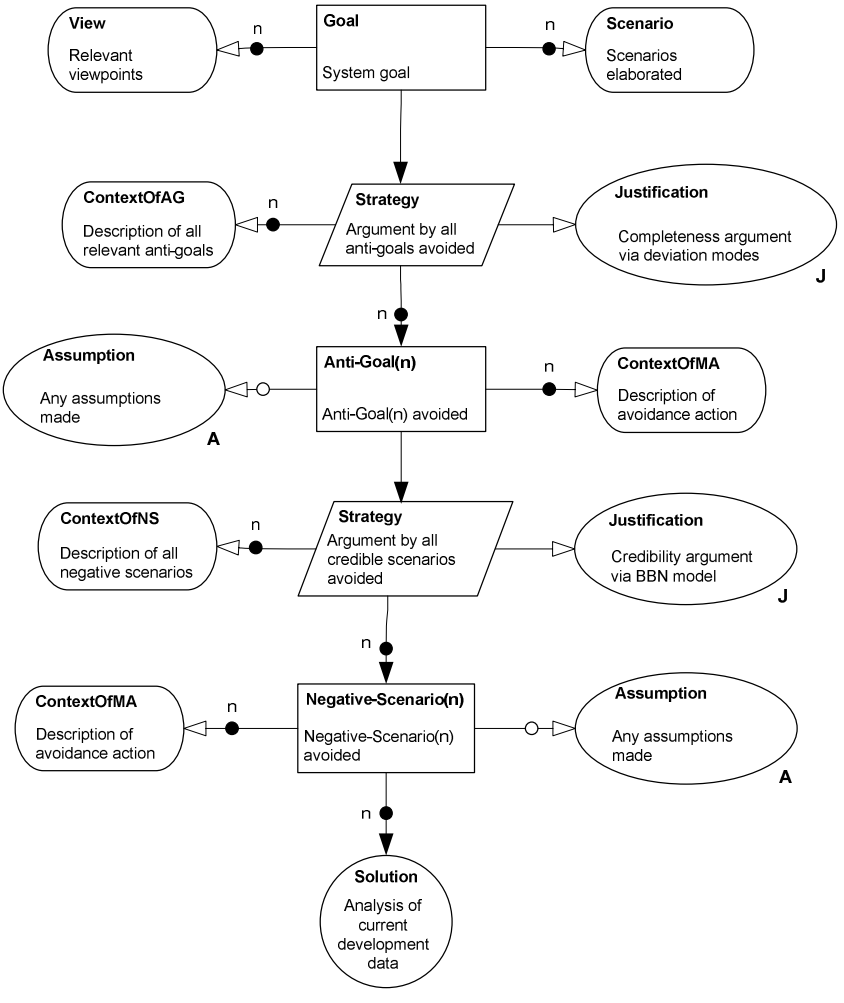


Fig. 4. A simple GSN pattern for the Triple Peaks process

for a specific system domain, system-specific development process model can be instantiated to guide architecting safety-critical software applications.

The proposed Triple Peaks framework represents much of the existing, but implicit, design practices in the dependability community. In order for the deployment of this framework to be successful, the transitions between the three models must be elaborated. Questions may be raised, for example:

- How do we capture and express the model elements such as goals and anti-goals within this framework?
- How do we reason about the properties of the model elements such as completeness, credibility and sufficiency?

Sections 4 and 5 elaborate the linkage between system model and deviation model, and between deviation model and mitigation model, respectively. The WBS example introduced in ARP 4761 is also used to illustrate our approach.

4 Moving from the System Model to Deviation Model

The system model comprises the description of the system under design. There are three essential elements of the system model within our framework: goals, scenarios and viewpoints. While a system model captures the desired properties of a composite system, the deviation model features the undesirable states of the system. In contrast with goals and scenarios in the system model, there are anti-goals and negative scenarios in the deviation model.

4.1 Goals

A goal is an objective that a composite system should meet. Through seeking goals explicitly from the requirements specification, the *core* system functionality and qualities can be effectively elicited and justified. Despite the diverse forms of goal formulation techniques (as discussed in section 2), there are four common elements of a goal we found:

- *Artefact.* The artefact is the composite system or its parts onto which a goal is applied.
- *Context.* The context are the pre-conditions that a goal refers to and evolves over.
- *Stimulus.* The stimulus is the trigger condition for the initiation of a goal.
- *Response.* The response captures the desired properties (i.e., postconditions) that the artefact should hold over time. Quality constraints (e.g., deadline or failure rate) can be specified in this part if they exist.

A sample goal can thus be expressed in the following form:

```
"The <artefact> shall <respond> upon <stimulus> when
<context>"
```

This goal formulation is consistent with the SEI's quality attribute scenario framework and thus can be applicable to both functional and quality goals. As an example, consider the wheel braking system (WBS) of an aircraft [1]. We assume there are a number of top-level goals that can be stated in terms of aircraft functionality (e.g., controlling the aircraft on ground in this case) and qualities (e.g., safety). Each goal can be formulated in the stimulus-response form in spite of their high level of abstraction. Each functional goal can be decomposed further into a set of sub-goals and should evolve separately given that they are independent. The goal decomposition may be guided by the use of scenarios, as described in the next subsection. Goal structures can thus be constructed. Safety goals cannot simply be decomposed via functional goals or system structures; their refinement is based upon the results of deviation analysis (see sections 4.4 and 4.5) and the chosen mitigation. For example, deviation analysis may reveal that a 'late' output of a controller is safety

significant. A performance goal is thus derived and added into the safety goal structures.

Figure 5 shows a part of the goal structure in which the core functionality of WBS is elicited. All the goals in this structure are expressed using the above form. The expression language used is a structured natural language, and some expression can be very abstract at this level. For example, both the stimulus and response parts of the top-level goal *FnG1* are very general and need to be refined. This should be acceptable, however, in the early development lifecycle in which many requirements are volatile and unclear. Figure 6 shows the undeveloped goal structure of the aircraft-level safety goal. In most cases, the top-level safety goal (i.e., *SafeG1*) is simply derived from the certification authority. This achievement of this root goal is based upon the satisfaction of all the supporting safety-related functional and quality goals. Likewise, expressions of safety goals can be very abstract, as the concrete forms of deviations and mitigations are still unknown. It can be seen that the modular features of GSN makes it feasible to isolate development of different goal structures (e.g., functional and safety goal structures) and link them effectively via the notation of Away Goal and Module References.

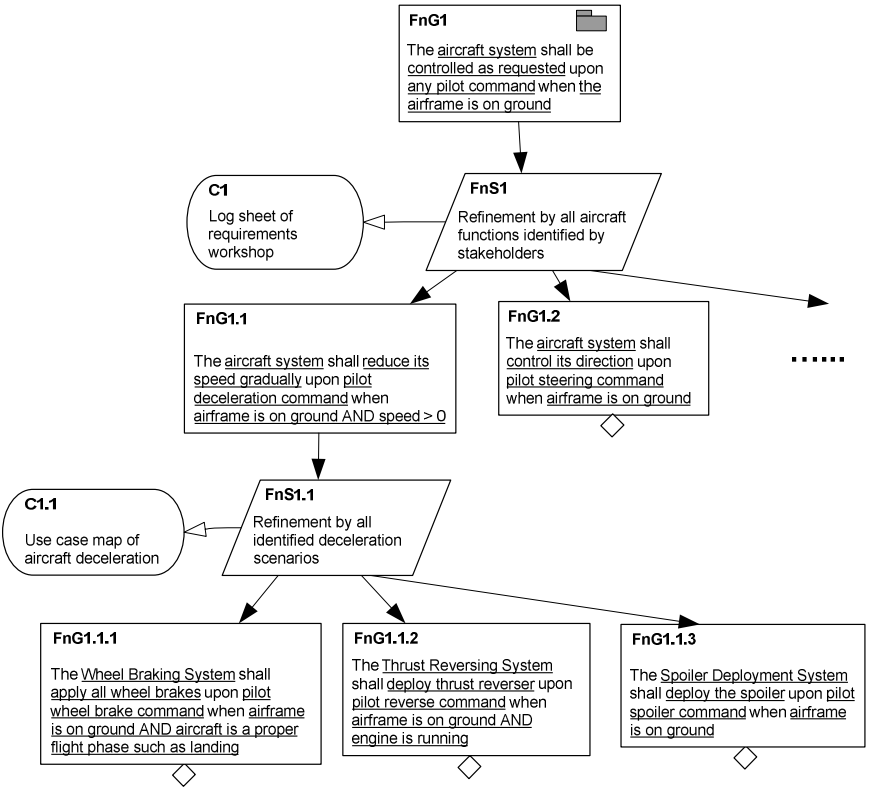


Fig. 5. A functional goal structure for the WBS example – Control the aircraft on ground

4.2 Scenarios

A scenario is a sequence of actions performed by objects instantiated within the known structure of the composite system. Scenarios provide an effective way to elaborate a goal in a white-box view. In other words, a goal defines a set of possible scenarios; a scenario defines a possible realisation of a goal. There are many forms of scenario formulation (see section 2). The use of UCMs is preferable based upon our experience, as it offers a clear-cut notation of causal relationships between architectural components in terms of responsibility points [14]. This is beneficial when we conduct deviation analysis over scenarios, as described in section 4.5. UCMs can be refined, as the underlying system structure is developed in more detail.

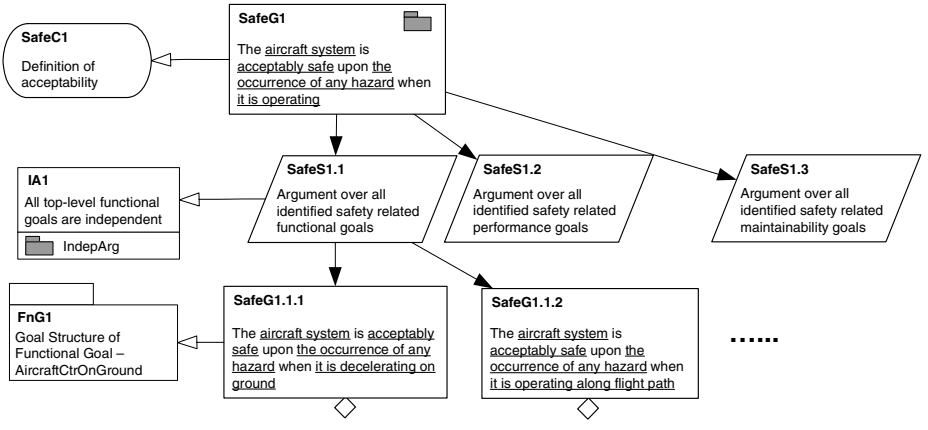


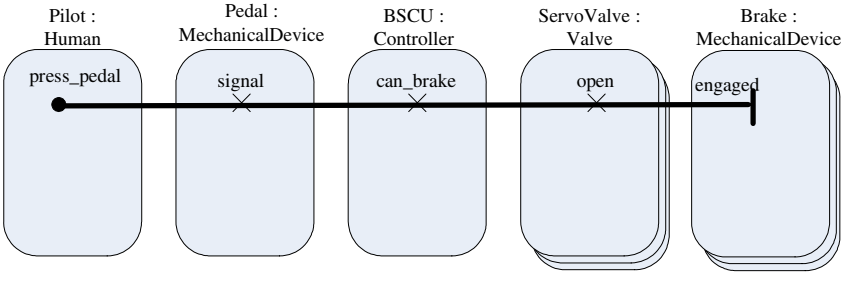
Fig. 6. The top-level safety goal structure for the WBS example

In order to define a scenario (i.e., a UCM model), a goal must be provided. A system structure (i.e., *a component context diagram* in UCM terminology) must also be defined as the context of elaborating the goal. The system structure should be derived from the predefined viewpoints as described in the next subsection. The system structures should be independent of the requirements allocated. This is another benefit of using UCMs, as component context diagrams and use case paths can be developed separately. For the WBS example, the wheel braking goal has been identified in Figure 5. The system architecture of the WBS has been defined in the system description (see [1]) as the input of the design process. Therefore we can elaborate the wheel braking goal through the system architecture. Figure 7 illustrates an example scenario for the wheel braking goal – manual braking in normal mode. As all relevant scenarios (in this example, manual braking in normal mode/alternative mode/emergent mode and auto braking) are elicited for a specific goal, the goal can be decomposed further given the responsibility points allocated. For instance, a sub-goal of the WBS goal will be the goal of BSCU expressed as follows:

The BSCU controller shall output the brake command upon arrival of pedal signal when airframe is on ground AND aircraft is in landing/taxing/RTO flight phase.

4.3 Viewpoints

In most cases, a system model has multiple structures due to the increasing size, complexity and heterogeneity of modern software systems [42]. In practice, these structures are classified in terms of viewpoints. An instance of a viewpoint is called a view (i.e., a system structure). Yet there is no consensus on the number of appropriate viewpoints in both research and practice that are considered necessary to describe software architectures adequately. From the viewpoint of embedded systems development, we define five essential viewpoints, as shown in Table 1. The first two are defined at system level to capture system boundaries and its physical structures. The remaining three viewpoints are defined at software level and consistent with common viewpoint approaches in the software community (e.g., SEI's three viewtypes [15]). The recognition of multiple viewpoints has a significant impact on the completeness of deviation analysis, as viewpoints are interconnected and deviation arising from one view can propagate through another view (see section 4.5).



Preconditions:
- Aircraft is on ground
- Aircraft is moving
- Aircraft is in landing/takeoff/rejected takeoff phase

Postconditions:
- Eight wheel brakes are applied

Fig. 7. An example scenario – manual braking in a normal mode

Table 1. The information description of the five viewpoints

Viewpoint	Description	Intent
Contextual Viewpoint	How an embedded system interacts with its operating environment	To reason about the environmental properties of the system
System Architecture Viewpoint	How an embedded system is structured in terms of physical units. At least one of these units should be the controller or software	To reason about the physical characteristics of the system
Development Viewpoint	How the system's software is structured in terms of implementation units	To reason about the software functions and maintenance
Run-Time Viewpoint	How the system's software is structured in terms of run-time units	To reason about the runtime behaviours of the software
Allocation Viewpoint	How the system's software is allocated onto non-software structures (e.g., hardware platform)	To reason about the impacts of the underlying hardware platform and development environment on software

4.4 Anti-goals

An anti-goal is a condition that, if true, would immediately prevent the composite system from achieving the corresponding goal. Both goals and anti-goals are complementary and thus capture the possible desired and undesired end states of a composite system respectively. A common example of an anti-goal is the loss of a system function where the function was a goal. Nevertheless, simple negation of a goal in terms of propositional logic cannot guarantee the *completeness* of the corresponding anti-goals. A less obvious but perhaps more severe anti-goal would be inadvertent application of that function. Given some goal formulation, it is important to ensure the exhaustiveness of deviations from that goal, at least from the viewpoint of safety. In the safety community, the possible deviations of a system are often characterised in terms of deviation or failure modes. Previous York's work has developed a collection of deviation modes for software systems: SHARD guidewords [19]. We interpret the SHARD modes with respect to the goal formulation in the following Table 2.

Table 2. The anti-goal interpretation using SHARD guidewords

SHARD	Anti-Goal Interpretation
Omission	Response part does not hold while stimulus and environment parts hold
Commission	Stimulus or context parts do not hold while response part holds
Timing	Timing constraint specified in the response part is violated while the other parts hold
Value	Value constraint specified in the response part (e.g., accuracy or cost) is violated while the other parts hold

By allocating the SHARD modes onto a formulated goal and interpreting them using the above table, there can exist a high level of confidence on the exhaustiveness of the set of anti-goals elicited. Notably, not all anti-goals can have safety implications; anti-goals must be evaluated with respect to safety consequences (see section 5.1). Let us return to the WBS example. As soon as the system goals of WBS are formulated, the identification of anti-goals can start by considering the SHARD deviations first without information about the elaborated scenarios. In this example, only omission and commission modes are applicable. Table 3 illustrates an example anti-goal by negating the context part – wheel braking when the context is not as intended. The definition of the stimulus part is trivial in this case. The anti-goal elicited is abstract.

Table 3. An example anti-goal formulation

Portion of Goal	Possible Value
Artefact	WBS
Context	NOT (Airframe is on ground AND aircraft is in landing/taxiing/RTO flight phase)
Stimulus	N/A
Response	All wheel brakes are applied

By expanding the negation operation on the context part using Boolean logic, we can derive a set of well-refined anti-goals: e.g., wheel brakes applied when aircraft is taking off or when aircraft is in air. It must be stressed that the expansion here cannot

be achieved solely by formal Boolean logic and in many cases may need the help of domain experts. For the example of inadvertent wheel braking when the aircraft is taking off, we may need to distinguish further whether the aircraft is taking off before the decision speed *V1*, as the corresponding safety consequences would be different [1]. Obviously, this is impossible for formal logic alone to identify the two anti-goals. When all anti-goals are identified and refined (say, eight anti-goals for the WBS example), they should be linked to the anti-goals of the parent goal of the WBS (i.e., aircraft deceleration) in a bottom up manner, thereby forming an anti-goal structure. The anti-goal structure in the WBS example is shown in Figure 8.

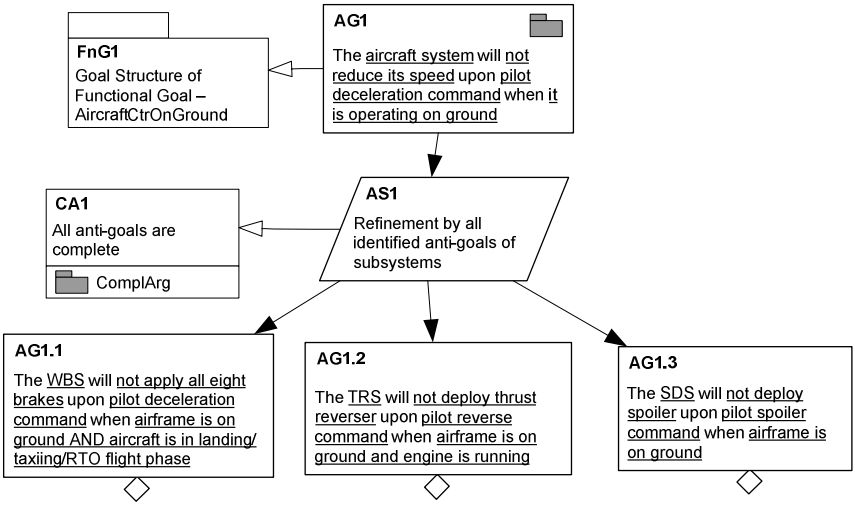


Fig. 8. The anti-goal structure – Total loss of aircraft deceleration

As seen in the Figure 8, the anti-goal structure should refer to the corresponding functional goal structure. It should be noted that the expression languages used in functional goals, safety goals and anti-goals are slightly different. The functional goals are simply requirements and thus 'shall' statements are suitable; the safety goals are claims in which 'is/are' statements are applicable; the anti-goals are hypotheses about states & events of the system and thus 'will' statements should be used. The construction of anti-goal structures will prompt the refinement of the safety-goal structure in which decisions need to be made regarding how to mitigate these identified anti-goals.

4.5 Negative Scenarios

Like a (positive) scenario, a negative scenario is a possible realisation of an anti-goal with respect to the known system structure. In other words, negative scenarios can be formulated using conventional scenario formulation techniques such as use case templates. We previously proposed a stimulus-effect form [54] to emphasise the status of causal propagation within a negative scenario. Not surprisingly, it can be seen as the negation of the stimulus-response form of goals and positive scenarios. To

ensure an exhaustive set of negative scenarios elicited, a broad spectrum of candidate stimuli must be considered. This spectrum should be based upon consideration of all *possible* deviations. Credibility of these deviations will be discussed in section 5.1. We here distinguish two classes of negative scenarios, which are consistent with the decomposition principle of fault tree construction [51]:

- *Primary negative scenarios.* Those stimuli are identified from deviation analysis over a positive scenario — a UCM model which is derived from a specific viewpoint. This is often done by inductive analysis or "what-if" questions conducted on every single action of a scenario.
- *Supporting negative scenarios.* Those stimuli are identified from deviation analysis over other viewpoints but with *possible* contribution to primary negative scenarios. For example, the failure of a run-time component in a runtime view can be caused by malfunction of underlying hardware platform in the deployment view.

Furthermore, the systematic identification of anti-goals offers an effective means of ensuring a broad spectrum of the end effects of the negative scenarios. By linking the candidate stimuli with anti-goals through the stimulus-effect framework, the exhaustiveness of negative scenarios can be justified. New anti-goals may also be identified during negative scenario development. Another concern is the effectiveness of the negative scenario elicitation. In practice, deviation analysis is aided by a set of pre-defined component deviation modes. For example, the common deviation modes for the components of valve type can be: stuck open, stuck close and leakage. Automated analysis is thus possible by extending system structures with component failure modes defined in the deviation knowledge base.

Let us continue the WBS example in which a wheel braking goal, four positive scenarios in a form of UCMs, and eight anti-goals have been defined. Now we need to perform deviation analysis over each of the four positive scenarios. Guidance on deviation analysis over UCMs can be found in our previous work [52]. Simply put, deviation analysis starts by forward search of each responsibility point across the use case path in order to identify possible primary negative scenarios. For a given responsibility point, deviation modes are allocated with respect to the type of the component in which the responsibility point resides, and the end effect of that deviation is identified along the use case path and linked to the identified anti-goals. The forward search procedure for a specific responsibility point is similar to ETA in which the initiating event is the deviation of that responsibility point and all possible event sequences are analysed along the use case path. Figure 9 illustrates the deviation analysis procedure for the scenario – the manual braking in normal mode.

The output of deviation analysis over the four UCM scenarios are fourteen primary negative scenarios. Supporting scenarios must be identified by deviation analysis over other views. To do this, we need to identify how the components in a UCM model can be mapped onto other views. In the WBS example, the UCM models are defined within the system architecture view, and all other views are still undefined. In this case, the architect needs to make some assumptions such as a uni-processor configuration and single monolithic software module in order for the remaining views to be produced. Put another way, the BSCU controller is first mapped to a single software module in both development and run-time views, and to a processor in the

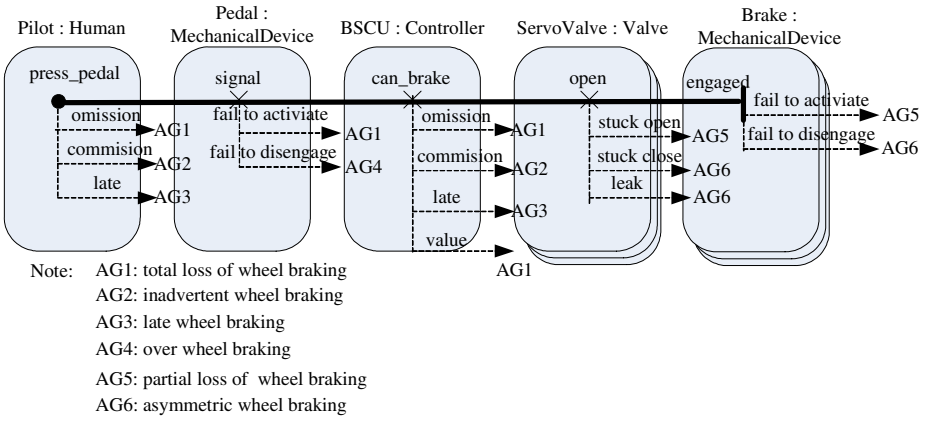


Fig. 9. Deviation analysis over the example scenario (manual wheel braking)

allocation view. Deviation analysis is then performed upon the two components by deviation mode allocation. Consequently, a set of supporting negative scenarios are identified. All identified negative scenarios should be formulated in the stimulus-effect form. Table 4 illustrates an example formulated negative scenario.

Table 4. An example negative scenario formulation

Source	Stimulus	Context	Course of Propagation	End Effect
BSCU	Fails to output brake command	Airframe is on ground AND aircraft is in landing/taxiing/RTO flight phase AND pilot presses the pedal	Eight servo valves, eight brakes	AG1: Total loss of wheel braking

5 Moving from the Deviation Model to Mitigation Model

All the negative scenarios and anti-goals are hypothesised on the basis of current knowledge about the system and its domain. Yet not all anti-goals identified are safety significant. Moreover, it is impractical to completely address all the identified safety concerns within a single design iteration. It must be possible to evaluate the deviation model in terms of safety risks. A negative scenario leading to a safety-significant anti-goal is a risk scenario. Only a small number of high-risk scenarios (say three) will be considered in the mitigation model for every single design iteration. As a result, the management of negative scenarios and assessment of acceptability must be a continuous process through the whole architectural design. The purpose of the mitigation model is to capture plausible mitigation space (i.e. a set of mitigation action candidates) against the high-risk scenarios identified through severity and credibility estimation. Cost-benefit analysis and design tradeoffs may be performed in order to make optimal decisions. The following subsections will describe our solutions to evaluating the deviation model using a BBN framework, identifying mitigation space and performing safety design tradeoffs.

5.1 Severity and Credibility

Like most risk assessment methods, the evaluation is performed in terms of severity and credibility estimation. Notably, we prefer the term "credibility" to the standard term "likelihood" or "frequency", because the former is more consistent with the Bayesian interpretation of probability [25] that lies at the heart of our risk assessment framework. Severity estimation is conducted by considering the safety consequences of all identified anti-goals. Estimation proceeds from the anti-goal structures and takes into account of the contribution to their parent anti-goals if they exist. For the WBS example, the occurrence of the anti-goal *AG1.1* does not necessarily lead to the occurrence of its parent anti-goal *AG 1* unless the anti-goals *AG 1.2* and *AG 1.3* also hold, as shown in Figure 8. In fact, the anti-goal *AG 1.1* should be detectable by the pilot when the wheel braking is commanded and the pilot will be able to use spoiler and thrust reversal to the maximum extent possible in order to achieve deceleration. Hence, the severity classification of the anti-goal *AG1.1* should be hazardous rather than catastrophic.

Credibility estimation should be conducted over all the negative scenarios identified and formulated in a stimulus-effect form. There are two parts to be estimated: the stimulus and propagation parts – how credible is it for the occurrence of the hypothesised stimulus and its *propagation* to manifest the anti-goal? At the beginning of architectural design, it is plausible to make the worse-case assumption that the propagation is completely credible (i.e., its credibility is 1) given that no mitigation mechanisms are employed. Once mitigation actions are chosen against the propagation, the credibility of the propagation should be re-estimated with respect to the effectiveness of the chosen mitigation. Now our focus would be the stimuli of all negative scenarios. To do so, we first distinguish two classes of stimuli:

- Stimuli of a random nature. The sources of the stimuli will be non-agent objects such as hardware and natural environment.
- Stimuli of a systematic nature. The sources of the stimuli will be agents such as human and software.

For the first class of stimuli, the architect should seek historical data to justify their credibility. For the systematic stimuli, an analytic model should be constructed and evaluated through data collected from the real world. The selection of analytic models depends upon the classification of the stimulus: is it human operation error or software design fault? For the former, further investigation is required to check if it is a slip-related error or mistake-related error [47]. Task network analysis [28] may be chosen to predict the credibility of slip-related errors, for instance. For the software design faults, current design artefacts and the progress of development process must be considered. If fault data determined by testing are available, for example, Rome Laboratory's software reliability prediction models [37] may be suitable. Alternatively, if product metrics (e.g., quality of requirements specification) and process metrics (e.g., competencies of the developers) can be estimated, Fenton's defect prediction model may be applicable. Notably, no model is complete or even representative. One model may work well for a set of certain software, but may be completely off track for other kinds of problems. Assumptions and justification made during the selection procedure must be explicitly identified. Knowledge about historical data, the

classifications of analytic models as well as the applicability rules could be codified in a reasoning knowledgebase for the purpose of automation. If no applicable historical data and analytic models are available, expert judgements would be required.

Let us carry on the WBS example. An assumption of 100% propagation hold for all identified negative scenarios can be made. We then need to type-check the stimulus and source parts of every scenario. Analytic models are then selected for credibility estimation. Table 5 illustrates a portion of credibility estimation results for the WBS example. It should be noted that the results of credibility are by no means precise, as the level of design detail increases. In many cases, the credibility of the scenario will need to be updated as the design process progresses and subsequent design decisions are made upon the source of the stimulus. For example, the BSCU software module will inevitably be decomposed in more finer-grained modules in which the brake control responsibilities will be allocated. The update of evidence is possible to be incorporated within the BBN framework described in the next subsection. It must be stressed that the role of quantification is to prioritise scenarios instead of obtaining precise numerical data – through which dominant scenarios are identified and drive the design decision procedure.

Table 5. Example results of credibility estimation

Stimulus	Classification	Estimation Forms	Assumptions/ Justification	Credibility
Pilot fails to press pedal	Slip-related error	Task network analysis	Reason's human error classification [47]	1.5E-6
Power supply loss	Random failure	Historical data		1E-7
Existence of software fault	Design error	Fenton's defect prediction model [50]	No fault/failure data but some process and project management metrics are available at this stage.	2.5E-3

5.2 Causal Bayesian Modelling

A negative scenario is inherently a causal chain starting from a stimulus and leading to undesired end states (i.e., anti-goals). Through composing all identified negative scenarios together, a causal structure can thus be formed. In BBN terminology, a causal structure C is defined in a directed acyclic graph (DAG) form: $C = (V, E)$, where V is defined as a set of nodes, and E is defined as a set of directed edges among V. We distinguish further between two subsets of V: $V1$ and $V2$, where $V1$ corresponding to the set of all the leaf nodes, $V2$ represents the set of the remaining nodes such that $V = V1 \cup V2$ and $V1 \cap V2 = \varnothing$. Each element of the set $V1$ corresponds to a *distinct* anti-goal identified, whilst each element of the set $V2$ corresponds to a *distinct* architectural component identified from the source of stimulus and propagation parts in the negative scenario framework. Each element of E captures a *distinct* causal relation identified by knowledge about the sequence of propagation of a stimulus as specified in the negative scenario framework. Figure 10 illustrates a portion of a causal structure for the WBS example. To simplify the BBN computation, we remove the nodes *WheelBrake(i)* and *ServoValve(i)*, as our main focus here is the controller BSCU.

To form a causal model, the first step is to transfer all the nodes in *V* into variables in which their value domains must be defined. For elements in *V1*, their value domain will be simply Boolean true and false to indicate if an anti-goal occurs or not. For elements in *V2*, their value domains will be their deviation modes allocated and the normal mode (say, ok) during deviation analysis, as described in section 4.5. Finally, we need to define a conditional probability table (CPT) for each variable with respect to the credibility estimation results. Often, this would be a tedious step. However, many BBN tools such as Netica allow us to define probability table by equation. Figure 11 shows the equation definition for the BSCU variable using Netica expression language. In general, careful justification is required when defining a CPT. These judgements need to be captured in the safety arguments (i.e., GSN forms). The following are a number of rules learnt from our experience:

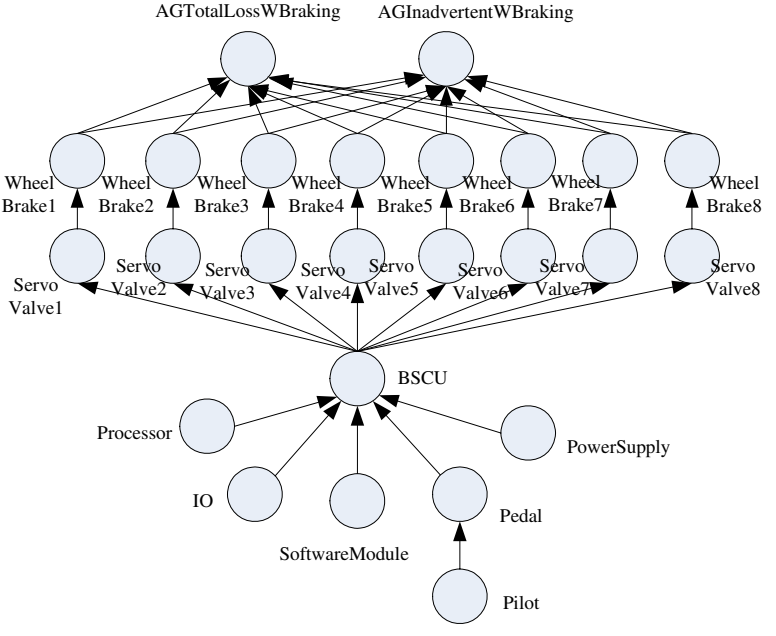


Fig. 10. The causal structure for WBS example

- During the elicitation of negative scenarios, a single deviation is considered as the stimulus for one scenario. But when composing scenarios to form a causal model, the occurrence of multiple deviations must be taken into account. For the WBS example, what if both power supply loss and processor failure happen simultaneously? Obviously, the effect of process failure will not be exhibited by the occurrence of power supply loss, thereby leading to no output of BSCU.
- Some deviations such as transient failure of processor or software faults may have multiple effects in a non-deterministic manner [53]. In those cases, the architect needs to make some assumptions: e.g., all chances are equal.

```
P (BSCU | CPU, IO, SW, Pedal, Power) =
(Power == ok && CPU == ok && Pedal == ok && SW == ok && IO == ok) ? (BSCU == ok ? 1.0 :
0.0) :
(Power == loss) ?  (BSCU == fail_to_output_brake ? 1.0 : 0.0) :
(Power == ok && Pedal == fail_to_activate) ? (BSCU == fail_to_output_brake ? 1.0 : 0.0) :
(Power == ok && Pedal == fail_to_disengage) ? (BSCU == ok ? 1.0 : 0.0) :
(Power == ok && Pedal == activate_inadvert) ? (BSCU == output_brake_inadvert ? 1.0 : 0.0) :
(Power == ok && Pedal == ok && CPU == crash) ? (BSCU == fail_to_output_brake ? 1.0 : 0.0) :
(Power == ok && Pedal == ok && IO == crash) ? (BSCU == fail_to_output_brake ? 1.0 : 0.0) :
 (Power == ok && Pedal == ok && IO == transient_failure) ?
 (BSCU == fail_to_output_brake ? 0.3 : BSCU == output_brake_late ? 0.5 : BSCU == ok ? 0.1 :  0.1) :
 (Power == ok && Pedal == ok && (CPU == transient_failure || SW == faulty))?
    (BSCU == fail_to_output_brake ? 0.3 : BSCU == output_brake_inadvert ? 0.3 : BSCU ==
output_brake_late ? 0.2 : 0.2) : 0
```

Fig. 11. The equation expression for the BSCU variable

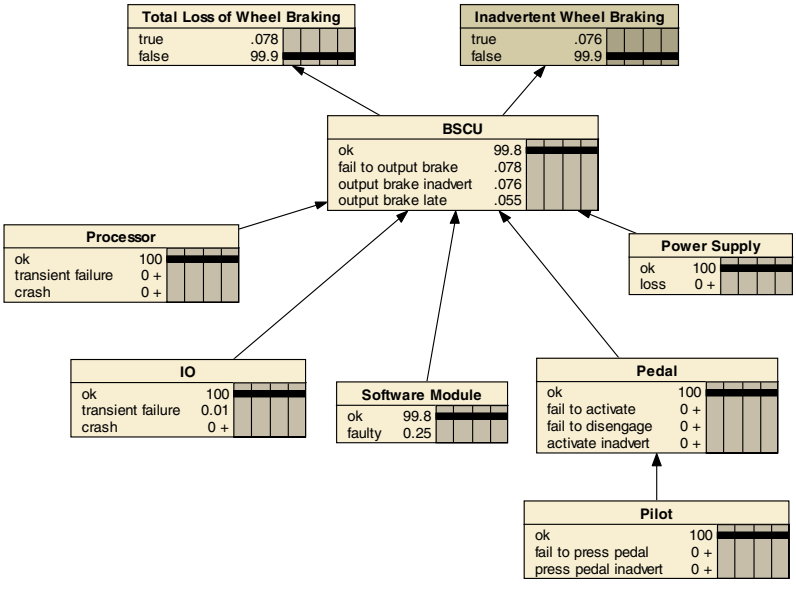


Fig. 12. The BBN evaluation results using Netica

- Like programming, comments for each statement (though not shown in the figure) will be provided to enhance understanding and readability.

Figure 12 shows a fragment of the compiled BBN model produced by Netica. Note that the belief numbers shown in the figure are based upon percentage and some numbers are not shown completely (i.e., 0+) due to limitations of display. For example, the probability of the anti-goal "Total Loss of Wheel Braking" calculated is 0.0007846, which falls short of civil aviation target 1E-7. Therefore, mitigation is required. To do so, we need to identify the high-risk scenarios. This can be done

automatically by sensitivity analysis through Netica. There are three high-risk scenarios, which must be considered in the mitigation model:

S1. Residual faults in the BSCU software leading to malfunctions of the BSCU
S2. I/O (transient or permanent) failures leading to malfunctions of the BSCU
S3. Processor (transient or permanent) failures leading to malfunctions of the BSCU

5.3 Mitigation Space and Safety Tradeoffs

We previously developed a mitigation action model in which mitigation actions are organised in a tree form of five branches (i.e., *elimination*, *reduction*, *detection*, *resistance* and *minimisation*) and codified by a template, which can be implemented in a mitigation knowledgebase [54]. If we treat all the codified mitigation actions as the whole design space, our concern then lies at how to identify the *most appropriate* subset of these actions with respect to a specific negative scenario. Since negative scenarios are formulated in a form of causal chains, a plausible mitigation space will be defined by means of *controllable* parts of the causal chain for the purpose of stopping the propagation. Therefore, the identification of the mitigation space is an iterative procedure of locating the reachable BBN non-leaf nodes (i.e., architectural components) with respect to a given anti-goal and searching applicable mitigation action branches. For the WBS example, consider the scenario S1 identified in the previous subsection. Clearly, the mitigation options would lie within the *software module* and *BSCU* nodes. Mitigation actions applied to the BSCU can also help address scenarios S2 and S3. We then need to determine which of the five branches of mitigation will be applicable. For instance, the *minimisation* action branch (i.e. minimisation of the effect of total loss of wheel braking) is irrelevant, as there is nothing to act when BSCU fails to output the brake command as requested. The procedure continues until all action candidates are located. Table 6 illustrates the mitigation options for S1, accompanied by their rationale.

Table 6. A mitigation space for the negative scenari S1

ID	Node	Branch	Mitigation	Intent
S1	Soft-ware module	Eliminat-ion	Simplification	Correctness of software design can be verified
		Reduct-ion	Rigorous testing	A amount of faults can be detected by testing
			Following safety standard	The process for high Development Assurance Level (DAL) [5] produces 'better' software
			Functional redundancy	The likelihood of faults in different designs is sufficiently low
	BSCU	Detection	Timeout	No response of the BSCU is assumed to fail
			Comparison	Deviations can be detected in case of discrepancy
			Voting	Deviations can be detected and tolerated in case of discrepancy
		Resistan-ce	Recovery	Any error detected can be fixed
			Reconfiguration	Any error detected can be removed by replacement
			Degradation	Any error detected can be removed by removal of the faulty component

To make decisions in response to the mitigation space identified, cost-benefit-risk analysis must be performed. The benefit of each action candidate is determined in terms of its impact upon the credibility of the scenario that it is intended to address. The cost of each option is estimated in terms of orders of magnitude. Both cost and benefit estimates usually require the aid of domain experts and past experience. The known vulnerabilities and side effects of each option are identified by the use of the codified mitigation knowledge. As an example in Table 6, the effectiveness of deploying functional redundancy to reduce software faults lies at a high degree of diversity between module designs, which may be hard to implement in practice. This can be done by means of tables [54] and the architect is free to choose specific options based upon judicious considerations of the mitigation space. It must be stressed that our method does not make decisions for the architects but aids them in eliciting and rationalising their design decisions. For the WBS example, we chose the simplification tactic against the dominant scenario S1 because of the limited number of software input, and comparison and reconfiguration tactics against S2 and S3 by assuming a stringent cost budget. Once mitigation actions are chosen, the system model needs to be refined in the following possible ways:

- Add new components or remove existing components in specific view(s).
- Add new responsibilities (a.k.a., derived requirements) in specific view(s).
- Re-allocate existing responsibilities in specific view(s)

For the WBS example, the correctness of the monolithic software module in the development view must be verified. In this case, no refinement of software module is required by this decision. However, non-safety such as modifiability-related design decisions can drive the decomposition of the monolithic module into a control function module and compiler module so that the BSCU software can be portable to different compilers. Two dual processors and buses are introduced in the allocation view and the output of the BSCU is arbitrated, as indicated by the chosen comparison tactic. Transient processor/bus failures can be repaired by rebooting the processor and reloading its copy of software. Behaviours regarding the run-time comparison and reboot behaviours must be captured in the scenario forms. At this point, both the structures of functional goals and safety goals can be decomposed further to reflect the refinement of the system model and derivation of safety requirements.

Likewise, a new deviation model will be generated upon the refinement of the system model. In most cases, the step of identification of anti-goals can be skipped unless new system goals (e.g., system monitoring) are identified. The main focus is thus the elicitation of new negative scenarios. For the WBS, example negative scenarios elicited can be failure of both processors and the use of potentially faulty compiler. The process loops until all the core system requirements have been elicited and all identified anti-goals are mitigated sufficiently with respect to the risk acceptance. A stable architecture is therefore formed at the end of the design process.

6 Discussion

We have so far applied the proposed framework to a number of medium-size case studies such as AGV [54] and WBS systems. The WBS example was selected for the

purpose of comparison with the ARP process which has been widely applied in practice. We discuss our results in terms of the following three aspects:

Exhaustiveness. Establishing an argument of exhaustive identification of the design issues (i.e., negative requirements) and design alternatives is recognised as the key to the robustness of dependability design. Within the ARP framework, negative requirements are identified through deviation analysis purely on system functions (i.e., FFA) and refined through FTA during architecture definition. The transitions from FFA to FTA and from system level to software level are undefined, however. The exhaustiveness argument is thus implicit and cannot be validated. The techniques presented in our approach provide semi-formal and multi-viewpoint support for deviation identification: deviation analysis is started at goal level through top-down goal formulation and refined at scenario level and component level through pre-defined architectural viewpoints in a bottom-up manner. Moreover, there is no step for identification and justification of design alternatives in the ARP process. The exhaustiveness argument is inherently limited and subject to the competencies of domain experts and the past experience of architects.

Effectiveness. Most software standards advocate heavyweight development approaches in which the upfront specification of all system requirements is mandatory before the design proceeds [12]. For the ARP example, a sequential 'waterfall-like' process is defined, starting from aircraft-level FFA, system-level FFA to system-level FTA and software-level FTA, from which system and software safety requirements are derived and architectures are validated. This form of waterfall model has been known to be inadequate for handling the volatility and uncertainty commonly involved in the real-world problems. The Triple-Peaks process presented in this paper inspired from the Twin-Peaks model addresses requirements specification, design issues and corresponding design alternatives iteratively and incrementally. The process is receptive to requirements change, as only core system requirements are analysed and achieved by a stable architecture defined. Incremental construction of safety evidence is facilitated by means of GSN in a top-down manner, though it remains to be seen whether the explicit recording of safety arguments is best done in order to 'fake' a rational design process as described by Parnas and Clements [43]. The effectiveness of the design process is also enhanced by available knowledge sources such as deviation modes, component deviation types and mitigation tactics, though there is still lack of tool support available to integrate these techniques.

Rationality. Existing software design approaches often rely upon implicit reasoning, through which design decisions are mainly promoted by design intuition. The linkage between design decisions and requirements is largely undefined. Though the design decisions are clearly identified within the ARP process, the steps moving from the identified safety requirements to design decisions are still unclear. Our approach elaborates the linkage between safety requirements and mitigation options chosen by means of requirements formulation and prioritisation, design space analysis and cost-benefit-risk analysis. With the aid of BBN tools, the notion of credibility can be deployed in design and a level of confidence can thus be established. Deciding the

stopping rules of the design process (i.e., when design issues identified are complete and mitigation is adequate) is based upon risk acceptability criteria.

7 Summary

In this paper, we have presented an integrated approach to architectural design for safety-critical software applications through a Triple Peaks framework. In particular, we have demonstrated that how it is practical to conduct deviation analysis simultaneously at both the requirements and architecture level. The key principle underlying this paper is that software safety evidence must be collected in the early development lifecycle, and architectural design decisions must be informed based upon this evidence & quantitative risk assessment. Our future work includes seeking possible automation of the linkage between BBN models and architectural choices, and expanding the proposed method into other critical domains such as security-critical software applications.

References

1. ARP 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, Society of Automotive Engineers, Inc. (1996)
2. Australian Defence Standard Def(Aust) 5679: Procurement of Computer-based Safety Critical Systems, Australian Department of Defence (1998)
3. IEC 615038 – Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems, International Electrotechnical Commission (1998)
4. Netica, Norsys Software Corp. (2006), http://www.norsys.com/
5. RTCA/DO-178B: Software Considerations in Airborne Systems and Equipment Certification, Radio Technical Commission for Aeronautics (1992)
6. The United Modelling Language (UML) Specification. The Object Management Group (2005)
7. Achour, C.B., Rolland, C., Souveyet, C.: Guiding Goal Modelling Using Scenarios. IEEE Trans. on Software Engineering 24(2), 1055–1071
8. Alexander, I.: Misuse Cases: Use Cases with Hostile Intent. IEEE Software 20(1), 58–66
9. Allenby, K., Kelly, T.: Deriving Safety Requirements using Scenarios. In: the 5th IEEE International Symposium on Requirements Engineering(RE'01), p. 228. IEEE Computer Society Press, Los Alamitos (2001)
10. Bachmann, F., Bass, L., Klein, M.: Deriving Architectural Tactics: A Step Toward Methodical Architectural Design, SEI (2003)
11. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison Wesley, Reading, MA, USA (2003)
12. Boehm, B., Turner, R.: Balancing Agility and Discipline: A Guide for the Perplexed. Addison-Wesley Professional, Reading (2003)
13. Brandozzi, M., Perry, D.E.: From Goal-Oriented Requirements to Architectural Prescriptions: The Preskriptor Process. In: Proceedings of Third International Workshop From SofTware Requirements to Architectures (STRAW'03), pp. 107–113 (2003)
14. Buhr, R.J.A., Casselman, R.S.: Use Case Maps for Object-Oriented Systems. Prentice-Hall, Englewood Cliffs (1996)
15. Clements, P.: Documenting software architectures: views and beyond. Addison-Wesley, Boston (2003)

16. Conklin, J., Begeman, M.L.: gIBIS: A Hypertext Tool for Exploratory Policy Discussion. ACM Transactions on OfficeInformation Systems 6(4), 303–331
17. Easterbrook, S., Lutz, R., Covington, R., Kelly, J., Ampo, Y., Hamilton, D.: Experiences Using Lightweight Formal Methods for Requirements Modeling. IEEE Trans. on Software Engineering 24(1), 4–14
18. Feather, M.S., Cornford, S.L.: Quantitative risk-based requirements reasoning. Requirements Engineering 8(4), 248–265
19. Fenelon, P., McDermid, J., Nicholson, M., Pumfrey, D.: Towards Integrated Safety Analysis and Design. ACM Computing Reviews 2(1), 21–32
20. Galliers, J., Sutcliffe, A., Minocha, S.: An impact analysis method for safety-critical user interface design. ACM Transactions on Computer-Human Interaction (TOCHI) 6(4), 341–369
21. Gregoriades, A., Sutcliffe, A.: Scenario-Based Assessment of Nonfunctional Requirements. IEEE Trans. on Software Engineering 31(5), 392–409
22. Gurp, J.v., Bosch, J.: SAABNet: Managing Qualitative Knowledge in Software Architecture Assessment. In: 7th IEEE International Symposium on Engineering of Computer-Based Systems (ECBS 2000), IEEE Computer Society, Los Alamitos (2000)
23. Hall, J., Jackson, M., Laney, R., Nuseibeh, B., Rapanotti, L.: Relating Software Requirements and Architectures using Problem Frames. In: Proceedings of the 10th International Conference on Requirements Engineering, IEEE Computer Society, Los Alamitos (2002)
24. Jacobson, I., Christerson, M., Jonsson, P., Oevergaard, G.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison Wesley, Reading, Mass (1992)
25. Jaynes, E.T.: Probability Theory: The Logic of Science. Cambridge University Press, Cambridge (2003)
26. Kelly, T.: Using Software Architecture Techniques to Support the Modular Certification of Safety-Critical Systems. In: Proceedings of Eleventh Australian Workshop on Safety-Related Programmable Systems (2006), http://www-users.cs.york.ac.uk/ tpk/scs2006.pdf
27. Kelly, T.P.: Arguing Safety - A Systematic Approach to Safety Case Management Department of Computer Science, DPhil Thesis, University of York, York (1999)
28. Kirwan, B., Ainsworth, L.K. (eds.): A Guide to Task Analysis: The Task Analysis Working Group. Taylor & Francis, Abington (1992)
29. Lamsweerde, A.v.: Elaborating Security Requirements by Construction of Intentional Anti-Models. In: Proceedings of the 26th International Conference on Software Engineering, pp. 148–157. IEEE Computer Society, Los Alamitos (2004)
30. Lamsweerde, A.v.: Goal-Oriented Requirements Engineering: A Guided Tour. In: Lamsweerde, A. (ed.) Proceedings of 5th IEEE International Symposium on Requirements Engineering (RE'01), pp. 249–263. IEEE Press, Los Alamitos (2001)
31. Lamsweerde, A.v., Dardenne, A., Fickas, S.: Goal-directed Requirements Acquisition. Science of Computer Programming 20, 3–50
32. Lamsweerde, A.v., Letier, E.: Integrating Obstacles in Goal-Driven Requirements Engineering. In: Lamsweerde, A. (ed.) Proceedings of the 20th International Conference on Software Engineering, pp. 53–62. IEEE Computer Society Press / ACM Press, Los Alamitos (1998)
33. Lemos, R.d., Saeed, A., Anderson, T.: On the Safety Analysis of Requirements Specifications. In: Proceedings of the 13th International Conference on Computer Safety, Reliability and Security, Instrument Society of America, pp. 217–227 (1994)
34. Leveson, N.G.: Intent Specifications: An Approach to Building Human-Centered Specifications. IEEE Trans. on Software Engineering 26(1), 15–35

35. Leveson, N.G.: Safeware: System Safety and Computers. Addison-Wesley, Reading (1995)
36. Leveson, N.G., Modugno, F., Reese, J.D., Partridge, K., Sandys, S.D.: Integrated Safety Analysis of Requirements Specifications. In: Proceedings: 3rd International Conference on Requirements Engineering (1997)
37. Lyu, M.R. (ed.): Handbook of Software Reliability Engineering. McGraw-Hill, New York (1996)
38. McDermid, J.A.: Software Safety: Where's the Evidence? In: McDermid, J.A. (ed.) The 6th Australian Workshop on IndustrialExperience with Safety Critical Systems and Software (SCS'01) (Brisbane, 2001), Australian Computer Society (2001)
39. Morgan, C.: Of Probabilistic Wp and SP-and Compositionality. In: Symposium on the Occasion of 25 Years of CSP (London, 2004), pp. 220–241. Springer, Heidelberg (2004)
40. Mylopoulos, J., Chung, L.: B.N. Representing and Using Non-Functional Requirements:A Process-Oriented Approach. IEEE Trans. on Software Engineering 18(6), 497–497
41. Nuseibeh, B.: Weaving Together Requirements and Architectures. IEEE Computer 34(3), 115–114
42. Nuseibeh, B., Kramer, J., Finkelstein, A.: Expressing the relationships between multiple views in requirements specification. In: Proceedings of the 15th international conference on Software Engineering, pp. 187–196. IEEE Computer Society Press, Los Alamitos (1993)
43. Parnas, D.L., Clements, P.C.A: rational design process: How and why to fake it. IEEE Trans. on Software Engineering 12(2), 251–257
44. Pearl, J.: Causality: models, reasoning, and inference. Cambridge University Press, Cambridge (2000)
45. Potts, C., Antón, A.I.: Inquiry-based Requirements Analysis. IEEE Software. 21–32.
46. Ramesh, B., Dhar, V.: Supporting systems development by capturing deliberations during requirements engineering. IEEE Trans. on Software Engineering 18(6), 498–510
47. Reason, J.: Human Error. Cambridge University Press, Cambridge (1990)
48. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Englewood Cliffs (1996)
49. Stamatelatos, M., Apostolakis, G., Dezfuli, H., Everline, C., Guarro, S., Moieni, P., Mosleh, A., Paulos, T., Youngblood, R.: Probabilistic Risk Assessment Procedures Guide for NASA Managers and Practitioners, NASA Office of Safety and Mission Assurance (2002)
50. The SERENE Partners: CSR, E., ERA, OT, TUV. The SERENE Method Manual SafEty and Risk Evaluation using bayesian NEts: SERENE, ERA Technology Ltd. (1999)
51. Vesely, W.E.: Fault Tree Handbook. Nuclear Regulatory Commission (1987)
52. Wu, W., Kelly, T.: Deriving Safety Requirements as Part of System Architecture Definition. In: Proceedings of 24th International System Safety Conference, System Safety Society (2006)
53. Wu, W., Kelly, T.: Failure Modelling in Software Architecture Design for Safety. SIGSOFT Softw. Eng. Notes 30(4), 1–7
54. Wu, W., Kelly, T.: Managing Architectural Design Decisions for Safety-Critical Software Systems. In: Proceedings of the 2nd International Conference on the Quality of Software Architectures, Springer, Heidelberg (2006)
55. Wu, W., Kelly, T.: Safety Tactics for Software Architecture Design. In: Proceedings of the 28th International Computer Software and Applications Conference, IEEE Computer Society, Los Alamitos (2004)