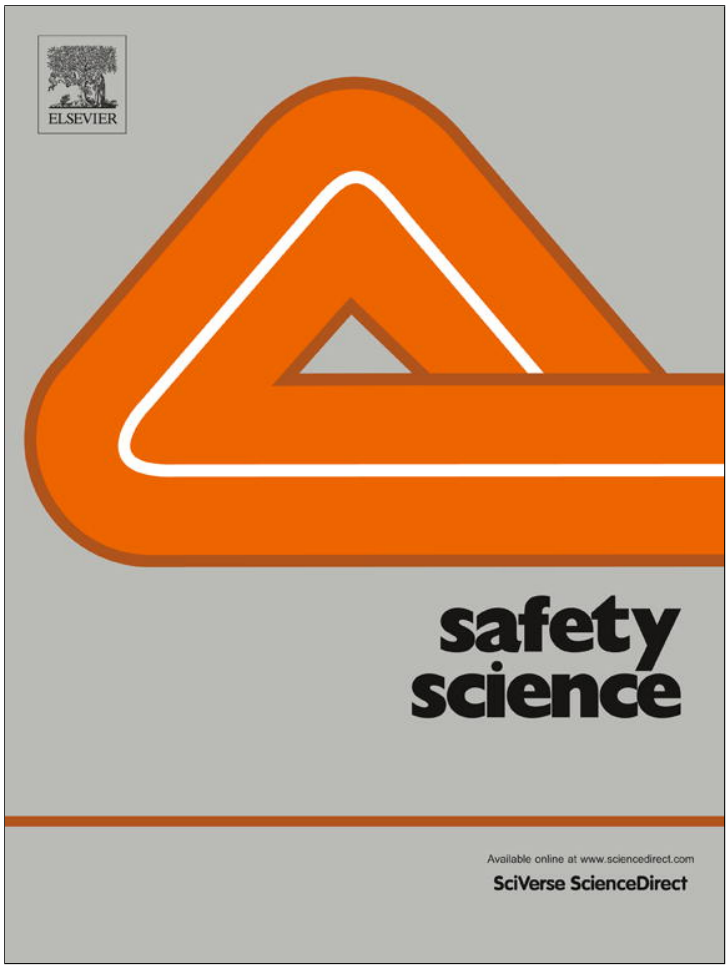

Safety Science 59 (2013) 55–71

Contents lists available at SciVerse ScienceDirect

Safety Science

journal homepage: www.elsevier.com/locate/ssci

Assurance cases and prescriptive software safety certification: A comparative study

Richard Hawkins *, Ibrahim Habli, Tim Kelly, John McDermid

Department of Computer Science, The University of York, York YO10 5GH, UK

ARTICLE INFO

Article history:
Received 28 September 2012
Received in revised form 26 February 2013
Accepted 8 April 2013

Keywords:
Software safety
Assurance cases
Safety cases
Certification
DO178C

ABSTRACT

In safety–critical applications, it is necessary to justify, prior to deployment, why software behaviour is to be trusted. This is normally referred to as software safety assurance. Within certification standards, developers demonstrate this by appealing to the satisfaction of objectives that the safety assurance standards require for compliance. In some standards the objectives can be very detailed in nature, prescribing specific processes and techniques that must be followed. This approach to certification is often described as prescriptive or process-based certification. Other standards set out much more high-level objectives and are less prescriptive about the particular processes and techniques to be used. These standards instead explicitly require the submission of an assurance argument which communicates how evidence, generated during development (for example from testing, analysis and review) satisfies claims concerning the safety of the software. There has been much debate surrounding the relative merits of prescriptive and safety assurance argument approaches to certification. In many ways this debate can lead to confusion. There can in fact be seen to be a role for both approaches in a successful software assurance regime. In this paper, we provide a comparative examination of these two approaches, and seek to identify the relative merits of each. We first introduce the concepts of assurance cases and prescriptive software assurance. We describe how an assurance case could be generated for the software of an aircraft wheel braking system. We then describe how prescriptive certification guidelines could be used in order to gain assurance in the same system. Finally, we compare the results of the two approaches and explain how these approaches may complement each other. This comparison highlights the crucial role that an assurance argument can play in explaining and justifying how the software evidence supports the assurance argument, even when a prescriptive safety standard is being followed.

1. Introduction

Software safety assurance is often demonstrated by compliance with national or international safety standards. Compliance with standards aims to provide assurance that the software functions are performed with a level of confidence which is proportionate to the safety criticality of those functions (RTCA, 2012; IEC, 2010; MoD, 2007). That is, high levels of confidence have to be achieved for software-based functions whose failures could lead to harm, i.e. death, injury or damage to property or the environment in their system context. Within existing safety certification standards, typically developers demonstrate that a software system is acceptably safe by appealing to the satisfaction of a set of objectives that the safety standards require for compliance. In some standards the objectives can be very detailed in nature, prescribing specific processes and techniques that must be followed. This approach to certification is often described as prescriptive or process-based certification. Different sets of objectives are normally associated with different safety integrity or assurance levels. These levels are assigned according to the level of risk associated with the hazards to which the software can contribute. The means for satisfying these objectives are often tightly defined within the prescriptive standards, leaving little room for developers to apply alternatives means for compliance which might better suit their software products and processes. On the other hand, goal-based standards set out higher level objectives and explicitly require the submission of an argument which communicates how evidence, generated from testing, analysis and review, supports claims concerning the safety of the software functions. The argument and evidence within this approach are often presented in the form of a safety or assurance case. There are some concerns regarding the adequacy of the guidance available for the creation of assurance arguments which comply with the objectives set within these standards (i.e. lack of sufficient worked examples of arguments, sample means for generating evidence and guidance on how to assess assurance cases) (Penny et al., 2001; Habli et al., 2010).

Recently, the requirement for a safety and assurance case has been considered in emerging national and international standards.

* Corresponding author. Tel.: +44 01904 4325463.
E-mail addresses: richard.hawkins@york.ac.uk (R. Hawkins), ibrahim.habli@york.ac.uk (I. Habli), tim.kelly@york.ac.uk (T. Kelly), john.mcdermid@cs.york.ac.uk (J. McDermid).

0925-7535/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.ssci.2013.04.007

For example, a safety case is required for compliance with the new automotive functional safety standard ISO 26262 (ISO, 2011). The development of a software assurance case is also a recommended practice in the US Food and Drug Administration's draft guidance on the production of infusion pump systems (FDA, 2010). As a consequence, there is an increased interest in understanding how safety case development compares to, and possibly complements, traditional means for safety compliance by satisfying prescriptive guidelines.

The hypothesis in this paper is as follows: assurance cases and compliance with prescriptive standards are complementary for providing a reasoned justification concerning the contributions of software to system safety. In considering this hypothesis, we provide a comparative examination of the above two approaches to software safety assurance. This comparison highlights, based on a specific case study of an aircraft wheel braking system, the advantages and limitations of both assurance case and prescriptive approaches. Our study shows how these approaches complement each other and discuss the role that an assurance argument can play, even when a prescriptive safety standard is being followed.

1.1. Methodology

The research reported in this paper falls under the category of case study research (Yin, 2003), i.e. it aims to provide a deeper understanding of a phenomenon by investigating the phenomenon in its context (Runeson and Höst, 2009). The phenomenon we investigate is safety assurance for software systems. Specifically, the natural context for the software system addressed in this paper is the aircraft, the wheel braking system, its operational environment and relevant certification requirements. Although our comparative case study includes a combination of formal, informal, qualitative and quantitative data, most of the significant findings are presented qualitatively, e.g. in the form of feedback from the engineers and assessors on the defined criteria. Although quantitative data could be generated, such data would fall short of supporting our findings with statistical significance. The integrity of the study and findings is maintained through preserving the chain of evidence between the evaluation of findings (Section 5) and their supporting data (Section 4).

1.2. Comparison criteria

For any safety assurance approach at least three main stakeholder groups can be identified. These are the developers with responsibility for developing and assuring the system, the reviewers who must understand and make judgments as to the sufficiency of the safety assurance achieved (as part of the acceptance task), and managers who have responsibility for ensuring the successful delivery of a certified system. Our examination therefore focuses on a number of criteria relevant to the three main groups of stakeholders. The criteria are:

1. *Clarity*: does the approach produce a comprehensible and well-structured set of documentation of the safety assurance data?
2. *Rationale*: are the reasons as to why the chosen means for assurance achieve confidence in software safety explicitly described and justified?
3. *Amenability to review*: are reviewers provided with sufficient safety assurance data to identify flawed reasoning and untrustworthy evidence?
4. *Predictability*: do the certification guidelines and past experience provide enough indication about what constitutes adequate safety assurance data?
5. *Effort*: what are the cost implications of using the assurance approach?

The clarity and rationale criteria are important for both developers and reviewers. Amenability to review is particularly relevant to reviewers. The predictability and effort criteria are important for managers. These criteria are derived from our experience of reviewing assurance cases and DO178C Software Accomplishment Summaries (SAS), and in providing advice to companies on software processes including adopting novel technologies (Hawkins, 2007; Denney et al., 2012). The clarity criterion is clearly key to any certification approach. The rationale criterion is important when changing processes, or employing new technology. The amenability to review criterion reflects a common problem – reviewers are often presented evidence with insufficient explanation of how it demonstrates safety; the Nimrod review (Cave, 2009) is further evidence to support the relevance of this criterion, which highlighted the need for review methods to assess that sufficient confidence can be placed in safety cases. The predictability and effort criteria are linked. For example, a process may be comparatively cheap, but also very unpredictable, or vice versa.

In order to get further reassurance of the sufficiency of the criteria used, we obtained feedback, through structured questionnaires, from two representatives of regulators and assessment organisations, as well as five developers. The respondents stated that they understood the criteria and that they believed the criteria provide an effective basis for comparison of different approaches.

1.3. Organisation of the paper

We first introduce, in Sections 2 and 3, the concepts of assurance cases and prescriptive software assurance respectively. In Section 4, we describe an aircraft wheel braking system that will be used for the comparison study; this is chosen as it builds on material in one of the most widely used civil aerospace standards (SAE, 1996b). We then illustrate two different assurance approaches for this system. Firstly we describe how a software assurance case could be generated for the example system, in the light of guidance from standards such as the Defence Standard 00-56 (MoD, 2007) and the Civil Aviation Authority standard CAP 670 (CAA, 2007). Secondly we describe how more prescriptive certification guidelines could be used in order to gain assurance for the same system, in the light of the civil aerospace guidance DO178C (RTCA, 2012), and consider other standards which can be viewed as more prescriptive than DO178C. In Section 5 we evaluate the results of applying the two different software assurance approaches to the case study. Section 6 presents the conclusions.

2. Software assurance cases and related work

The concept of assurance is fundamental to any certification or acceptance regime. Certification refers to the "*process of assuring that a product or process has certain stated properties, which are then recorded in a certificate*" (Jackson et al., 2007). Assurance can be defined as justified confidence in a property of interest (Hawkins and Kelly, 2010).

An assurance case attempts to demonstrate that sufficient assurance has been achieved for a system; our focus here is on systems containing software. The concept of assurance cases has been long established in the safety domain, where the term safety case is normally used. The term assurance case refers to the increasing generalization of this approach to include addressing other attributes. For many industries, the development, review and acceptance of a safety case forms a key element of regulatory processes, this includes the nuclear (HSE, 2006) and defence (MoD, 2007) industries as well as civil aviation (CAA, 2007) and rail (Rail Safety and Standards Board, 2007). Safety cases have been defined by Kelly (1998) in the following terms: "*A safety case should*

communicate a clear, comprehensible and defensible argument that a system is acceptably safe to operate in a particular context". The explicit presentation of an argument is fundamental to any safety case as it is used to demonstrate why the reader should conclude that a system is acceptably safe from the evidence available.

More recently there has been increasing interest in the use of assurance cases in other domains, particularly for demonstrating system security (Bloomfield, 2005; Goodenough et al., 2007). The similarities between assurance case concepts in different domains have been highlighted in the SafSec study which explored the similarities between safety and security cases (Dobbing and Lautieri, 2006). There have also been a number of initiatives to standardise the description of assurance cases, particularly by the Object Management Group (OMG) (OMG, 2010), and through the development of the ISO standard 15026 on assurance cases (ISO/IEC, 2010). Whilst these bodies of work vary in detail they agree that, in general, any assurance case must contain a compelling argument supported by strong evidence.

One area where there are potentially large benefits, but also particular challenges in producing assurance cases is for software. The potential benefits of an assurance case approach have been discussed in detail by a number of authors (McDermid, 2001; McDermid and Pumfrey, 2001). However, there has also been seen to be a reluctance to move away from the more prescriptive approach to certification of many of the most widely used software safety and development standards. A major study into the development of a safety case for a military aircraft software system was undertaken by the Industrial Avionics Working Group (IAWG, 2007; Hawkins, 2007). In particular, this looked at the benefits that a safety case approach can bring in facilitating modular and incremental certification, with promising results. Similar results of the use of software assurance cases for medical devices were reported by Weinstock and Goodenough (2009), adding that "*adoption will take buy-in by a significant number of device manufacturers, and for this to happen they must be educated*".

Bloomfield and Bishop (2010) discussed the current practice and uptake of safety and assurance cases for software-based systems. They concluded that, while the application to complex systems is a significant undertaking, the use of assurance cases for software is very appealing, supporting as it does innovation and flexibility. Further, Graydon et al. (2007) proposed an approach to integrating assurance into the development process by co-developing the software system and its assurance case. This approach enables assurance requirements to influence the design, assessment and operation of a critical system from the earliest stages (Lutz and Mikulski, 2003).

It is important for any assurance argument to be reviewed and for its supporting evidence to be audited, prior to acceptance, in order to identify any incorrect reasoning within the argument or limitations in evidence (Kelly, 2007). Greenwell et al. (2006) examined fallacies in assurance arguments for safety–critical systems, i.e. arguments which may seem to be correct, but which prove, on examination, not to be so. They maintained that if an assurance argument is fallacious, the system could be subject to unidentified hazards that could lead to accidents. To this end, engineers have to be able to identify poor reasoning in order to avoid creating fallacious arguments. Likewise, representatives of certification authorities should be familiar with fallacies in order to detect them in the process of reviewing assurance arguments.

Finally, software assurance arguments are predominantly inductive, i.e. offering support for the top-level claim which is short of certainty. They typically fall within the scope of informal logic, compared to formal and mathematical logic which is based on mathematical semantics and theory. However, there has been increasing interest in the formalisation of parts of the assurance argument (Littlewood and Wright, 2007; Habli and Kelly, 2008b; Basir et al., 2008, 2010; Rushby, 2010).

3. Prescriptive software assurance and the DO178C guidance

This section introduces prescriptive certification in the context of the civil aerospace guidance DO178C, titled: *Software Considerations in Airborne Systems and Equipment Certification* (RTCA, 2012). DO178C has been chosen as our 'exemplar' prescriptive standard because it has a long and extensive history of use. It is also the most appropriate standard for the domain of our chosen case study. Other standards and guidelines that are often categorised as prescriptive (McDermid and Pumfrey, 2001) include IEC 61508 (IEC, 2010), ISO 26262 (ISO, 2011).

The purpose of the DO178C document is "to provide guidance for the production of software for airborne systems and equipment that performs its intended function with a level of confidence in safety that complies with airworthiness requirements". The guidance specifies certification requirements (i.e. in the form of objectives), means for achieving these requirements and acceptable forms of evidence (typically in the form of testing, analysis and review data). DO178C defines a consensus of the aerospace community concerning the approval of airborne software. To obtain certification credit, developers submit lifecycle plans and data that show that the production of the software has been performed as prescribed by the DO178C guidance. Any deviation or alternative means for compliance should be justified in the Plan for Software Aspects of Certification (PSAC) and Software Accomplishment Summary (SAS). As a result, the norm in the development of civil airborne software is to use the guidance (i.e. means for compliance) as prescribed in the DO178C document. This avoids the challenge of, and potential project delays due to, the need to justify the deployment of alternative techniques. It also reduces the risk of not achieving certification due to non-compliance.

The DO178C guidance distinguishes between different levels of assurance based on the safety criticality of software, i.e. how software components contribute to system hazards. The safety criticality of software is determined at the system level during the system safety assessment process based on failure conditions associated with software components. These failure conditions are categorised as follows:

- Catastrophic.
- Hazardous/Severe-Major.
- Major.
- Minor.
- No Effect.

The DO178C guidance defines five different assurance levels which relate to the above categorisation of failure conditions (Levels A to E, where Level A is the highest and therefore requires the most rigorous processes). Each level of software assurance is associated with a set of objectives, mostly related to the underlying lifecycle process, e.g. planning, development and verification activities (Fig. 1). The objectives of DO178C are sufficiently detailed that it effectively circumscribes much of the development process, even where it does not prescribe specific techniques as in standards such as IEC 61508 (IEC, 2010).

To demonstrate compliance with DO178C, applicants are required to submit the following data to the certification authority:

- Plan for Software Aspects of Certification (PSAC).
- Software Configuration Index.
- Software Accomplishment Summary (SAS).

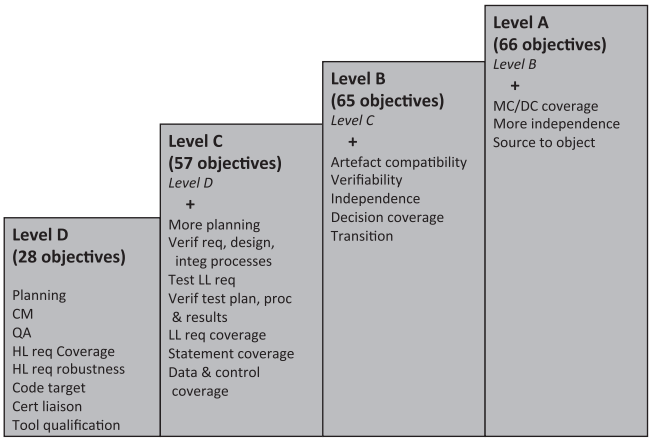

Fig. 1. Overview of DO178C assurance levels.

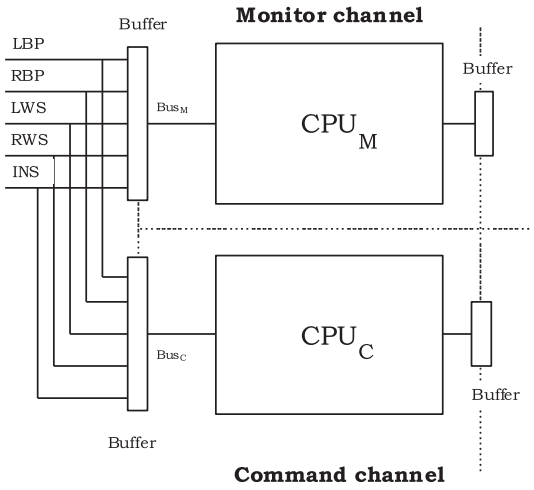

Fig. 2. BSCU architecture.

They should also make all software lifecycle data, e.g. related to development, verification and planning, available for review by the certification authority. In particular, the SAS should provide evidence that shows compliance with the PSAC. The SAS should provide an overview of the system (e.g. architecture and safety features) and software (e.g. functions and partitioning strategies). It should also provide a summary of the potential software contributions to system hazards, based on the system safety assessment, and how this relates to the allocated assurance level. The SAS then references the software lifecycle data produced to satisfy the objectives associated with the allocated assurance level.

4. Wheel braking system case study

In order to make the comparison between assurance cases and prescriptive certification more focused, an example of software for an aircraft system is used. The system under consideration is a Wheel Braking System (WBS) for a new aircraft, called the S18, based on the system specification in the Aviation Recommended Practice (ARP) 4761 Appendix L (SAE, 1996b). The WBS provides the braking of the wheels of the Main Landing Gear during landing and rejected takeoff (RTO).

The system contains two independent Brake System Control Units (BSCUs). Each BSCU contains a command and a monitor channel as discussed in ARP4761 Appendix L and illustrated in the architecture in Fig. 2. Brake pedal inputs (LBP and RBP), wheel speeds (LWS and RWS) and aircraft speed (INS) are provided to the command and monitor channels. The command channel computes the necessary braking commands. The monitor channel checks the plausibility of the sensor data and reports the validity. A failure reported by either channel will cause the BSCU to disable its outputs and set the BSCU Validity to invalid. The Validity of each BSCU is provided to an overall System Validity Monitor. Failure of both BSCUs will cause an independent Alternate Brake System to be selected.

To demonstrate that the software system described above is sufficiently safe to operate as part of the S18 aircraft, it is necessary to provide assurance that the contribution that the software may make to the aircraft hazards are acceptably managed. This involves understanding the role that software could play in bringing about the system level hazards. Functional hazard analysis of the WBS, as documented in SAE (1996b), identifies the following main system hazards: Loss of Deceleration Capability, Inadvertent Deceleration, Partial Loss of Deceleration Capability and Asymmetric Deceleration. Given the direct control exercised by the BSCU, the following hazardous software contributions relating to these hazards can be easily identified:

- **Contribution 1:**: Software fails to command braking when required.
- **Contribution 2**: Software commands braking when not required.
- **Contribution 3**: Software commands incorrect braking force.

From the software contributions to system hazards identified above, it is possible to use knowledge of the BSCU high-level software architecture design shown in Fig. 2 to specify a set of software safety requirements (SSRs) for the Monitor and Command channels of the BSCU. SSRs are required to define what each of the components must do in order to mitigate the hazardous contributions of software. In this example, we only consider the Command component. In particular, we consider the following SSRs:

- **SSR1**: On receipt of brake pedal position, Command shall calculate braking force and output braking command (from contribution 1).
- **SSR2**: Command shall not output a braking command unless brake pedal position is received (from contribution 2).
- **SSR3**: Braking commands generated by the Command component meet defined criteria (from contribution 3). This requirement would need to define safe criteria for the braking commands generated by both the simple braking function, and also the more complex ABS function (see Fig. 3). For reasons of brevity, we do not define those criteria here.

These software safety requirements will be further refined as more design detail becomes available (Lee et al., 2010). For example, with the low-level design for the command channel, as shown in Fig. 3, SSRs will be defined for the individual 'In', 'Braking', 'ABS', 'CMD Modifier' and 'OUT' functions of the Command component.

In the next sections, we firstly describe how an assurance case could be generated for the BSCU software. We then describe how the DO178C guidance could be applied to the same system. In Section 5 we compare the outcome of the two approaches, and also broaden the discussion to consider other standards.

4.1. Developing an assurance case argument for BSCU software

In order to represent the BSCU software safety assurance argument clearly, we represent the argument graphically. We have chosen to use the Goal Structuring Notation (GSN) (Kelly, 1998) to

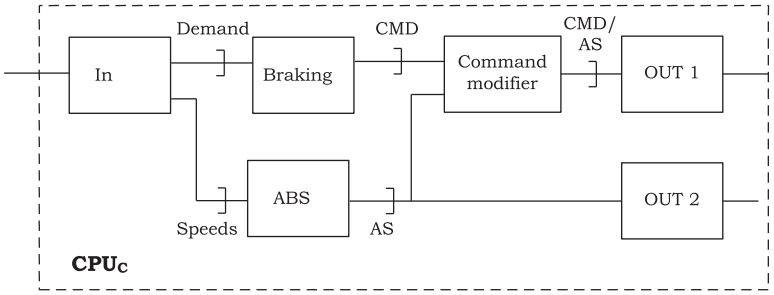

Fig. 3. Design of the BSCU command channel software.

represent the assurance argument as this is a widely used notation and is also the argument notation with which the authors are most familiar. The basic elements of GSN are shown in Fig. 4. These symbols can be used to construct an argument by showing how claims (goals) are broken down into sub-claims, until eventually they can be supported by evidence (solutions). The strategies adopted, and the rationale (assumptions and justifications) can be captured, along with the context in which the goals are stated. GSN is defined in an industry standard (GSN, 2010).

When constructing more complex arguments, such as those often required for software systems, it can sometimes be useful to split the argument into separate, but interrelated modules of arguments (Kelly, 2001). When splitting an argument into modules it becomes necessary to be able to refer to goals that exist within other modules. To refer to goals in other modules, the GSN element "Away Goal" is used. Each away goal contains a module identifier, which is a reference to the module where the goal can be found (see Fig. 5).

Due to large size of the complete BSCU safety assurance case, we only provide and discuss some of the key parts of the argument and evidence. Any goals in the argument structures presented in this section whose support we choose not to show are represented with a diamond symbol beneath them (undeveloped goals). Such goals would be developed further in the complete safety case.

The structure of the assurance argument detailed below is based around demonstrating the following principles:

1. Software safety requirements shall be defined to address the software contribution to system hazards.
2. The intent of the software safety requirements shall be maintained throughout requirements decomposition.
3. Software safety requirements shall be satisfied.
4. Hazardous behaviour of the software shall be identified and mitigated.

The argument structure we use to demonstrate assurance in each of these principles follows that proposed in Hawkins and Kelly (2010) and Hawkins et al. (2011). It should be noted that

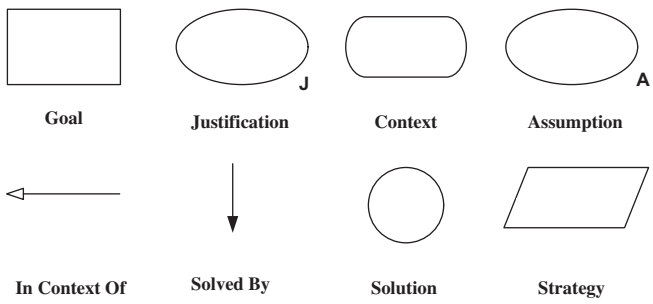

Fig. 4. Main elements of the GSN notation.

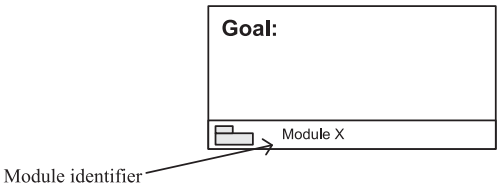

Fig. 5. GSN away goal.

although the assurance approach described is a product-based approach (driven by the specific system hazards and based on consideration of the properties of the specific software system) it is still crucial that the rigor of the processes used is considered. In the following sections we describe in detail the argument structure created for the BSCU software.

4.1.1. High level argument structure

Fig. 6 shows the high-level structure of the software safety assurance argument for the BSCU represented using GSN. The aim of the assurance argument is to demonstrate that the contribution made by the BSCU software to S18 WBS hazards is acceptable. This safety claim is represented using a goal element (Goal: swContributionAcc). This top-level goal is supported by sub-goals relating to each of the identified software contributions. The GSN strategy element is used to make clear what the strategy adopted is (Strat: swContributionAcc). The context in which the top-level argument claim is made is provided using a number of GSN context elements. The context for the top-level claim are system and software descriptions and a list of the identified hazards. 'Goal:contIdent' provides backing for the strategy in a separate argument module. We discuss the provision of backing arguments in more detail in Section 4.1.5.

Fig. 7 shows a simple argument for 'Goal: contIdent'. This shows how the base events from system level fault tree analysis may be used to identify the software contributions to the system level hazards. One of the key features of this argument is the provision of a trustworthiness argument (Goal:WBSFTATrust) relating to the evidence used in the argument (in this case evidence from fault tree analysis).

4.1.2. Trustworthiness arguments

It is desirable to provide a trustworthiness argument for all key evidence items in an argument (such as fault tree evidence in Fig. 7). The concept of trustworthiness relates to freedom from flaw. The flaws with which we are concerned are those that may prevent the item of evidence from fulfilling its role in the argument. In the legal field the notion of integrity of evidence is often used to refer to the soundness or quality of the evidence put forward in a case. Lawyers may look to undermine the integrity of evidence by questioning the way in which the evidence was gathered. Similarly, in considering the trustworthiness of an item of evidence

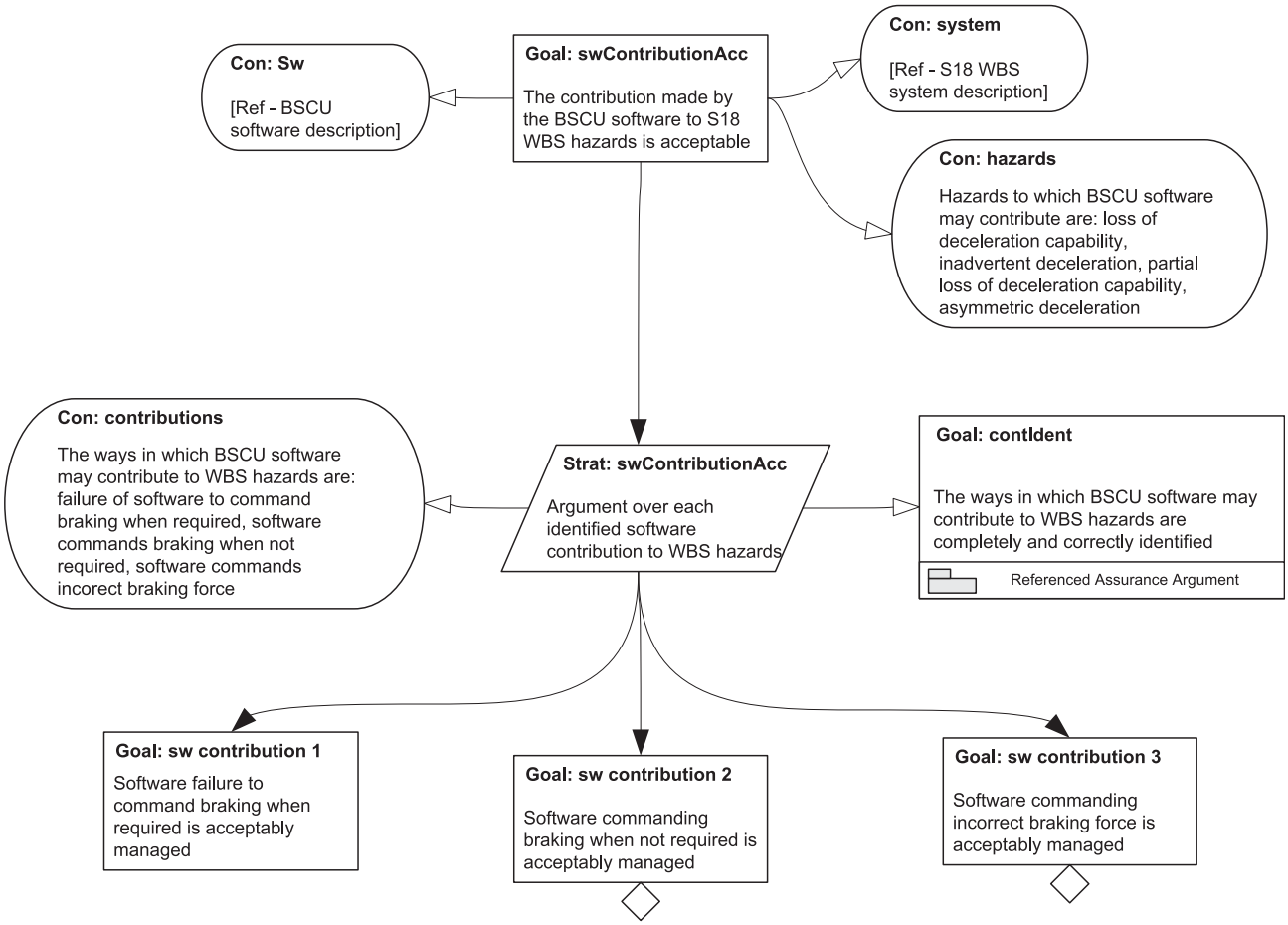

Fig. 6. Top-level software assurance argument.

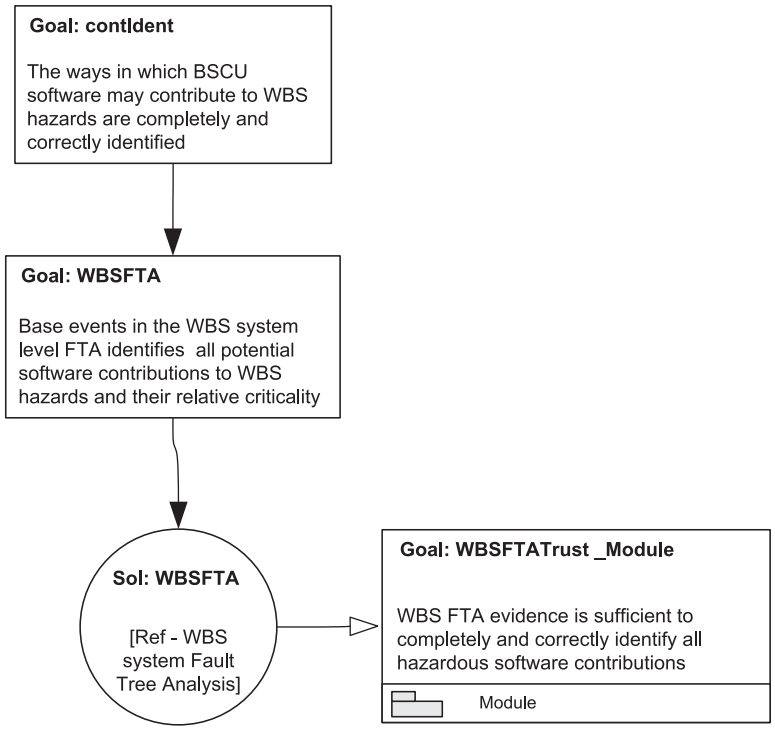

Fig. 7. Software contribution argument.

used to support an assurance argument, the processes used to generate that artefact are often considered. The structure of such an argument is discussed in Habli and Kelly (2007) and Habli and Kelly (2008a).

4.1.3. Arguments over the software design

Claims are made in the argument in Fig. 6 that each of the identified software contributions is acceptably managed. In our example in Fig. 8 we develop the argument relating to just contribution 1 (i.e. software failure to command braking when required). To support this claim in the argument it is necessary to consider the design and development of the BSCU software. In line with the earlier stated principles, the argument must demonstrate assurance in the establishment of valid and complete software safety requirements (SSRs) for each 'tier' of decomposition of the software design for the BSCU software. It must then be demonstrated that those SSRs are satisfied throughout design decomposition, and that there is an absence of hazardous errors at each tier. In the example BSCU system the tiers of design decomposition to be considered are the software architecture design, the channel design, the low-level module design and the source code.

4.1.4. Argument at software architecture level

In Fig. 8 we begin by considering the BSCU architecture design (the highest tier of design decomposition). The software architecture design was shown in Fig. 2. It can be seen in Fig. 8 that claims are made that the SSRs are addressed for each of the channels identified in the architectural design (Command and Monitor channels). The support for each of these claims is provided in separate modules of argument. We consider the arguments in these modules later.

4.1.5. Backing arguments

The use of argument modules helps to simplify the argument structure. In Fig. 8, the modular argument notation has also been used to separate out the backing arguments into separate modules from the main argument. The backing arguments in Fig. 8 are 'Goal: SSRidentify' and 'Goal: hazContArch'. 'Goal: SSRidentify' demonstrates that the SSRs defined for the software architecture are appropriate given the identified software contributions. 'Goal: hazContArch' demonstrates that potentially hazardous failures are identified for the software architecture (for example mechanisms for interference between the Command and Monitor channels), and are acceptably managed. Both of these backing arguments are crucial to the validity of the safety argument.

4.1.6. Argument for the software channels

Fig. 9 shows the argument relating to the Command channel software (Goal: SSRaddComm). This module of argument was referenced by the argument in Fig. 8 (the argument that would be provided for the Monitor channel would also have a similar structure to this). The argument demonstrates that the SSRs specified for the Command channel are addressed (the SSRs may include functional, timing or data related requirements as necessary). A claim is made regarding each of the identified SSRs. In this case, as an example, we provide the argument in Fig. 9 for one of the SSRs (SSR01).

The context to 'Goal SSR1AddArchCommand' defines this SSR as, "*On receipt of brake pedal position, Command channel shall calculate braking force and output braking command*". There can be seen in Fig. 9 to be two legs to the argument for each SSR.

- Firstly, evidence generated at the software architecture level can be provided in the argument to demonstrate that the SSR is satisfied. In this case system integration testing is used to show that the SSR is met.
- Secondly it is argued that the SSR is addressed through the next level of design decomposition of the software (in this case the low-level design for the command channel). This will involve defining SSRs for each of the low-level design modules for the command channel which are sufficient to realise SSR01. The low-level design modules are shown in Fig. 3.

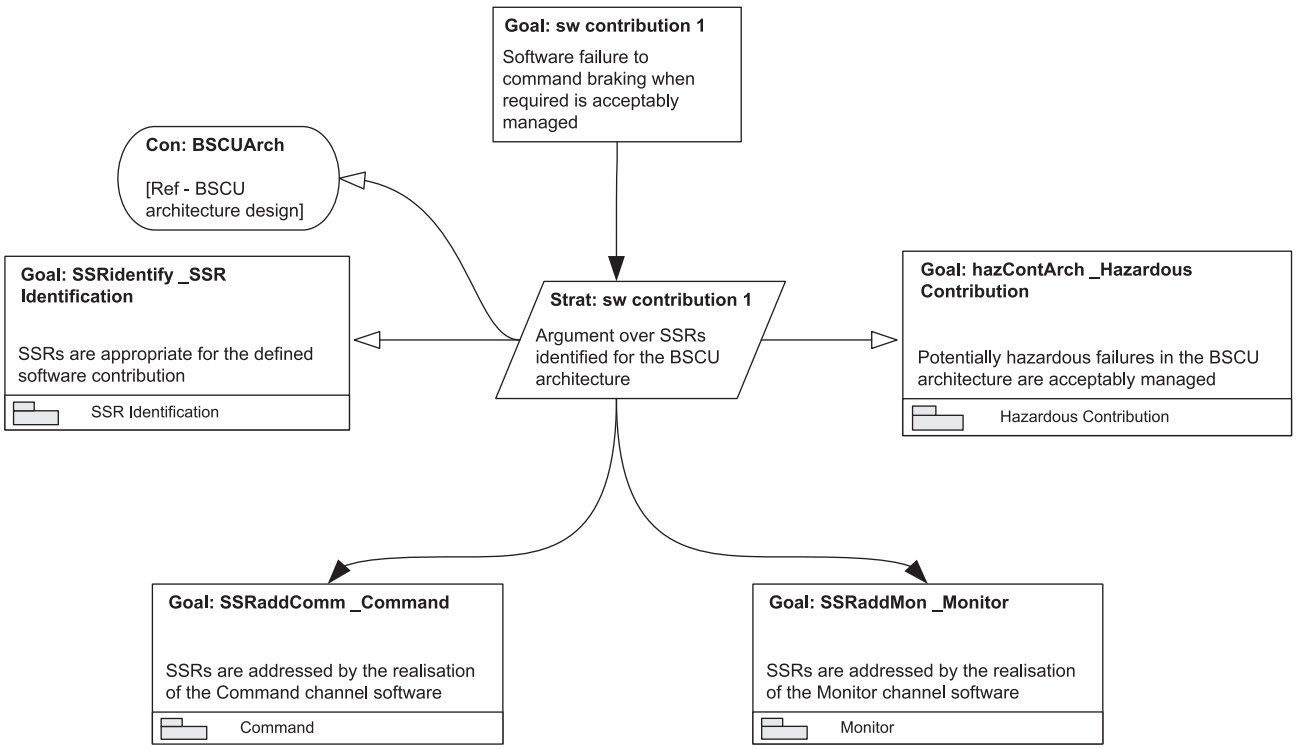

Fig. 8. BSCU architecture assurance argument.

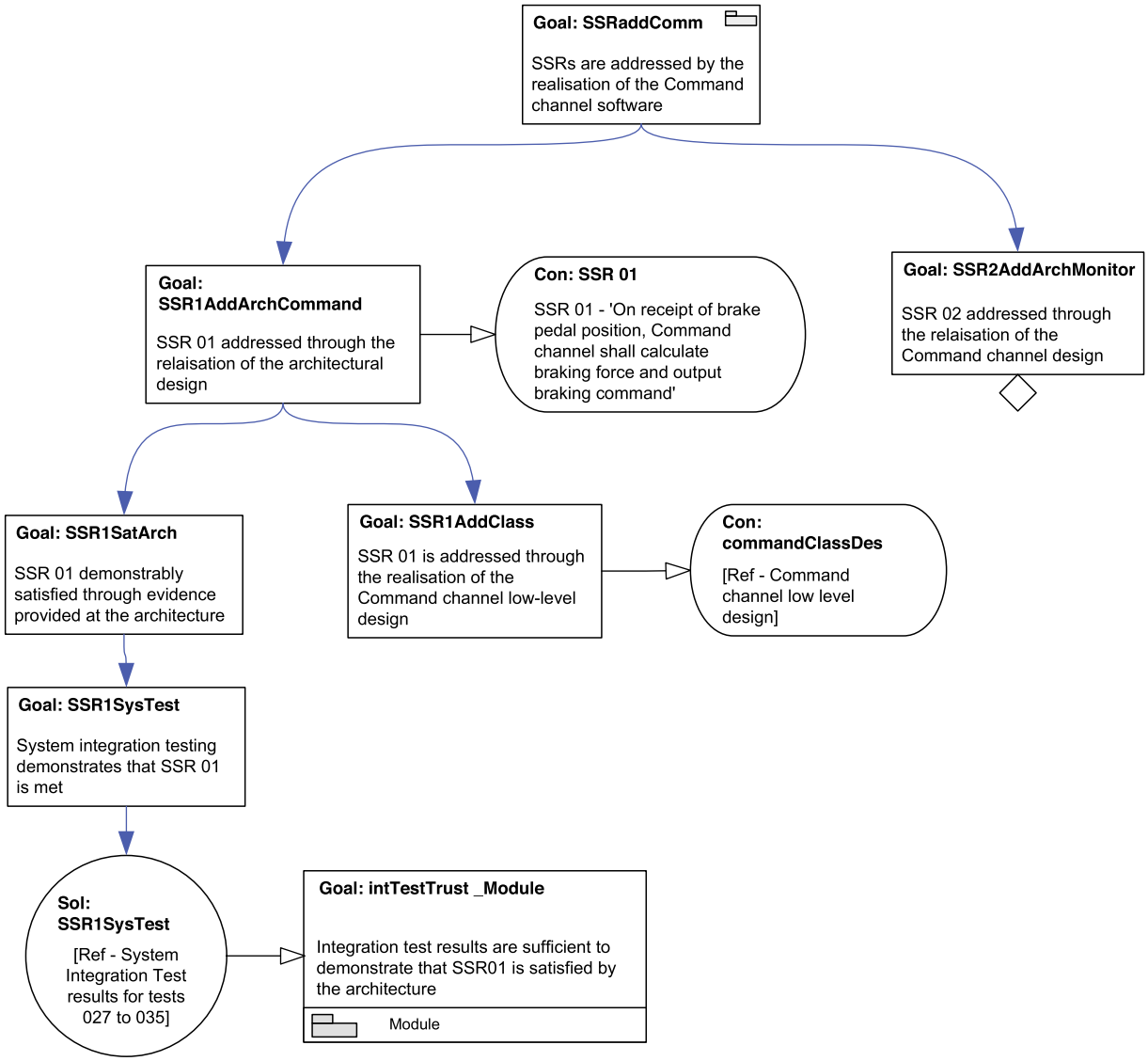

Fig. 9. BSCU command channel assurance argument.

4.1.7. Argument for the channel design

An argument is then presented over each of the Command channel SSRs. In Fig. 10, an argument module is created to reason about the SSRs for each of low-level design modules of the Command channel (Goal: SSRaddBraking to Goal: SSRaddOut). Note that both Output 1 and 2 were considered as part of the same argument module (Goal: SSRaddOut).

As at the architectural design (fig. 8), the argument must demonstrate that the intent of the software safety requirements has been maintained in the design decomposition ('Goal: SSRidentifyDes'). Also that possible hazardous failures that may manifest themselves in the Command channel software design have been identified and mitigated. This is done under 'Goal: hazContDes' as shown in Fig. 11. There are two aspects to the argument in Fig. 11:

- Firstly the argument considers the integrity of the Command channel design itself. In Fig. 11 this is demonstrated simply through evidence provided from a manual review of the Command channel deign that checks the design for errors. Again a trustworthiness argument is provided to demonstrate why that manual review of the design is felt to be sufficient in this case.
- Secondly the argument considers possible unintended (unspecified) behaviour of the software and whether the set of SSRs for the Command channel are sufficient to address these. There are various techniques available for identifying deviations from intended behaviour in software designs that could provide assurance here. In the example in Fig. 11, a Software HAZOP (Redmill et al., 1999) has been applied to the low-level design (details of this are omitted for brevity).

4.1.8. Argument for the module design

Fig. 12 continues the argument from Fig. 10 by showing the argument for the Braking module. It can be seen that the argument relating to this low-level design module has the same basic structure as that provided for the architectural design level in Fig. 9. In this case the SSRs defined for the low-level design are considered (SSRs 1.1, 1.2, and 1.3). The example we have chosen is SSR 1.1 – "*If Demand input is received, then CMD output shall be provided*". A similar argument could be presented for SSR 1.2 and 1.3 also. The evidence that was chosen to demonstrate that the SSRs are satisfied at this level of design can be seen to be unit test results. It should be noted that the results of specific unit tests relating to the particular SSR (SSR 1.1) are provided. Again the safety

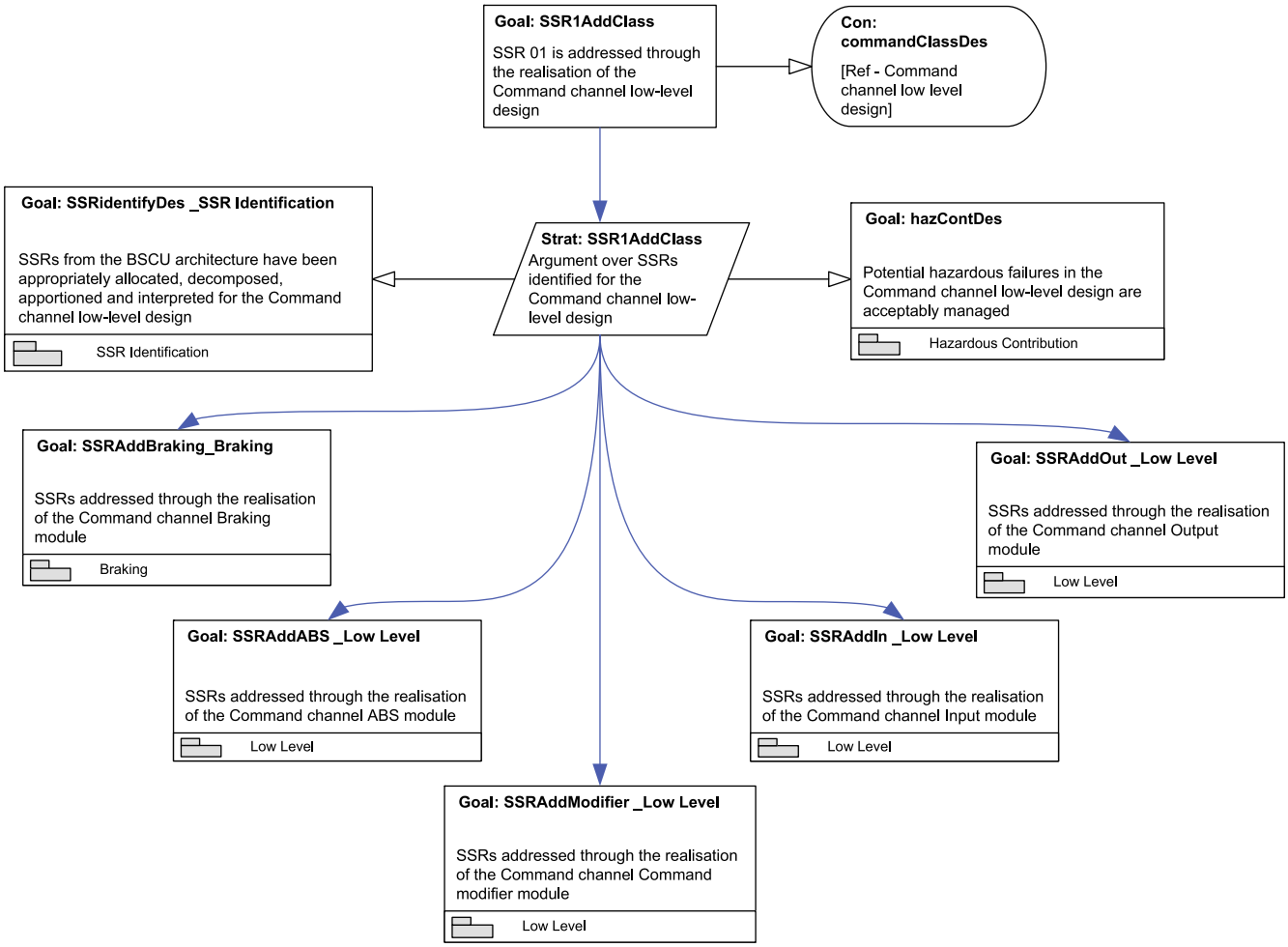

Fig. 10. Command channel design modules argument.

requirement, SSR 1.1, is traced through to the next tier of decomposition in the design, which in this case is the source code itself.

A trustworthiness argument is provided for the unit testing ('Goal: SSR1.1UnitTestTrust'). This argument considers the adequacy of the testing in terms of:

- Its coverage of the required software function
- The integrity of the evidence – considering the process used to produce the evidence as well as the integrity of the tools and people used in its generation.

4.1.9. Argument for the source code

Fig. 13 shows the argument that SSR1.1 is addressed by the source code for the Braking module. At the level of the source code it is not necessary to argue that the SSRs are correctly defined since no new SSRs are specified at the code level (this is captured in Fig. 13 as assumption 'Ass: CodeLevelSSRs'). It is necessary however to demonstrate that the source code does not contain potentially hazardous errors. This is argued under backing argument Goal: desErrorCode shown in Fig. 14. To show that SSR1.1 is satisfied at the source code level, semantic analysis of the code is performed, the report on which is provided in the evidence. To complete the argument it is necessary to show that the SSR is also met by the compiled object code. As indicated in Fig. 13, this could be an argument over the integrity of the compiler used.

Fig. 14 demonstrates assurance that the source code produced for the Braking module does not contain potentially hazardous errors. This is demonstrated through the use of both static information flow analysis to show the absence of run-time errors, and also through a code review of the Braking module source code.

The example discussed above illustrates how, by demonstrating assurance throughout all the development tiers of a software system it is possible for a safety assurance argument to be built up. Guidance on how to produce software safety arguments in the form of a catalogue of software safety argument patterns can be found in Hawkins and Kelly (2010). This builds upon existing work such as Kelly (2001), Weaver (2003), Ye (2005) and Menon et al. (2009) and also takes account of current good practice for software safety, including from existing standards.

4.2. Assurance of BSCU software using DO178C

In the previous section, we illustrated how an assurance argument can be created to justify the safety of the BSCU software. In this section, we present how assurance for the same software can be provided by means of compliance with a prescriptive certification approach based on the DO178C guidance (RTCA, 2012).

Based on the system safety assessment, which is carried out against the Aviation Recommended Practice (ARP) 4754 and 4761 (SAE, 1996a, 1996b), the command channel is allocated assurance level 'A' while the monitor channel is allocated assurance level 'B'. The rationale is that producing and verifying the software against the processes of level 'A' and level 'B' objectives and techniques should preclude software "*design flaws of concern*"

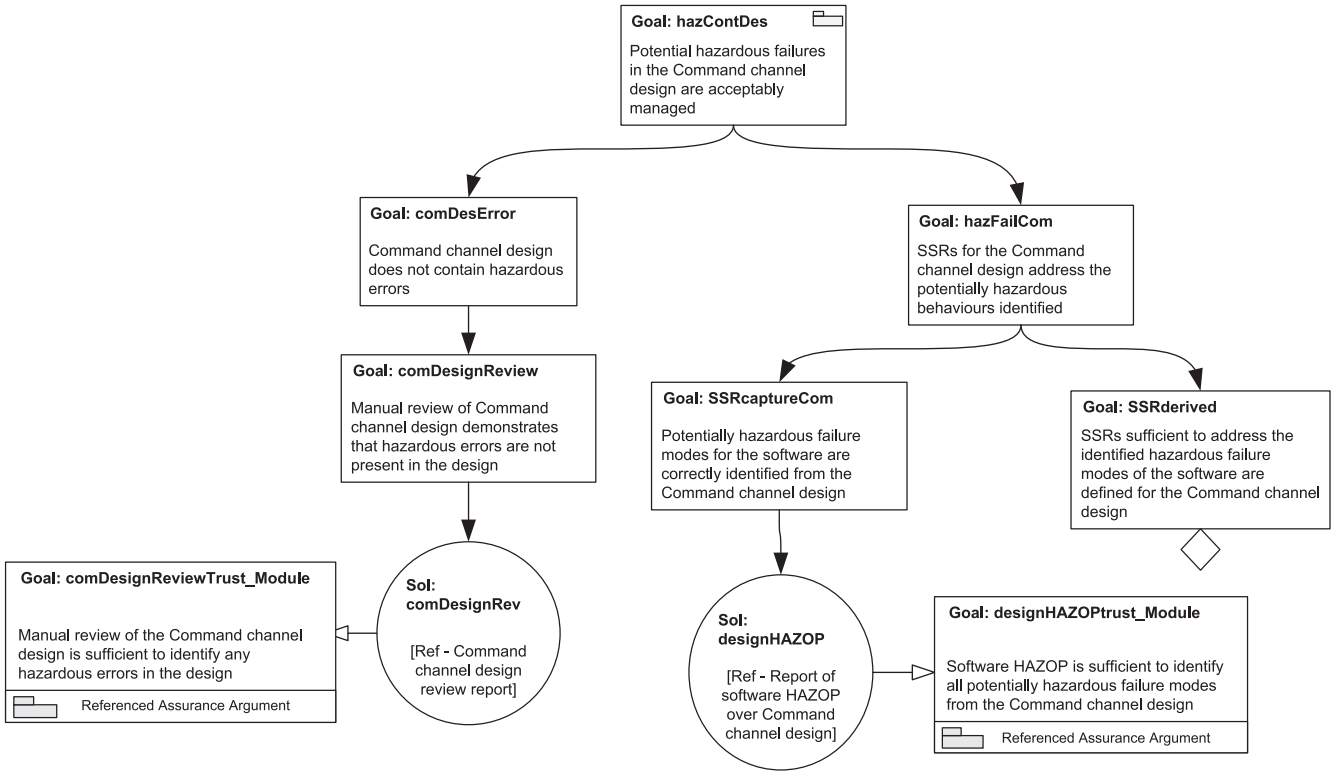

Fig. 11. Command channel hazard analysis software argument.

(SAE, 1996a). This is based on the results of the system fault tree analysis which indicate that there shall be no "*loss of BSCU ability to command braking*" (SAE, 1996a). The results also indicate that "*there must be no common mode failures of the command and monitor channels of a BSCU system that could cause them to provide the same incorrect braking command simultaneously*" (SAE, 1996a). These system safety aspects are flown down from the system processes in the form of safety requirements, as part of the systems requirements allocated to software, based on the Preliminary System Safety Assessment (PSSA).

At this stage, unlike safety assessment at the system level, no explicit hazard and failure analysis is required at the BSCU software level. All of what is needed at this stage is for the command and monitor channels to be produced to assurance levels 'A' and 'B' respectively. For the command channel, which is Level 'A', an additional objective related to the accomplishment of the Modified Condition/Decision Coverage (MC/DC) of software structure is required (Hayhurst et al., 2001). Further, for level 'A' software, more objectives, such as the review of the software architecture and source code, have to be achieved with independence, i.e. verification is carried out by individuals other than the developers of the artefact under verification (CAST, 2006). Finally, traceability between source and object code should be justified if structural coverage analysis is performed on the source code rather than on the executable object code. Evidence generated by compliance with Levels 'A' and 'B' guidance is based on data produced by the planning, development, verification, configuration management and quality assurance activities. These activities are specified in the DO178C guidance. The data generated by these activities is summarised in Table 1.

Although most of the data described in Table 1 is necessary to provide assurance for the BSCU software, the software verification results provide the most targeted evidence concerning the satisfaction of the system requirements, particularly those related to safety. The certification objectives satisfied by the software verification results for software level 'A' are listed in Table 2. For example, for the system safety requirement that there shall be no "*loss of BSCU ability to command braking*" (SAE, 1996a), verification results, generated from review, analysis and testing, should provide evidence to substantiate the following:

- One or more high-level software requirements have been developed that satisfy the system safety requirement that there shall be no "*loss of BSCU ability to command braking*". These requirements are allocated onto the Monitor and Command channels of the BSCU and specify the required behaviour of these channels in order to mitigate the hazardous contributions of the "*loss of BSCU ability to command braking*".
- These high-level software requirements have been refined into software architecture and low-level software requirements which are in turn developed into, and satisfied by, source code. For example, low-level software requirements specify how the software components within the Command channel should respond to external inputs, e.g. brake pedal position.
- Executable object code satisfies all the software requirements (including derived software requirements), and is robust against predictable unintended inputs.
- Executable object code does not perform any unintended function.

Given that exhaustive testing is infeasible for a complex software system such as the BSCU (Butler and Finelli, 1991), DO178C defines various verification criteria for judging the adequacy of the software testing (i.e. *has the software been tested enough?*). Two of these criteria relate to the coverage of requirements and code structure (see '*Verification of Outputs of Software Testing*' in Table 2). The rigour of these criteria is associated with the safety criticality of the software. The analysis of the achievement of the

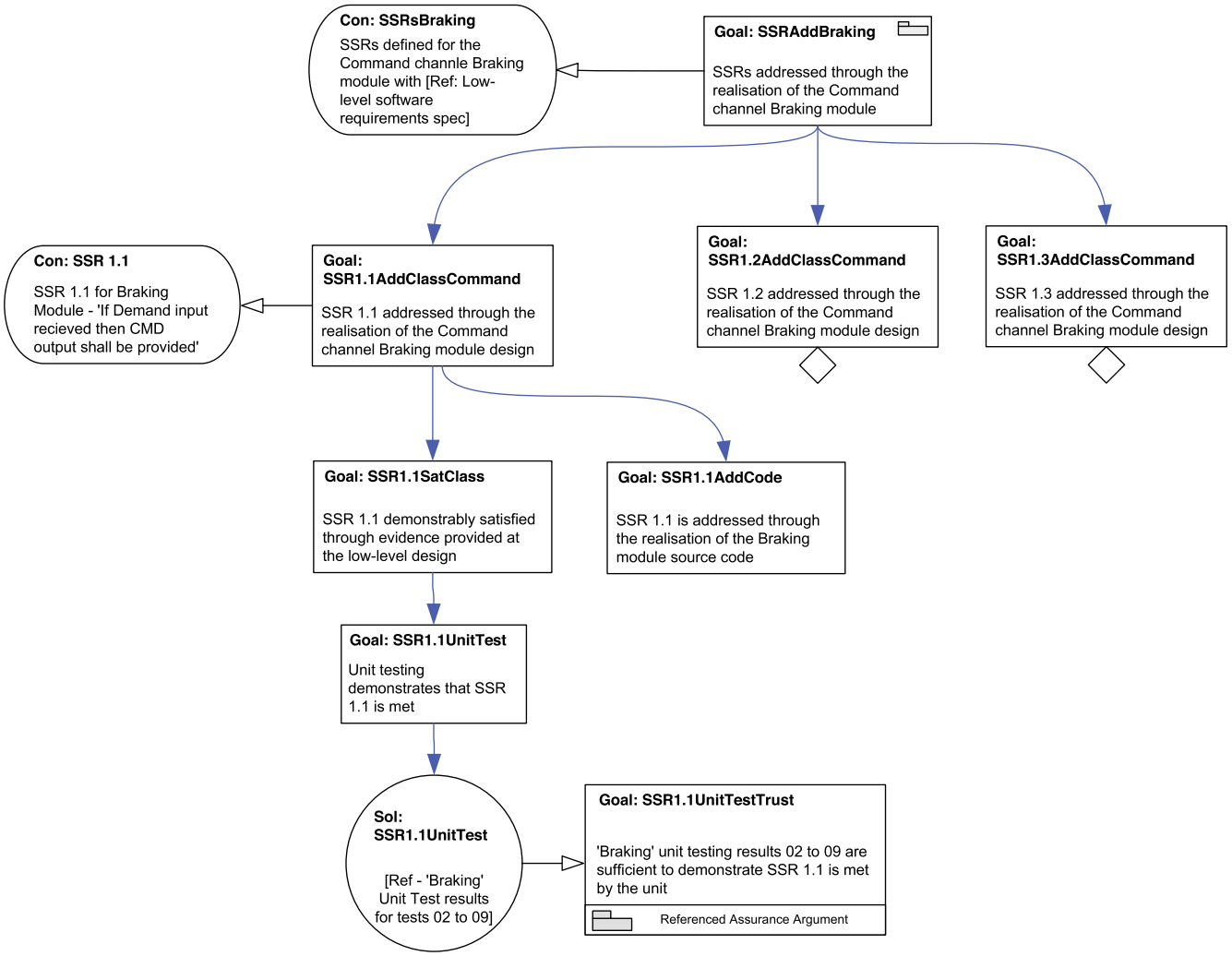

Fig. 12. Braking module assurance argument.

requirements coverage criterion should show that normal and robustness test cases exist for every software requirement. However, this is often not enough, as it does not demonstrate absence of unintended functionality. The absence of unintended functionality is addressed by structural coverage analysis which measures how well the software structure is exercised by the test cases (i.e. have we exercised the software structure enough to get confidence concerning the absence of unintended functionality?). For example, for the monitor channel, which is level 'B', structural coverage analysis should demonstrate decision coverage of the code structure. On the other hand, for the command channel, which is level 'A', structural coverage analysis should demonstrate Modified Condition/Decision Coverage (MC/DC) of the code structure, a more rigorous coverage criterion requiring that "*each condition is shown to independently affect the outcome of the decision*" (Hayhurst et al., 2001).

The way in which verification evidence is specified in DO178C is a pragmatic approach, based on consensus within the aerospace community, concerning the required rigour for evidence against the different levels of safety assurance (e.g. MC/DC for level 'A' and decision coverage of level 'B'). These objectives reflect a compromise between the "ideal" of exhaustive coverage and realistic testing strategies. Whilst the choice of MC/DC might now be questioned as formal techniques make more exhaustive analysis practicable, the consensus of the community has been to preserve this objective in DO178C.

In short, DO178C promotes a clear process of requirements refinement. This process starts at the point at which systems requirements are allocated to software. The refinement process concludes when the executable object code is generated and tested against the high-level and low-level requirements. The rigour with which the software is tested is proportionate to the safety criticality of the software. Further, the level of trustworthiness in the items of evidence varies between different assurance levels. For example, many items of evidence for satisfying the level 'A' criteria are required to be generated with independence. In contrast to trustworthiness arguments discussed in Section 4.1.2 however, the trustworthiness of an individual item of evidence with respect to its role in the safety argument is not explicitly considered in DO178C. This process also demands that any new functions generated at any refinement stage are addressed by derived software requirements and fed to the system safety assessment process for further analysis.

5. Evaluation of case study results

In this section we evaluate the results, as reported above, of applying the two different software assurance approaches to the case study. Firstly we provide discussion of the two approaches before providing a direct comparison, using the comparison criteria

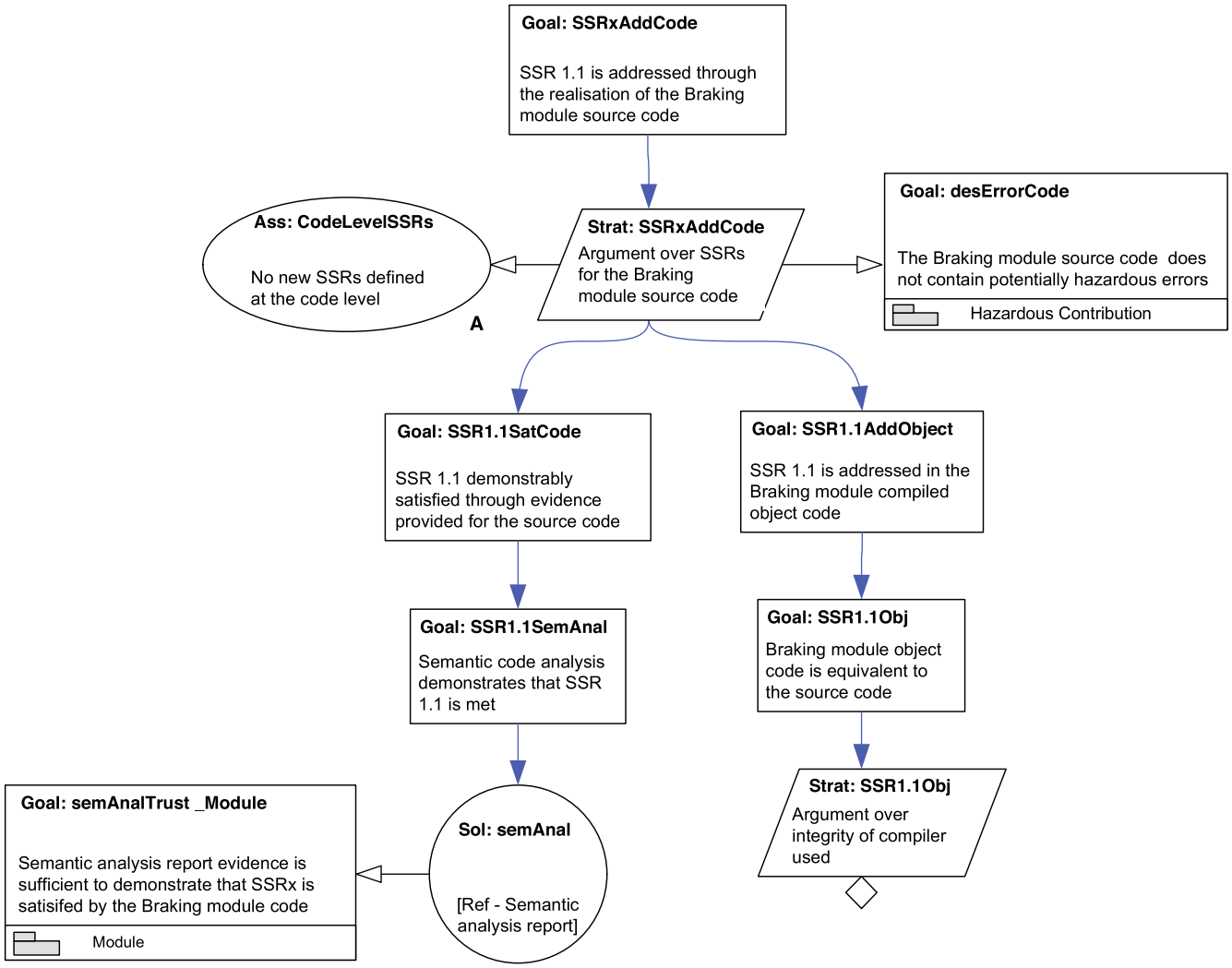

Fig. 13. Braking module source code assurance argument.

described in Section 1.2. Finally we discuss the validity and limitations of the case study results.

5.1. Assurance case argument discussion

The true strength of an assurance argument stems from the fact that it allows the developer to clearly and systematically communicate why it is believed that there is sufficient assurance in the software. A hazard-focused, structured argument such as that described in the example in the previous section can make it much easier to identify areas where the assurance may be insufficient. In particular, rather than providing general claims about requirements traceability and verification, the software assurance argument provides specific claims relating to the safety requirements identified for the software. Crucially, the safety argument also considers all aspects of the safety of the software which may undermine assurance in the specific safety claims made about the product. This may involve analysing the potential uncertainties or assurance deficits in the software design and development. Nevertheless, this level of flexibility in the construction of a software assurance case increases the burden on the certification auditors since the assurance case argument and evidence for different software systems will be different and therefore very difficult to assess in a systematic and repeatable way (Wassyng et al., 2010).

To be effective, the assurance case should not be constructed at the end of a software development project, but developed in parallel with the system itself. In that way the assurance argument can inform design decisions and verification activities. The sufficiency of these activities can be determined and assessed through constructing an explicit software assurance argument. In addition, the role of each item of evidence in the assurance argument can be clearly communicated, thus making it easier to justify its sufficiency for its role in the overall argument. By developing the assurance argument at the same time as the software system, assurance deficits identified through consideration of the assurance argument can be addressed in a timely (and hopefully in a more cost-effective) manner.

Finally, the argument presented in Section 4.1 is based on informal and inductive logic. For example, the argument is not mathematically-based and does not follow a deductive reasoning process. To this end, unlike formal methods, it is not possible to deduce, or formally demonstrate, the substantiation of the safety claims using the available evidence (Rushby, 2010; Wassyng et al., 2010). The use of informal logic in the representation of assurance cases is a common practice. However, as discussed in Section 2, there has been an increased interest in improving the integration between formal and informal reasoning in assurance cases (Basir et al., 2010).

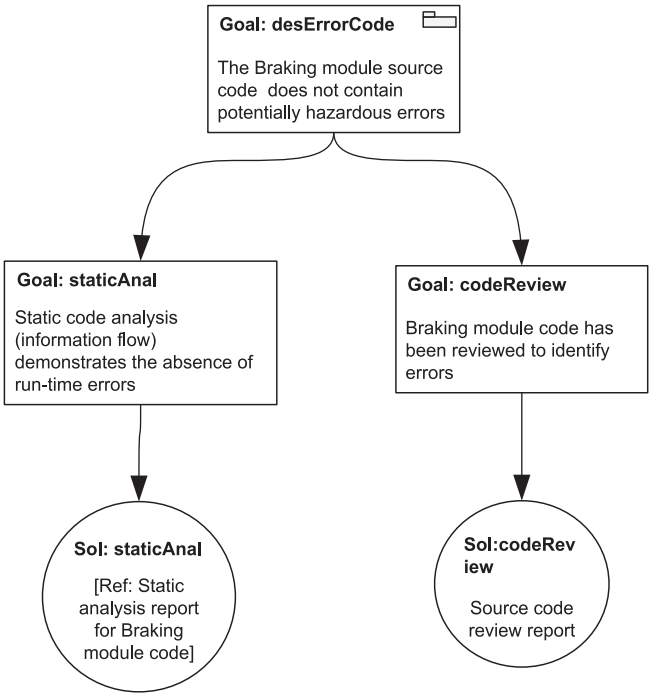

Fig. 14. Code-level hazard analysis argument.

5.2. Discussion of the BSCU software compliance against DO178C

Compliance with DO178C generates different items of evidence in order to support various certification objectives such as those described in Table 2. When these items of evidence are compared with those supporting the assurance argument defined in Section 4.1, it can be noticed that the outcome of compliance with DO178C can provide full coverage of the evidence within the assurance case. Nevertheless, it is important to note that this mapping is not one-to-one. For example, Table 3 shows a mapping between the items of the evidence included in the software assurance case in Section 4.1 and how they correspond to the objectives satisfied by DO178C verification results. In particular, consider the two items of evidence in the assurance case (classDesignReview and designHAZOP) which substantiate the claim that "*potential hazardous failures in the Command channel low-level design are acceptably managed*". These items of evidence correspond to the A.4.1 objective in DO178C which states that "*Low-level requirements comply with high-level requirements*". The text in DO178C explains that this objective aims "*to ensure that the low-level requirements satisfy the high-level requirements and that derived requirements and the design basis for their existence are correctly defined*". Many of these derived requirements are defined in order to address situations in which the "*software design processes activities could introduce possible modes of failure into the software or, conversely, preclude others*". Here, there is a difference between DO178C and an assurance case approach in how hazardous software failure modes are considered. In the example assurance case, the sufficiency of the identification and mitigation of hazardous software failure modes is explicitly justified within the software argument. In DO178C, software failure modes – where identified and where considered of possible significance with respect to system level hazards – are fed back to the system safety assessment process for further analysis. However, there is no explicit requirement, or suggested process, for the identification of software failures modes as part of the software development lifecycle. As a result, our experience is that this is not done in a systematic or consistent manner, if at all.

Another important aspect is that not all items of evidence in the software assurance case are within the scope of DO178C. For example, as shown in Fig. 7, the item of evidence 'WBSFTA' is generated by the system fault tree analysis at the system level, against another aerospace standard 4754 and 4761 (SAE, 1996a, 1996b). This is clearly an input to the DO178C process rather than an output of it. Therefore, it is important to consider the DO178C guidance in the context of the wider system certification process.

In short, various types of evidence have to be generated for compliance with DO178C. The rigour of these types of evidence is predetermined by the DO178C guidance and varies according to the allocated assurance level. In particular, verification evidence, generated from review, analysis and testing, should show the achievement of objectives relating to the development, satisfaction and traceability of systems requirements allocated to software. The achievement of these objectives reflects a core argument within DO178C, which may be implicit, unless it is summarised within the PSAC and SAS.

Whilst this paper has focused upon one particular software assurance standard – DO178C – as an exemplar of a prescriptive standard, there are many other standards that share similar characteristics. For example, IEC61508 similarly assumes and organizes its requirements according to lifecycle phase, defines levels of rigour in the process according to Safety Integrity Levels, and provides recommendations as to the required forms of evidence.

Table 1
DO178C lifecycle data.

Plan for Software Aspects of Certification
Software Development Plan
Software Verification Plan
Software Configuration Management Plan
Software Quality Assurance Plan
Software Requirements Standards
Software Design Standards
Software Code Standards
Software Accomplishment Summary
Trace Data
Software Requirements Data
Design Description
Source Code
Executable Object Code
Software Verification Cases and Procedures
Software Verification Results
Software Life Cycle Environment Configuration Index
Software Configuration Index
Problem Reports
Software Configuration Management Records
Software Quality Assurance Records

5.3. Comparing an assurance case and a DO178C approach

The discussion in Section 4.2 highlights the range of evidence that is generated for the WBS software through following a DO178C process. It can be seen that all of such evidence could be used to support an assurance argument as described in Section 4.1. The assurance argument can in such a way be used to explain how the evidence generated from the DO178C process provides sufficient assurance.

In this section, we revisit the comparison criteria introduced in Section 1.2 and explain the advantages and limitations of assurance cases and prescriptive certification based on the outcome of Sections 4.1 and 4.2. We then use this to identify two ways in which the approaches might be reconciled.

1. *Clarity*: Compliance with the DO178C guidance produces a set of lifecycle data covering the whole of the software engineering process (as shown in Table 1). The aims for the production of this data are clearly stated in the guidance.

Further, templates are already provided for the documentation of certain lifecycle data. For example, the organisation of the Plan for Software Aspects of Certification and Software Configuration Management Plan is explicitly defined by the guidance. The DO178C guidance insists that all lifecycle data "*is written in terms which only allow a single interpretation, aided if necessary by a definition*". Similarly, the structural elements of an assurance case are clearly defined in most standards that require them, i.e. claims, arguments and evidence. In addition, many standards require that the arguments being presented be comprehensible and clear. However, making an assurance argument *clear* requires more than simply using the syntax of structured argumentation. Clarity of the data generated from prescriptive standards and assurance cases can therefore be seen to serve two different purposes. Clarity in prescriptive standards such as DO178C targets compliance (Graydon et al., 2012), e.g. is it clear how the generated lifecycle data achieve the certification objectives? Clarity in assurance arguments targets safety assurance, i.e. is it clear how strong the argument and evidence are in supporting the claims about software safety?.

2. *Rationale*: DO178C provides valuable guidance on how to implement high-quality and repeatable software engineering processes. Whilst there can be said to be an implicit rationale (as established by the standards writers) for the specific processes and forms of evidence required by standards such as DO178C, there is a lack of *explicit* rationale. Being implicit, it can be harder for developers to appreciate the significance of the standard's requirements, and why they vary according to level of risk. In addition, it can be harder to judge the applicability of the requirements to the specific application under development.

   Assurance cases demand rationale (for the development decisions made and assurance evidence generated) to be presented (by the developer) on a case by case basis. This enables the rationale to be closely and explicitly tied to the specific features of the application and criticality of the required behaviour. However, without the practice of developing and sharing assurance case patterns, this could be seen as detrimental to the propagation of good practice.

   It is worth noting that the evidence generated by following a DO178C approach may be sufficient to support a compelling safety assurance argument *if* the rationale accompanying the use and interpretation of DO178C is explicitly communicated. The experience of the authors is that normally the only rationale for producing the evidence is that DO178C requires it. The link to the required safety behaviour of the system is therefore implicit and potentially tenuous. The areas of the argument *most implicitly* supported by DO178C evidence are those related to the analysis of hazardous failure behaviour. A DO178C process does not require any explicit software hazard analysis to be performed at the software level. Whilst this can also be overlooked in software safety case development (e.g. where the case relies simply on appeal to development and assurance processes) the requirements for explicit arguments and evidence should make this visible, and identifiable in review.

3. *Amenability to review*: From a mere compliance point of view, reviewing compliance with DO178C is easier than reviewing the acceptability of an assurance case since most of the required items of evidence needed for satisfying the DO178C objectives are prescribed. In effect the standard gives checklists through the definition of the scope of a SAS and means for compliance. The absence of information, or the inadequacy of information, e.g. on the handling of change, is often immediately apparent on reading a SAS. However, from a safety assurance point of view, the level of confidence that can be placed in the evidence is more difficult to review in the absence of an explicit assurance argument that justifies why the evidence supports claims about software safety. DO178C focuses, almost exclusively, on the generation of evidence. The assurance argument is implicit within the approach. By definition it is very difficult to understand an implicit argument, and therefore it is also difficult to review. By producing an explicit assurance argument, the suitability of the DO178C evidence for demonstrating that the software is acceptably safe to operate in a particular context could be more clearly reviewed and judged.

   The explicit representation of an assurance argument is particularly important for developers wishing to provide alternative items of evidence, e.g. evidence generated from formal mathematical analysis rather than testing (Hayhurst et al., 1998). Where alternative approaches are used, a reviewer must be convinced of the relevance and suitability of the alternative evidence. This is much easier to achieve using an explicit assurance argument.

   In summary, it is often easier to review a SAS than an assurance case with respect to *compliance*. However, it is more effective for a reviewer to judge the adequacy of the evidence with respect to the *assurance of risk mitigation* when provided with a well-structured safety argument.

4. *Predictability*: Our consideration of using DO178C for certification of the WBS case study has indicated that the achievement of compliance with DO178C is far more predictable than establishing and demonstrating assurance by means of assurance cases. The certification objectives, means for compliance and types of evidence are already clearly defined in the DO178C guidance.

   To achieve a similar level of predictability for an assurance case approach would require that objective criteria for judging a successful assurance case are clearly articulated from the outset. Currently standards requiring an assurance case provide little guidance on what these criteria are, or how they may be judged. Indeed, this has been one of the criticisms leveled at standards such as UK Defence Standard 00-56 (MoD, 2007). One of the motivations for extracting and documenting patterns of assurance case arguments is to help promulgate best practice, and provide extra guidance in the development and review of assurance cases. Using existing patterns as part of planning the assurance approach to be adopted on a project can improve predictability and consistency of the process. The example in Section 4.1 illustrates how the use of patterns to structure an argument can be more predictable than an entirely open-ended assurance case approach. Indeed, recent experience supports this finding (Hawkins et al., 2011).

5. *Effort*: The assurance argument can help to target the effort where it adds most value from a safety assurance perspective rather than from a correctness perspective. It is unclear however how much effort the assurance case approach adds to a software development project – or, conversely, if it can save effort by avoiding the generation of evidence which is not of significance for a particular system. Cost-benefit studies are essential in order to begin to answer these questions. This is an important area for further work. An area where cost may be reduced is through separating safety–critical and non-safety–critical behaviours within the same software application, thus developing some of the application to a lower assurance. This could be achieved through

Table 2
DO178C verification objectives.

Verification of outputs of software requirements process	Verification of outputs of software coding and integration processes
A7.3.1: Software high-level requirements comply with system requirements	A7.5.1: Source Code complies with low-level requirements
A7.3.2: High-level requirements are accurate and consistent	A7.5.2: Source Code complies with software architecture
A7.3.2: High-level requirements are compatible with target computer	A7.5.3: Source Code is verifiable
A7.3.2: High-level requirements are verifiable	A7.5.4: Source Code conforms to standards
A7.3.2: High-level requirements conform to standards	A7.5.5: Source Code is traceable to low-level requirements
A7.3.2: High-level requirements are traceable to system requirements	A7.5.6: Source Code is accurate and consistent
A7.3.2: Algorithms are accurate	A7.5.7: Output of software integration process is complete and correct
Verification of outputs of software design process (LL requirements)	**Testing of outputs of integration process**
A7.4.1: Low-level requirements comply with high-level requirements	A7.6.1: Executable Object Code complies with high-level requirements
A7.4.2: Low-level requirements are accurate and consistent	A7.6.2: Executable Object Code is robust with high-level requirements
A7.4.3: Low-level requirements are compatible with target computer	A7.6.3: Executable Object Code complies with low-level requirements
A7.4.4: Low-level requirements are verifiable	A7.6.4: Executable Object Code is robust with low-level requirements
A7.4.5: Low-level requirements conform to standards	A7.6.5: Executable Object Code is compatible with target computer
A7.4.6: Low-level requirements are traceable to high-level requirements	
A7.4.7: Algorithms are accurate	
Verification of outputs of software design process (architectures)	**Verification of outputs of software testing**
A7.4.8: Software architecture is compatible with high-level requirements	A7.7.1: Test procedures and expected results are correct
A7.4.9: Software architecture is consistent	A7.7.2: Test results are correct and discrepancies explained
A7.4.10: Software architecture is compatible with target computer	A7.7.3: Test coverage of high-level requirements is achieved
A7.4.11: Software architecture is verifiable	A7.7.4: Test coverage of low-level requirements is achieved
A7.4.12: Software architecture conforms to standards	A7.7.5: Test coverage of software structure (MC/DC) is achieved
A7.4.13: Software partitioning integrity is confirmed	A7.7.6: Test coverage of software structure (decision coverage) is achieved
	A7.7.7: Test coverage of software structure (statement coverage) is achieved
	A7.7.8: Test coverage of software structure (data coupling and control coupling) is achieved

Table 3
Assurance case evidence vs. DO178C verification results.

Assurance case evidence	DO178C verification results
Sol: WBSFTA [Ref - WBS system Fault Tree Analysis]	Outside the scope of DO178C (addressed at the systems level by ARP4761)
Sol: SSR1SysTest [Ref – System Integration Test results for tests 027 to 035]	A7.4.8, A7.6.3, A7.6.4, A7.7.5, A7.7.8 and A7.7.4
Sol: classDesignReview: [Ref – low-level design review report]	A7.4.1
Sol: designHAZOP: [Ref – report of software HAZOP over low level design]	
Sol: SSRxUnitTest:[Ref – 'Braking' Unit Test results for tests 02 to 09]	A7.6.3 and A7.6.4, A7.7.5, A7.7.8 and A7.7.4
Sol: semAnal [Ref – semantic analysis report]	A7.5.1
Sol: staticAnal [Ref: static analysis report for Braking module code]	A7.5.6
Sol:codeReview Source code review report	A7.5.4 and A7.5.6

including an argument of non-interference between software of differing levels of criticality as part of an assurance case. There are questions however as to whether, in the light of the increasing complexity of modern software-intensive systems, it is feasible to make such an argument with any confidence. It may in fact be cheaper to develop all the software using a common assurance approach, especially as verification technology improves, rather than perform the required detailed analysis of non-interference.

What the comparison above has indicated is that neither the prescriptive approach of DO178C, nor an assurance case approach provides a perfect solution for software safety assurance. Both approaches have advantages and limitations. What this paper seeks to demonstrate however is that if used correctly, the two approaches can be complementary, and could lead to a better solution than either approach on its own. For example, a prescriptive standard could be used as guidance as to best practice on evidence selection (such as indicating the level of test coverage required, or defining the necessary level of independent review); the assurance case can be used to explain the role of that evidence as part of the demonstration of the safety of the particular system under consideration. The assurance case can also be used to provide justification for system-specific deviations from the obligations of a prescriptive standard, either where an alternative approach is more appropriate for the particular system design solution, or where the obligations of the standard, on their own, would be insufficient due to the nature of the system hazards.

The above observations are supported by other published research. Knight (2008) describes the need for the flexibility that an assurance case brings for software, particularly for the selection of verification evidence, even if working in a DO178C regime. Further, an incident is described by Johnson and Holloway (2007) where an aviation software system had been tested and developed to DO178C. The software testing was limited to the original specification and requirements of the component. These did not consider the particular scenario that arose during the incident. One of the motivations for proposing a hazard-directed assurance case approach is the belief that this will reduce the likelihood of such omissions.

The explicit consideration of software safety claims, arguments and evidence is also emphasised in a study sponsored by the National Research Council (Jackson et al., 2007). In this study, the authors argue that claims of achievement of dependability by merely complying with certain processes are "*unsubstantiated, and perhaps irresponsible*". The safety assurance of software should be only trusted if a credible case is produced in support of the software safety claims.

> Even in goal-based standards, compliance with a prescriptive certification approach, e.g. based on DO178C, is still considered good practice. For example, the Defence Standard 00-56 (MoD, 2007), which requires an assurance case approach, states that, in some domains, a certification approach such as DO178C "can work extremely well and it has the advantage of providing an authoritative definition of good practice".

5.4. Threats to validity

We conclude this section by considering internal and external threats to the validity of the results of this study.

Internal validity: The system considered in the study, i.e. the aircraft WBS, was based on the specification provided in the Aviation Recommended Practice (ARP) 4761 Appendix L (SAE, 1996b). Ideally, the study should be based on an operational system. However, publically releasing safety-related information that describes system-specific design features is often infeasible. As such, the use of the aircraft WBS specification provides a credible example since it was developed by a committee of practicing aerospace engineers and safety assessors. Further, it is extremely difficult, due to page limit constraints, to present a complete assurance case for the system. Nevertheless, the study has considered a diverse set of lifecycle data which is representative of the different stages of development and assessment: requirements specification, design, implementation and verification as well as how safety analysis is considered at each of these stages. The study also considered both the product and process perspectives of certification that often form necessary parts of software safety assurance. Finally, experimenter bias is always a key threat to research studies. To limit the degree of bias, the criteria against which the study was performed were independently reviewed through a set of questionnaires that were completed by practicing engineers and assessors. Further, the chain of reasoning between the study data and results is documented in detail in order to facilitate a critical evaluation by readers, including the provision of counter-evidence and counter-interpretations. This is inevitable given the qualitative nature of this type of research.

External validity: Our study has focused on an airborne system to which compliance with the DO-178C requirements is applicable. Of course, more studies based on systems from other safety–critical industries are needed before being able to generalise the results of our study. However, the process specified in DO178C is very similar to processes specified in other prescriptive standards, e.g. the automotive standard ISO 262626 (ISO, 2011). Further, DO178C, and more specifically DO-178B, has often been used in other domains, e.g. in defence (MoD, 2007). Finally, GSN was used to represent the assurance argument fragments for the WBS. These argument fragments are notation independent and as such can equally be represented in textual or tabular formats.

6. Conclusions

As interest in the application of assurance cases to software systems has grown, so has the debate surrounding the relative merits of an assurance argument approach and a more traditional (prescriptive) approach such as defined in the DO178C guidance. As this paper has described, there can in fact be seen to be a role for both in a successful software assurance regime. An approach such as that defined in DO178C provides very clear guidance on the processes and techniques that may be adopted in a particular domain. An explicit assurance argument provides a means of demonstrating and justifying the sufficiency of the evidence for a specific system context; for example, this might include justifying the methods and tools chosen for a DO178C process. The authors acknowledge the challenge of developing a compelling assurance argument for complex software systems, particularly for those with limited experience of developing assurance arguments; it is clear that the flexibility enabled by the use of assurance cases also introduces uncertainty in terms of what is sufficient argument and evidence. It is our intent that published guidance such as the software safety argument pattern catalogue in (Hawkins and Kelly, 2010) can help in this regard, and our sketch of two different ways of linking assurance cases and DO178C processes may also help. However, there remains further work to be done in this area; not the least, more work is needed to assess the predictability and effort associated with the different approaches. Finally, the initiatives to standardise assurance cases across domains (OMG, 2010) should also prove to be helpful in encouraging the sharing of experience and expertise between all those industrial sectors that employ assurance cases.

References

Basir, N., Denney, E., Fischer, B., 2008. Constructing a safety case for automatically generated code from formal program verification information. In: 27th International Conference on Computer Safety, Reliability and Security (SAFECOMP '08), Newcastle, England.

Basirm, N., Denney, E., Fischer, B., 2010. Deriving safety cases for hierarchical structure in model-based development. In: 29th International Conference on Computer Safety, Reliability and Security (SAFECOMP '10), Vienna, Austria.

Bloomfield, R., Bishop, P., 2010. Safety and assurance cases: past, present and possible future – an Adelard perspective. In: 18th Safety–Critical Systems Symposium, Bristol, UK.

Bloomfield, R., 2005. Assurance Cases for Security. Workshop Report v01c. <http://www.csr.city.ac.uk/AssuranceCases/Assurance_Case_WG_Report_180106_v10.pdf> (accessed 03.02.11).

Butler, R., Finelli, G., 1991. The infeasibility of experimental quantification of life-critical software reliability. IEEE Transactions on Software Engineering 19(1).

CAA, 2007. CAP 670 – Air Traffic Services Safety Requirements. Civil Aviation Authority (CAA), Safety Regulation Group.

CAST, 2006. Verification Independence. Position Paper, Certification Authorities Software Team (CAST), CAST-26.

Cave, C.H., 2009. An Independent Review into the Broader Issues Surrounding the Loss of the RAF Nimrod MR2 Aircraft XV230 in Afghanistan in 2006. The Stationary Office, London.

Denney, E., Pai, G., Habli, I., 2012. Perspectives on software safety case development for unmanned aircraft. In: 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012), Boston, Massachusetts USA.

Dobbing, B., Lautieri, S., 2006. SafSec Methodology: Standard. S.P1199.50.2 Issue 3.1, Praxis High Integrity Systems.

FDA, 2010. Guidance for Industry and FDA Staff – Total Product Life Cycle: Infusion Pump – Premarket Notification. Food and Drug Administration Draft Guidance.

Goodenough, J., Lipson, H., Weinstock, C., 2007. Arguing Security – Creating Security Assurance Cases. Software Engineering Institute (SEI). <https://buildsecurityin.us-cert.gov/bsi/articles/knowledge/assurance.html> (accessed 03.02.11).

Graydon, P., Knight, J., Strunk, E., 2007. Assurance based development of critical systems. In: 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.

Graydon, P., Habli, I., Hawkins, R., Kelly, T., Knight, J., 2012. Arguing Conformance, IEEE Software 29(3).

Greenwell, W., Knight, J., Holloway, M., 2006. A taxonomy of fallacies in system safety arguments. In: 2006 International System Safety Conference (ISSC '06), Albuquerque, NM, USA.

GSN Committee, 2010. Draft GSN Standard. Version 1.0. <http://www.goalstructuringnotation.info/> (accessed 19.11.10).

Habli, I., Kelly, T., 2007. Achieving integrated process and product safety arguments. In: 15th Safety Critical Systems Symposium (SSS'07), Bristol, United Kingdom.

Habli, I., Kelly, T., 2008a. A model-driven approach to assuring process reliability. In: 19th IEEE International Symposium on Software Reliability Engineering (ISSRE), Seattle, USA.

Habli, I., Kelly, T., 2008b. A generic goal-based certification argument for the justification of formal analysis. SafeCert 2008: Certification of Safety–Critical Software Controlled Systems. Budapest, Hungary.

Habli, I., Hawkins, R., Kelly, T., 2010. Software safety: relating software assurance and software integrity. International Journal of Critical Computer-Based Systems (IJCCBS) 1 (4).

Hawkins, R., 2007. The who, where, how, why and when of modular and incremental certification. In: IET System Safety Conference, London.

Hawkins, R., Kelly, T., 2010. A systematic approach for developing software safety arguments. Journal of System Safety 46 (4), 25–33 (System Safety Society Inc.).

Hawkins, R., Clegg, K., Alexander, R., Kelly, T., 2011. Using a software safety argument pattern catalogue: two case studies. In: 30th International Conference on Computer Safety, Reliability and Security (SAFECOMP '11), Naples, Italy.

Hayhurst, K., et al., 1998. Streamlining Software Aspects of Certification: Technical Team Report on the First Industry, Workshop, NASA/TM-1998-207648.
Hayhurst, K., Veerhusen, D.S., Chilenski, J.J., Rierson, L.K., 2001. A Practical Tutorial Decision Coverage. NASA/TM-2001-210876, NASA Report.
HSE, 2006. Safety Assessment Principles for Nuclear Facilities. Health and Safety Executive (HSE).
IAWG, 2007. Industrial Avionics Working Group Modular Software Safety Case Process. Industrial Avionics Working Group (IAWG) IAWG-AJT-301 Issue 2. <http://www.assconline.co.uk/iawg.asp> (accessed 03.02.11).
IEC, 2010. 61508 ed2.0 – Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems. International Electrotechnical Commission (IEC).
ISO, 2011. ISO26262 Road Vehicles – Functional Safety. Draft, FDIS, International Organization for Standardization (ISO).
ISO/IEC 15026-2, 2010. Systems and Software Engineering. Systems and Software Assurance. Part 2. Assurance case. Draft for Public Comment.
Jackson, D., Thomas, M., Millett, L.I., 2007. Committee on Certifiably Dependable Software Systems. National Research Council, Software for Dependable Systems: Sufficient Evidence? The National Academies Press.
Johnson, C., Holloway, M., 2007. The Dangers of Failure Masking in Fault-Tolerant Software: Aspects of a Recent In-Flight Upset Event 2nd Institution of Engineering and Technology International Conference on System Safety.
Kelly, T., 1998. Arguing Safety – A Systematic Approach to Managing Safety Cases. Dissertation. Department of Computer Science, The University of York.
Kelly, T., 2001. Concepts and Principles of Compositional Safety Case Construction. Technical Report COMSA/2001/1/1. The University of York.
Kelly, T., 2007. Reviewing Assurance Arguments – A Step-by-Step Approach. Workshop on Assurance Cases for Security – The Metrics Challenge, Dependable Systems and Networks (DSN).
Knight, J., 2008. Advances in Software Technology Since 1992 FAA 2008 National Software and Airborne Electronic Hardware Conference, Denver, CO.
Lee, J.S., Katta, V., Jee, E.K., Raspotnig, C., 2010. Means-ends and whole-part traceability analysis of safety requirements. Journal of Systems and Software 83 (9), 1612–1621.
Littlewood, B., Wright, D., 2007. The use of multilegged arguments to increase confidence in safety claims for software-based systems: a study based on a BBN analysis of an idealized example. IEEE Transactions on Software Engineering 33 (5), 347–365.
Lutz, R.R., Mikulski, I.C., 2003. Operational anomalies as a cause of safety–critical requirements evolution. Journal of Systems and Software 65 (2), 155–161.
Menon, C., Hawkins, R., McDermid, J., 2009. Interim Standard of Best Practice on Software in the Context of DS 00-56 Issue 4. SSEI-BP-000001 Issue 1, Software Systems Engineering Initiative (SSEI). <www.ssei.org.uk> (accessed 03.02.11).
McDermid, J., 2001. Software Safety: Where's the Evidence? Australian Workshop on Industrial Experience with Safety Critical Systems and Software.
McDermid, J., Pumfrey, D., 2001. Software safety: why is there no consensus? In: 19th International System Safety Conference. System Safety Society, Huntsville, AL.
MoD, 2007. Defence Standard 00-56 Issue 4: Safety Management Requirements for Defence Systems. UK Ministry of Defence (MoD).
Object Management Group, 2010. Argumentation Metamodel (ARM). Draft Specification, OMG Document: ptc/2010-08-36.
Penny, J., Eaton, A., Bishop, P., Bloomfield, R., 2001. The practicalities of goal-based safety regulation. In: 9th Safety–Critical Systems Symposium Bristol, UK.
Rail Safety and Standards Board, 2007. Engineering Safety Management (The Yellow Book). <http://www.yellowbook-rail.org.uk/> (accessed 03.02.11).
Redmill, F., Chudleigh, M., Catmur, J., 1999. System Safety: HAZOP and Software HAZOP. Wiley, New York.
Rushby, J., 2010. Formalism in safety cases. In: 18th Safety–Critical Systems Symposium, Bristol, UK.
RTCA, 2012. DO178C – Software Considerations in Airborne Systems and Equipment Certification. Radio and Technical Commission for Aeronautics (RTCA).
Runeson, P., Höst, M., 2009. Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering Journal 14 (2), 131–164.
SAE, 1996a. Aerospace Recommended Practice 4754: Certification Considerations for Highly-Integrated or Complex Aircraft Systems. Society of Automotive Engineers (SAE).
SAE, 1996b. ARP4761 – Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment. Society of Automotive Engineers (SAE).
Wassyng, A., Maibaum, T., Lawford, M., Bherer, H., 2010. Software Certification: Is There A Case Against Safety Cases? Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems, Redmond, WA, USA.
Weaver, R., 2003. The Safety of Software – Constructing and Assuring Arguments. Dissertation. Department of Computer Science, The University of York.
Weinstock, C., Goodenough, J., 2009. Towards an Assurance Case Practice for Medical Devices. CMU/SEI-2009-TN-018.
Ye, F., 2005. Justifying the Use of COTS Components within Safety Critical Applications. Dissertation. Department of Computer Science, The University of York.
Yin, R.K., 2003. Case Study Research Design and Methods, third ed. Sage, London.