Facilitating the Maintenance of Safety Cases

Omar Jaradat
Doctoral Student
Mälardalen University
Högskoleplan 1, 721 23
Västerås, Sweden
+46-21-101369
omar.jaradat@mdh.se

Iain Bate
Senior Lecturer
University of York
Deramore Lane, York
YO10 5GH, UK
+44-1904-325572
iain.bate@cs.york.ac.uk

Sasikumar Punnekkat
Professor
Mälardalen University
Högskoleplan 1, 721 23
Västerås, Sweden
+46-21-107324
sasikumar.punnekkat@mdh.se

ABSTRACT

Developers of some safety critical systems construct a safety case comprising both *safety evidence*, and a *safety argument* explaining that evidence. Safety cases are costly to produce, maintain and manage. Modularity has been introduced as a key to enable the reusability within safety cases and thus reduces their costs. The Industrial Avionics Working Group (IAWG) has proposed Modular Safety Cases as a means of containing the cost of change by dividing the safety case into a set of argument modules. IAWG's Modular Software Safety Case (MSSC) process facilitates handling system changes as a series of relatively small increments rather than occasional major updates. However, the process doesn't provide detailed guidelines or a clear example of how to handle the impact of these changes in the safety case. In this paper, we apply the main steps of MSSC process to a real safety critical system from industry. We show how the process can be aligned to ISO 26262 obligations for decomposing safety requirements. As part of this, we propose extensions to MSSC process for identifying the potential consequences of a system change (i.e., impact analysis), thus facilitating the maintenance of a safety case.

Keywords

Safety Case, Safety Argument, Maintenance, Impact Analysis, Change, IAWG MSSC.

1. INTRODUCTION

Constructing safety cases receives significant industrial attention as it is required for the certification process of many safety critical system domains. A safety case comprises both safety evidence (e.g. safety analyses, software inspections, or functional tests) and a safety argument explaining that evidence. Safety arguments show how system developers use each item of evidence to support claims, and how those claims, in turn, support broader claims about system behaviour, hazards addressed, and, ultimately, acceptable safety [1]. The production, management and evaluation of safety cases are considered difficult to achieve and time consuming. As an anecdotal example, the size of the preliminary safety case for surveillance on airport surfaces with ADS-B [2] is about 200 pages, and it is expected to grow as the operational safety case is created [3].

It is worth noting that a safety case is a living document that grows as the system grows. A safety case should be maintained as needed whenever some aspect of the system, its operation, its operating context, or its operational history changes.

Operational or environmental changes may invalidate a well-founded safety argument for different reasons as follows:

1. Changing the argument structure
2. Evidence is valid only in the operational and environmental context in which it is obtained, or to which it applies. During or after a system change, evidence might no longer support the developers' claims because it could reflect old development artefacts or old assumptions about operation or the operating environment
3. In the updated system, existing safety claims might be nonsense, no longer reflect operational intent, or they might be contradicted by new data

The certification process must be repeated after applying changes to an already certified system (i.e., re-certification). In other words, the safety case of the certified system should show that the system is acceptably safe to operate in its intended context after applying the changes. In order to achieve the re-certification, a safety argument should be maintained by determining whether the item of evidence still supports the claims made about it, check whether new or updated safety requirements are reflected in the argument, and review the overall logic of the argument. The main problem though is that the elements of the safety argument (i.e., safety goals, evidence, argument and the operating context) are highly interdependent so that what can be seen as a minor change in the argument may have a major impact to the contents and the structure of that argument [12]. Hence, maintaining a safety argument requires high awareness of the dependencies among its contents and how a change to one part may invalidate other parts. Without this vital awareness, a developer performing impact analysis might not notice that a change has compromised system safety. The Ariane 5 rocket which crashed forty seconds after take-off in 1996 is a costly example of omitting affected parts of a system due to a change. Ariane 5 inertial reference system (SRI) tried to stuff a 64-bit number into a 16-bit space which led to a conversion error. This part of the system was reused from an older version of the SRI that was implemented for Ariane 4 rocket. Seemingly, an assumption was made as since the code was successfully used in an older version of the system then it is suitable to be reused for the newer version [15]. Hence, system developers focused on more complex parts of the system and no attention was paid to the out-of-date code or to any related assumption.

A fundamental step prior to update a safety case due to a change is to assess the impact of this change in the safety argument. This is referred to as safety case impact analysis. It is probably clearer now how the continuous maintenance efforts to keep the safety case always up-to-date add more burden on top of the discussed difficulties above. Moreover, the cost of change has become a major part of the cost of ownership of a system [4].

As a response to these challenges, an ambition emerged to modularize safety cases by applying the principles of software architecture and design to the safety case domain. The main idea of the modularity is to align boundaries of safety case modules with design boundaries to contain changes. Having done that, a change to a design element should then affect the corresponding safety case module, and not impact the entire safety argument [4].

To this end, the Industrial Avionics Working Group (IAWG) represented by a team of highly experienced engineers, experts in software development and safety assurance, defined the Modular Software Safety Case (MSSC) process [5] as a means for containing the cost of change by dividing the safety case into a set of argument modules. The process has been refined through experience gained from large-scale trial applications of the prototype process, and further trials of the refined process. MSSC process establishes component traceability mechanism between system design elements and safety argument modules by using the concepts of Dependency-Guarantee Relationship (DGR) and Dependency-Guarantee Contract (DGC). The former is to highlight, and describe, safety-related properties and behaviour of a single design element. In other words, DGRs capture the relationships between input and output ports for each design element. A DGC, however, is used to match one design element's dependencies with another design element's guarantees [6].

The contributions of this paper are as follows: demonstrating how to apply the IAWG MSSC process. More specifically, apply the process to the Fuel Level Estimation System (FLES), which is a real safety critical system that was implemented by Scania AB — a major Swedish automotive industry manufacturer — to show (1) how the DGR and DGC concepts can be used to capture the safety requirements of the FLES, (2) how these two concepts can be used to build a safety case in conformance to the requisites of ISO 26262 for certification, and (3) extending IAWG's DGC to improve the impact analysis process thus facilitating the maintenance of safety cases.

This paper is composed of four further sections. In Section 2 we present background information. In Section 3 we present the IAWG MSSC process. In Section 4 we use the FLES to demonstrate the application of the IAWG MSSC process. Finally, in Section 5 we draw conclusions and identify future work.

2. BACKGROUND

This section presents background information about the safety standard ISO 26262, the Goal Structuring Notation (GSN), safety case maintenance and current challenges, and an approach to maintaining safety case evidence after a system change.

2.1 The Safety Standard ISO 26262

The rationale behind the selection of this standard for this work is that it is functional safety standard was adapted for automotive electric/electronic systems that Scania is working to qualify for its certification stamp. Since FLES is one of other systems in Scania's trucks, it is very appropriate to consider ISO 26262 for the given example in this paper.

ISO 26262 regulates the automotive domain, more specifically, the standard is intended to be applied to safety-related systems that include one or more electrical and/or electronic systems and that are installed in series production passenger cars with a maximum gross vehicle mass up to 3500 kg [7]. In this subsection, however, we focus only on the part of the standard that regulates the decomposition of safety requirements. The following parts are summarized descriptions of the safety requirements decomposition directly from ISO 26262 guidelines:

1. Successively after identifying hazards, the standard recommends to formulate the Safety Goals (SGs) related to the prevention or mitigation of the hazardous events, in order to avoid unreasonable risk. Basically, hazard analysis, risk assessment and Automotive Safety Integrity Level (ASIL) are used to determine the safety goals such that an unreasonable risk is avoided. The standard defines a safety goal as a top-level safety requirement resultant of the hazard analysis and risk assessment. Safety goals are not expressed in terms of technological solutions, but in terms of functional objectives. [7]
2. Identification of safety goals leads to the functional safety concept. The objective of the functional safety concept is to derive the Functional Safety Requirements, from the safety goals, and to allocate them to the preliminary architectural elements. To comply with the safety goals, the functional safety concept contains safety measures, including the safety mechanisms, to be implemented in the item's architectural elements and specified in the functional safety requirements. The standard defines a functional safety requirement as a specification of implementation-independent safety behaviour, or implementation-independent safety measure, including its safety-related attributes. [7]
3. Finally, both the functional concept and the preliminary architectural assumptions lead to the technical safety concept. The first objective of this concept is to specify the Technical Safety Requirements and their allocation to system elements for implementation by the system design. The second objective is to verify through analysis that the technical safety requirements comply with the functional safety requirements. The standard defines a technical safety requirement as a requirement derived for implementation of associated functional safety requirements. [7]

2.2 The Goal Structuring Notation (GSN)

A safety argument organizes and communicates a safety case, showing how the items of safety evidence are related and collectively demonstrate that a system is acceptably safe to operate in a particular context. The GSN [8] provides a graphical means of communicating (1) safety argument elements, claims (goals), argument logic (strategies), assumptions, context, evidence (solutions), and (2) the relationships between these elements. The principal symbols of the notation are shown in Figure 1 (with example instances of each concept).

A goal structure shows how goals are successively broken down into ("solved by") sub-goals until a point is reached where claims can be supported by direct reference to evidence. Using the GSN, it is also possible to clarify the argument strategies adopted (i.e., how the premises imply the conclusion), the rationale for the approach (assumptions, justifications) and the context in which goals are stated. It is worth noting that GSN has been extended to enable modularity in a safety case (i.e., module-based development of the safety case). Hence, modular GSN enables the partitioning of a safety case into an interconnected set of modules.

2.3 Safety Case Maintenance and Current Challenges

A safety case is a living document that should be maintained

whenever some aspect of the system, its operation, its operating context, or its operational history changes. In this paper, the process of updating the safety case after implementing a system change is referred to as safety case maintenance.

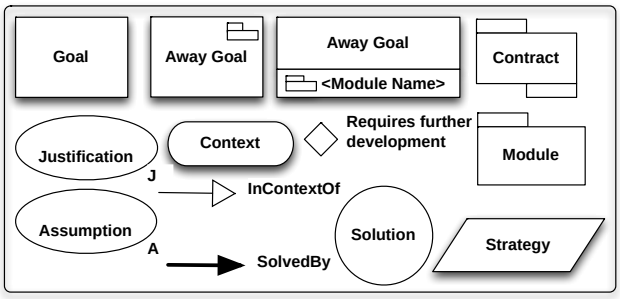

Figure 1. Overview of Goal Structuring Notation (GSN)

Developers of safety critical systems experience difficulties in safety case maintenance after implementing a system change. One of the main difficulties is identifying the impacted parts in the safety argument. The traceability between a system design and the corresponding safety argument contents, and the dependency among the contents of safety argument are considered two main burdens that encounter the identification of the impacted parts in an argument. Moreover, individual systems tend to become more complex as they are designed and constructed, this increasing complexity, as well as, the number of evidence items in a safety argument can exacerbate the maintenance difficulties. Any approach intends to manage safety argument due to system changes should consider:

1. A means for clearly capturing the underlying rationale of the safety argument in order to assess the impact of change on all parts of the argument
2. A traceability mechanism between a system domain and the safety argument to support the ability to track the changed part from the system design down to the corresponding affected part in the safety argument
3. Mechanisms to structure the argument so as to contain the impact of changes

The use of the GSN approach helps to produce well-structured arguments that clearly demonstrate the argument elements and their interdependencies (the relationships between the argument claims and evidence) [10]-[12]. Using GSN makes capturing the underlying rationale of the argument easier, which will in turn, help to scope areas affected by a particular change and thus helps the developers to mechanically propagate the change through the goal structure. However, GSN does not tell if the suspect elements of the argument in question are still valid. For example, having made a change to a model we must ask whether goals articulated over that model are still valid. Expert judgment, therefore, is still required in order to answer such questions. Hence, using GSN does not directly help to maintain the argument after a change, but it can more easily determine the questions to be asked to do so [11].

Current standards and analysis techniques assume a top-down development approach to system design. For component-based systems, monolithic evidence produced via these approaches is difficult to maintain those systems because it is hard to match a safety argument that has a different structure than the system design structure. However, safety is a system level property and assuring this property requires every piece of evidence generated for each component to be linked and compared to demonstrate consistency [5]. One may think that the matching (i.e., optimal level of traceability) can be achieved by designing a safety argument structure to be similar to the system design structure, where a clear one-to-one mapping of a system design component to a safety argument module can be established (see Figure 2).

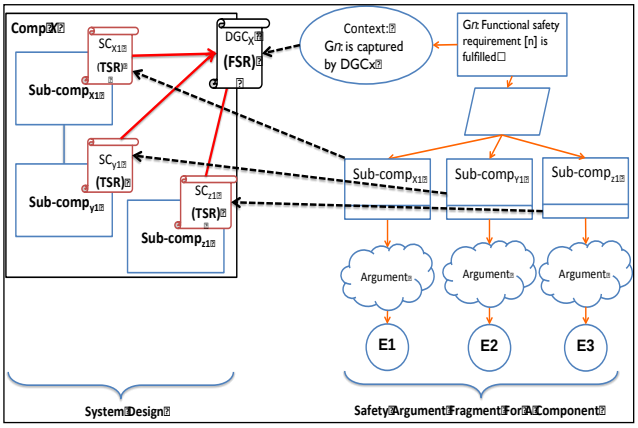

Figure 2. An illustration of the relationship between a system design and its safety argument

Theoretically, a one-to-one mapping may facilitate tracking down the components of a system design to the safety argument, but it is impractical due to four key factors: (1) modularity of evidence, (2) modularity of the system, (3) process demarcation (e.g., ISO 26262 items [7]), and (4) organisational structure (e.g., who is working on what). These factors have a significant influence when deciding upon the safety argument structure.

Enabling component and evidence traceability is very useful to analyse the impact of change on a safety argument, and eventually, facilitates the overall maintenance of the safety case. This paper deals with two forms of traceability: component (i.e. safety argument fragment to system design component) and evidence (i.e. safety argument fragment to supporting evidence). However, to the best of our knowledge there are no supporting process or method that provides detailed steps of how to analyse the impact of a change on a safety case using component or evidence traceability. That said there are well-regarded industry-lead initiatives that assume such methods exist. MSSC Process is one such example.

In this paper, we use the word "traceability" to indicate two different things. Firstly, we refer to the ability to relate safety argument fragments to system design components as component traceability mechanism (through a safety argument). Secondly, we refer to the ability to relate safety argument evidence across system's artefacts as evidence traceability.

2.4 Maintaining Safety Case Evidence after a System Change

In our previous work [1], we proposed a new approach to facilitating safety case change impact analysis. In the approach, automated analysis of information given as annotations to a safety argument (recorded in the GSN) highlight suspect safety evidence to bring it to engineer's attention. We proposed annotating each reference to a development artefact (e.g. an architecture

specification) in a goal or context element with an artefact version number.

We also proposed annotating each solution element with:

1. An evidence version number
2. An input manifest identifying the inputs (including version) from which the evidence was produced
3. The lifecycle phase during which the evidence obtained (e.g. Software Architecture Design)
4. A safety standard reference to the clause in the applicable standard (if any) requiring the evidence (and setting out safety integrity level requirements)

With this data, we can perform a number of automated checks to identify items of evidence impacted by a change. For example:

1. We can determine when two different versions of the same item of evidence are cited in the same argument
2. We can identify out-of-date evidence by searching for input manifests $m = \{(a1, v1),\ldots, (an, vn)\}$ and artefact versions (a, v) such that $\exists i \bullet a = ai \wedge v > vi$
3. Where we know a particular artefact has changed, we can search for input manifests containing old versions

If we had further information which inputs were used to produce each input listed in each input manifest, each input that was used to produce those, and so on, we could extend checks (2) and (3) above to indirect inputs. For example, suppose that life testing is used to establish the reliability of a component, that this component and its reliability appear in a Failure Modes and Effects Analysis (FMEA), and that the FMEA results are used in a Fault Tree Analysis (FTA). With the additional information, we could compute a closure of the FTA's input manifest that would include the life testing results. Other analyses may be possible. For example, we suggest storing the safety standard reference to facilitate analysis of impacts that change the safety integrity level of a requirement.

3. MODULAR SOFTWARE SAFETY CASE (MSSC) PROCESS

IAWG has proposed Modular Safety Cases as a means of containing the cost of change by dividing the safety case into a set of argument modules. IAWG's MSSC process facilitates handling system changes as a series of relatively small increments rather than occasional major updates (i.e., incremental certification). MSSC process manages system changes by breaking down a system into blocks. The process defines the block as an identifiable part (or group of parts) of the Software implementation that is chosen by the safety case architect to be the subject of a safety case module. Blocks cover all parts of a system design where each block may correspond to a single or multiple software component or unit of design, but it is subject to only one dedicated safety case module. In other words, each system block has one-to-one relationship with a safety argument module. [5]

The process establishes component traceability mechanism between system blocks and safety argument modules by using the concepts of DGR and DGC as shown in Figure 3 and 4, respectively. The former is to highlight and describe safety-related properties and behaviour of a system block. In other words, a DGR captures the relationships between input and output ports for each design block. A DGC, however, is used to match one block's dependencies with another block's guarantees [5][9]. Creating DGCs leads to the creation of a 'daisy chain' as a dependency in one block and a guarantee offered by another, whose associated dependencies are supported by further guarantees, and so on [9].

MSSC process is very dependent on the anticipated changes that should be identified in the first step of the process. The anticipated change scenarios will bring the highly likely changeable parts in the system to developer's attention.

Dependency — Guarantee Relationship |		*[Reference Name]*		
Guarantee				
Concise Definition		**Definitive Context**	**Incidental Note**	**Traceability**
[Guarantee]		*[Definitions] [Ports description]*	*[Note]*	*[Req. No.]*
Related Dependencies				
No	**Concise Definition**	**Definitive Context**	**Incidental Note**	**Traceability**
1	*[Dependency]*	*[Definitions] [Ports description]*	*[Note]*	*[Req. No.]*

Figure 3. A DGR tabular representation

Dependency – Guarantee Contract			**<Block Name>.<DGC Name>**
Consumer Dependency	**Integrator**	✓	**Provider Guarantee**
<Block A Name>.<DGR Name>.<Data1> Dependency	**has SC Contract with**		**<Block B Name>.<DGR Name>.Guarantee of <Data2> Provision**
Block A <data1> needed	is supported by		Block B <data2> provided
<data1 units>	is consistent with		<data2 units>
Northern hemisphere only	is consistent with		North of the equator
…	is consistent with		…

Figure 4. A DGC tabular representation

These scenarios are considered by system developers so that they can manage the containment of the impact of these changes in the system blocks boundaries more efficiently. Having done this, the impact of a change in one safety argument module will hopefully not propagate into another module, but it might impose one (or more) safety case contract update, and even if it is then the cost of changes can be minimised.

It is very important to distinguish between a DGC and a safety case contract. The former captures the required link between a dependency declared in one DGR and a satisfying guarantee provided by another. Hence, DGCs are created on the system design level. A safety case contract, however, is used to describe the linkage between a consumer goal in one Safety Case Module and a provider goal in another [5]. This is formed through the new GSN extension for modularity.

Figure 5 shows an example to describe the relationships between system blocks, DGR, DGC, safety case contract and the safety case architecture. It is worth noting that DGCs may be linked to safety case contracts.

The following is a list summarises MSSC process's steps [5]:

Step 1. Analyse the product lifecycle: it is important to predict the potential change scenarios over the projected system lifetime. One reason for that is because change scenarios will help assess the potential benefits that may be achieved through modular certification. If as a result of the analysis there are no changes expected, then the full benefits of modular certification may not be realised, and it may therefore be decided not to adopt a modular approach. [9]

Step 2. Optimise software design and safety case architecture: since each system block is subject to safety case module.

First, we need to divide the system into blocks and form public interfaces for the block safety case modules. All elements of the system are split into blocks and each corresponding block safety case module should present an argument about the safety-related behaviour of that block. Second, other necessary modules will be added, for example, software safety requirements, software system wide issues module, configuration data module, safety case contract modules, etc. Finally, we should define safety case integration modules — these provide the argument about the combined behaviour of interdependent safety case modules. [5]

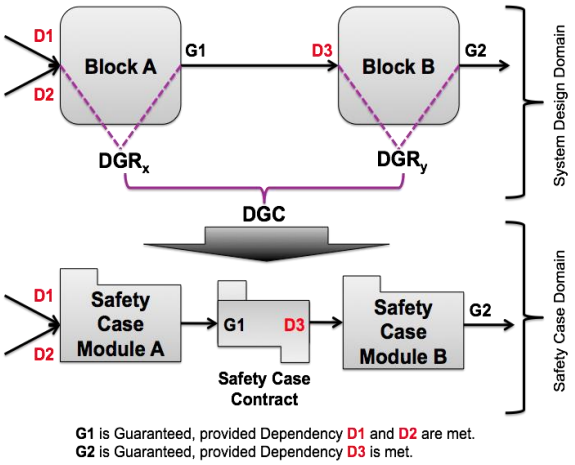

Figure 5. Linking blocks using DGRs and DGCs

Step 3. Construct safety case modules: A hazard mitigation argument should be formed and derived safety requirements are directed to SW blocks safety case modules. The guaranteed behaviour offered by each block in support of these is captured, along with dependencies on other blocks. A Block Safety Case Module is constructed providing argument and evidence for each Block based on the Guarantees and Dependencies. [5]

Step 4. Integrate safety case modules: the safety case modules are integrated so that claims requiring support in one Safety Case Module are linked to claims providing that support in others. This step of the process results in a fully integrated Safety Case. [5]

Step 5. Assess/Improve change impact: when a system change is implemented, the impact on the design modules and associated Safety Case Modules is assessed. [5]

Step 6. Reconstruct safety case modules
Step 7. Reintegrate safety case modules
Step 8. Appraise the safety case

The guidance of MSSC process [5] does not show detailed information about how to follow some steps including the impact analysis part. The provided example by the process abstracts the impact analysis step and shows its results only. The main work in this paper is not to consider all parts of MSSC process to give a full example on how to apply them but we rather focus on the impact analysis part and necessary prerequisite steps only.

4. ILLUSTRATIVE EXAMPLE: FUEL LEVEL ESTIMATION SYSTEM (FLES)

In our previous work [13] [14], we used FLES as a specimen system to illustrate the contribution of the architectural model checking to conduct preliminary safety assessment in line with the safety standard ISO 26262.

We used the Architecture Analysis & Design Language (AADL) to model the system as shown in Figure 6. In our current work we reuse the description as well as the AADL of FLES to partially apply MSSC process. We also propose a system change scenario and examine how the method helps to highlight the affected safety argument elements.

4.1 FLES Description

4.1.1 FLES Technical details

FLES estimates the volume of fuel in a heavy road vehicle's tank and presents this information to the driver through a dashboard mounted fuel gauge. Additionally, the system must warn the driver when this volume falls below a predefined threshold. This system is considered safety critical because its failure could lead to loss of control of the vehicle. For example, if there is less fuel remaining than the driver thinks, the vehicle might run out, bringing it to an unexpected halt, which can be hazardous in certain contexts. As well as bringing the vehicle to a halt, the power steering and braking mechanisms could also fail. These failures would compromise vehicle controllability and could also lead to a crash.

Fuel volume is estimated using a float sensor in the fuel tank. As the position of the float is affected by vehicle motion (negotiating steep hills, sharp bends, or rough terrain), the system has some challenging issues to be tackled within its design. The system must process this signal so that at all times the gauge displays an accurate measurement of the total volume of fuel remaining. The sensed value is sent to the *Estimator ECU*. An Analogue to Digital Converter (ADC) is used to convert and then the *SoftwareIN* thread reads the sensed fuel float position from the ADC and stores it in the real-time database *RTDB*. *FuelEstimation* reads this sensor value and computes an estimate of the current fuel volume in litres. When the vehicle might be moving (i.e., its parking brake is not set), the *FuelEstimation* thread uses a Kalman filter algorithm to reduce the noise introduced by vehicle motion. This algorithm requires the recent history of fuel volume estimates to be stored. *FuelEstimation* outputs a smoothed fuel volume estimate to the *RTDB*. *FuelLevelWarning* then reads this estimate, compares it to the low-fuel warning threshold (i.e., < 7% of the fuel tank capacity), and writes the low-fuel warning status to the *RTDB*. *SoftwareOUT* reads the fuel volume and low-fuel warning status from the *RTDB* and sends these over the Controller Area Network (CAN) bus to the *Presenter ECU*. The *Presenter ECU* adjusts the actuators (i.e., fuel gauge and low-fuel lamp) on the dashboard according to the received values.

4.1.2 FLES safety analysis

Hazard analysis and risk assessment made for FLES led to one hazard identification: "*Unannunciated lack of fuel*". Unannunciated is interpreted as (1) fuel estimates and low-fuel warning are not displayed at all, and (2) it is displayed incorrectly since the estimates are not identical to the real amount of fuel in

the vehicle's tank. The determined ASIL for the fuel level estimation system is "C".

The derived safety requirements to mitigate the hazard are decomposed as recommended by ISO 26262 as follows:

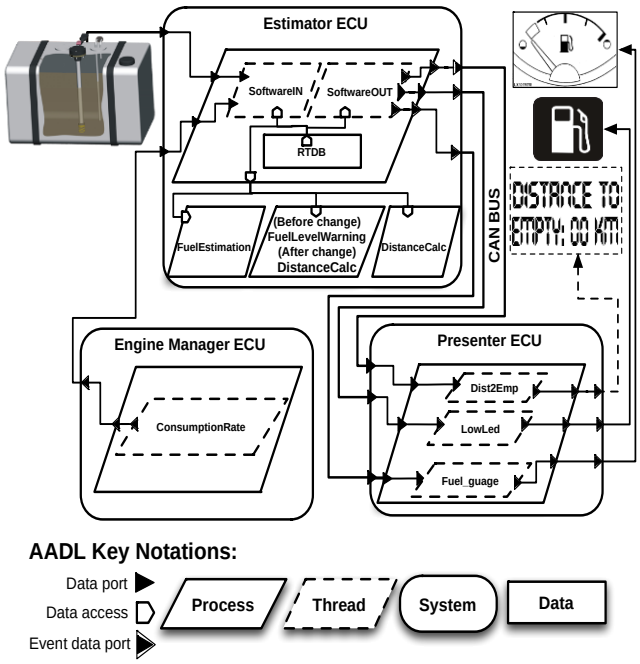

Figure 6. An AADL representation of Estimator's software architecture

1. ***Safety goals***: two safety goals were derived
   a. *SG1.0ImplAssur:* Vehicle's driver shall be constantly aware of the actual remaining fuel in the tank whenever the engine is in operation
   b. *SG2.0ImplAssur:* Vehicle's driver shall be warned when the fuel level is low and the engine is in operation
2. ***Functional Safety Requirements (FSR)***:
   Two functional safety requirements were identified to satisfy *SG1.0ImplAssur:*
   a. *ConFSR1.0.1.0:* A fuel gauge should promptly annunciate the actual fuel amount in the tank whenever the engine is in operation
   b. *ConFSR1.0.2.0:* The fuel gauge shall not display a fuel estimate that deviates more that 5% from the actual fuel volume in the tank
   One functional safety requirement was identified to satisfy *SG2.0ImplAssur:*
   c. *ConFSR2.0.1.0:* A fuel-low warning lamp should be promptly turned ON when the fuel level in the tank falls below a certain level whenever the engine is in operation
3. ***Technical Safety Requirements (TSR)***: There is a large set of technical safety requirements that was identified to specify the functional safety requirements. The work of the paper, however, considers the minimum set of technical safety requirements that specify *ConFSR1.0.1.0* and *ConFSR2.0.1.0* as shown in Table 1.

Table 1. A Subset of the identified TSRs for FLES

FSR ID	TSR ID	Description
FSR1.0.1.0	*F1010TSR1*	The *FuelEstimation* thread shall provide the *totalFuelLevel* value
FSR1.0.1.0	*F1010TSR2*	The *SoftwareOUT* shall send the *totalFuelLevel* value to the *Presenter*
FSR2.0.1.0	*F2010TSR1*	The *FuelLevelWarning* thread shall provide *lowFuelWarning* value
FSR2.0.1.0	*F2010TSR2*	The *SoftwareOUT* shall send the *lowFuelWarning* value to the *Presenter*

4.2 Applying the IAWG MSSC Process

A list of anticipated change scenarios during FLES's lifetime is required. This list may help assessing the potential benefits that may be achieved through modular certification. In this section, we present the details of the various MSSC process steps with respect to FLES:

4.2.1 Analyse the product lifecycle and identify change scenarios

We assume one potential change for FLES. The *Distance To Empty* feature might be added to FLES. The role of this anticipated change is to determine the distance (Km) that a vehicle can drive before it runs out of fuel. This new feature is dependent on (1) the estimation of the current fuel amount in the tank (L), and (2) the fuel consumption rate (L/Km) in the engine. Technically, this intended feature will be added as a new thread in the *Estimator ECU.* This thread should read the output of the *FuelEstimation* thread, as well as, the output of the *ConsumptionRate* thread that is implemented in the *EngineManager ECU.* To avoid dealing with timing and memory budgets, FLES engineers expect to remove the *FuelLevelWarning* thread and move the task it contains to the *FuelEstimation* thread (i.e., merge the two threads into one). Since the safety margin of the *FuelEstimation* thread allows adding a new task, the timing and memory budget for the thread will remain the same even after the merge. On the other hand, the new *DistanceCalc* thread will take the timing and memory budget, and the priority of the removed *FuelLevelWarning* thread. The same arrangements will be applied to the threads in the *Presenter ECU*.

4.2.2 Optimise software design and safety case architecture (define the safety case architecture)

For the sake of simplicity, we do not define a full set of the safety case modules, but we rather define the basic modules that are sufficient to make the example. We focus on the *Estimator* in our example by dividing it into two software blocks, namely, *FuelEstimationBK* and *FuelLevelWarningBk*. Each of them represents a safety case module. Additionally, we construct *Hazard Mitigation, SW Safety Requirements and SW Integration test* modules (as shown in Figure 7).

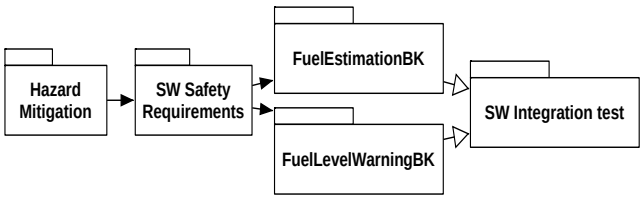

Figure 7. FLES safety case architecture

4.2.3 Construct safety case modules, and

4.2.4 Integrate safety case modules

We merge these two steps for the sake of simplicity. We identify the required DGRs of the *FuelEstimationBK* and *FuelLevelWarningBk* blocks. We also construct the *Hazard Mitigation, SW Safety requirements, FuelEstimationBK, FuelLevelWarningBk,* and *Software Integration test* safety case modules.

Table 2. DGR FuelEstimationBK

Dependencies — Guarantee Relationship| *FuelEstimationBK.G5*				
Guarantee				
Concise Definition		**Definitive Context**	**Incidental Note**	**Traceability**
Provides the *totalFuelLevel* value		The *totalFuelLevel* value is sent on port *SetSensorValue*. *The totalFuelLevel* format is defined by FLES {Interface Specification}.		***F1010TSR1***
Related Dependencies				
N	**Concise Definition**	**Definitive Context**	**Incidental Note**	**Traceability**
1	*FuelLevelSensor* is received via port *GetSetSensorValue.*	*FuelLevelSensor* format is defined by FLES {Interface Specification}		*F3010TSR8*
2	*SetSensorValue* port is available.	The port behaviour is as defined in the FLES {Interface Description}		*F3010TSR9*
3	*FuelEstimation* is correctly configured.	Is executing and has completed configuration		*F4010TSR5*

Table 2 shows one DGR for the software block FuelEstimationBK in which the block (i.e., represented as thread) guarantees that it can provide the estimated fuel level volume in the tank *totalFuelLevel* if the three dependencies are met. Table 3 shows one DGR for the software block *FuelLevelWarningBK* in which the block (i.e., represented as thread) guarantees that it can tell if the fuel is low or not (*lowFuelLevelWarning* is *True* if the fuel is below 7% of the tank capacity and *False* if the fuel is not) once the four related dependencies are met.

In Figure 8, we construct the hazard mitigation argument. Basically, *MitigationHazard1* goal is supported by implementing and assuring the two safety goals that were derived to mitigate it. The safety goals are represented by the two separated away goals *SG1.0ImplAssur*, and *SG2.0ImplAssur*. These goals also represent the integration between *Hazard Mitigation* safety case module and *SW Safety Requirements* (see Figure 9).

In *FuelLevelWarning.BK* Safety case module (see Figure 10), we show how arguing over the dependencies supports the guarantee that is represented by *FuelLevelWarningBK.G1*. The argument module uses *FuelEstimationBK.G5* as a dependency to support the guarantee. *FuelEstimationBK.G5* also relies on a set of dependencies to be guaranteed. Figure 11 shows an argument fragment of the *SW Integration test* safety case module. The objective of the module is to argue over the integration of the software elements within the *Estimator ECU*.

Table 3. DGR FuelLevelWarningBK

Dependencies — Guarantee Relationship | FuelLevelWarningBK.G1				
Guarantee				
Concise Definition		**Definitive Context**	**Incidental Note**	**Traceability**
Provides the *lowFuelLevel Warning* value		The *lowFuelLevelWarning* value is sent on port *setlowFuelLevelWarning* *lowFuelLevelWarning* format is defined by FLES {Interface Specification}.		*F2010TSR1*
Related Dependencies				
N	**Concise Definition**	**Definitive Context**	**Incidental Note**	**Traceability**
1	*totalFuelLevel* is received via port *GetEstimatedFuel LevelValue_2.*	*FuelLevelSensor* format is defined by FLES {Interface Specification}		*F1010TSR1*
2	*set*low*FuelLevelW arning* port is available.	The port behaviour is as defined in the FLES {Interface Description}.		*F3010TSR9*
3	*GetEstimatedFuel LevelValue_2* port is available.			*F4010TSR5*
4	*FuelEstimation* is correctly configured.	Is executing and has completed configuration.		*F4010TSR7*

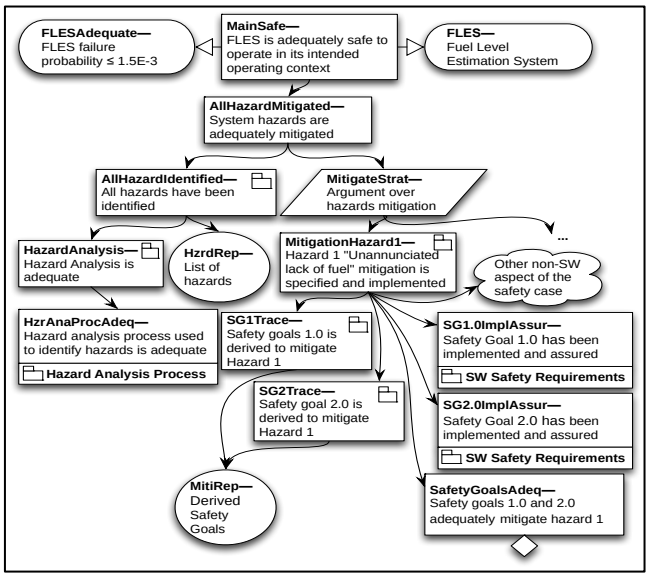

Figure 8. Hazard mitigation safety case module of FLES

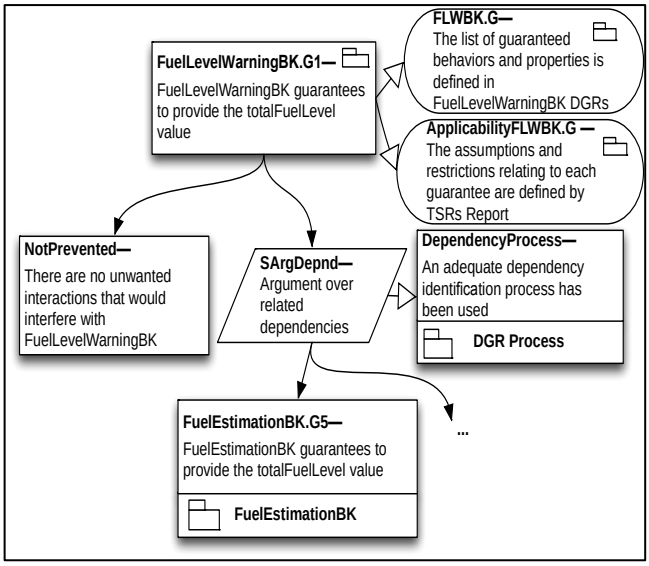

Figure 10. An argument fragment of FuelLevelWarning.BK safety case module

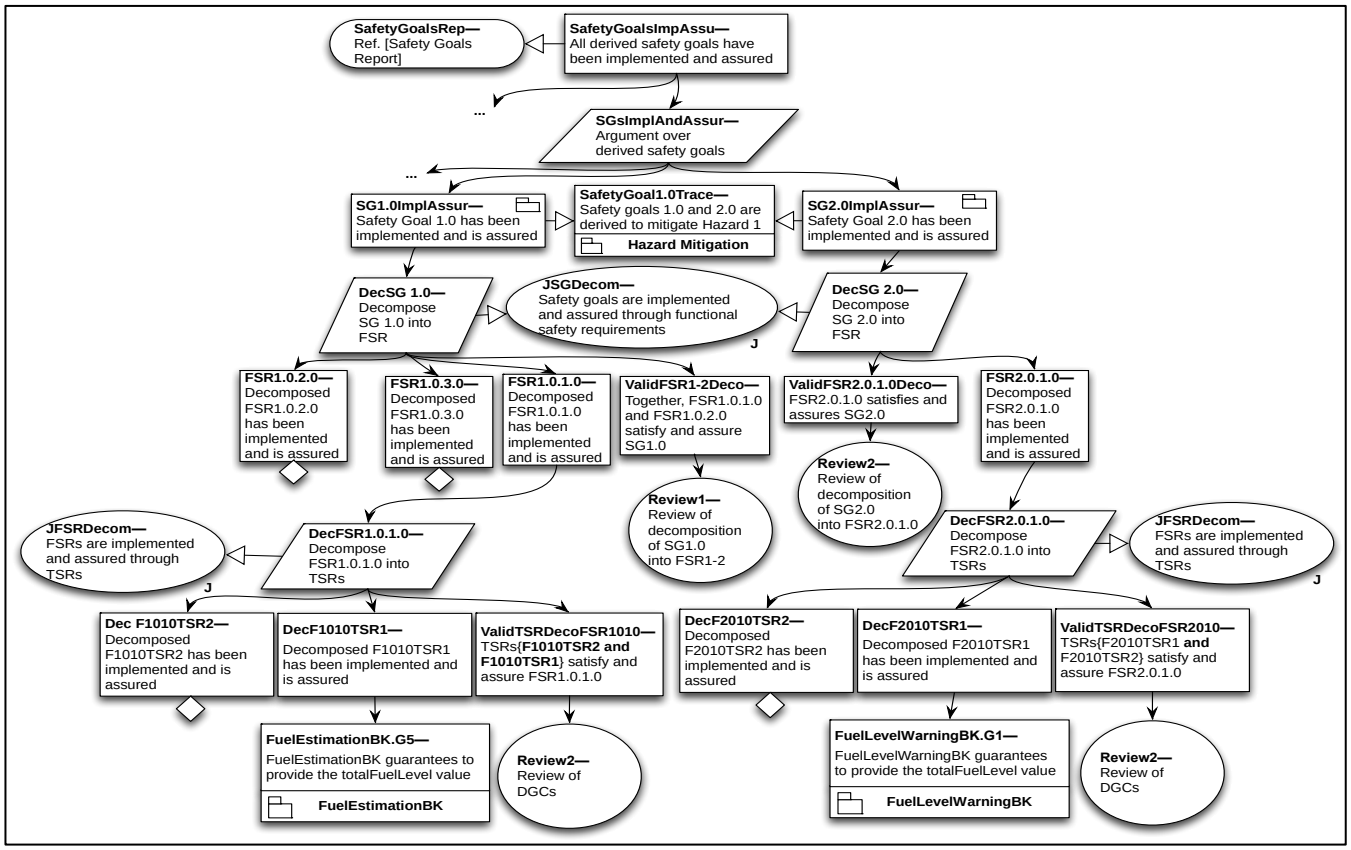

Figure 9. SW Safety Requirements safety case module

The *FuelLevelWarningBK.G1* DGR shows that in order for *FuelLevelWarningBK* being able to fulfil the TSR *F2010TSR1* it requires the TSR *F1010TSR1*, which is guaranteed by a different DGR (i.e., *FuelEstimationBK.G5)*. Here lies the importance of the DGC as it matches such dependencies. Table 4 shows a DGC that matches *F2010TSR1* to *F1010TSR1*. MSSC process requires performing the integration of safety case modules by using a safety case contract module. The latter uses a DGC to set out the matching between the DGRs of the goals involved. However, since our work is more focused on facilitating the impact analysis within the blocks, we do not use safety case contracts in this example thus no goals are supported by contracts. The integration, in our example, is done through public and away goals.

4.2.5 Assess/Improve change impact

In this step, we use our approach for maintaining safety cases (in Section 2.4) to extend IAWG's DGC. We use the extended DGC in the FLES example to show how the extension can help: (1) highlighting the affected argument elements, and (2) identifying inadequacies in the generated artefacts from the development lifecycle of FLES.

Table 4 shows an extended DGC of *FuelLevelWarning.BK* The extension is represented by the cells in grey. Moreover, figure 11 shows items of evidence (i.e., GSN solution) that support claims about the consistency among the ports of FLES blocks. The green elements in the figure represent the annotations described in Section 2.4.

Now, let us consider the potential change scenario in Section 4.2.1 to illustrate how the information contained within the annotations aids the change impact analysis in safety arguments. Merging *FuelEstimation* and *FuelLevelWarning* into one thread will impact the consistency of the interfaces and connections of FLES. Suppose that an engineer making this change had updated the artefact version annotation(s) in part of the argument that refers to the interfaces of those threads. An automated implementation of the described checks in Section 2.4 could highlight the need to re-run the interface consistency check, as well as, the *Estimator* internal interfaces testing. If the new version of the implementation is version 3.3, analysis of the manifest associated with *InConChk* and *TstInnInt* would reveal evidence based on an older version of the implementation and tools could flag *InConChk* and *TstInnInt* as out-of-date and suspect. Automated analysis might also highlight goal *EstimatorImpCorr* because its artefact version annotation refers to an out-of-date version of the Estimator implementation. The goal and its supporting argument are suspect because they might refer to parts of the implementation that no longer exist or make claims about the implementation that are no longer true.

Table 4. FuelLevelWarningDGC

Dependency — Guarantee Contract | FuelLevelWarningDGC			
Consumer Dependency	**Integrator**	**Provider Guarantee**	Artefact Version
FuelLevelWarning BK.G1	Supported by away goal *FuelEstimationBK.G5*	***FuelEstimationBK.G5***	V.3.2
totalFuelLevel value is received	Is Supported By	Provides the *totalFuelLevel* value	
totalFuelLevel value is received via *GetEstimatedFuelLevelValue_2 port*	Is Consistent with	The *totalFuelLevel* value is sent on port *SetSensorValue*.	InConChk TstInnInt
totalFuelLevel data format is defined by FLES {Interface Specification Ref.20}	Is Consistent with	*totalFuelLevel* data format is defined by FLES {Interface Specification Ref.20}	

Supporting Evidence					
No	**GSN element**	**Evidence Version**	**Input Manifest**	**Lifecycle Phase**	**Safety Standard Reference**
1	InConChk	V.3.2	(Inchecker, 1.5), (Code, 1.0)	SW Dev.	§ 8.4.2.2.4 ASIL "C"
2	TstInnInt	V.3.2	(Con1, 3.0), (Code, 3.2)	SW Dev.	§ 8.4.2.2.4 ASIL "C"

Table 5 shows the impacted elements of the safety case with a brief explanation for each element.

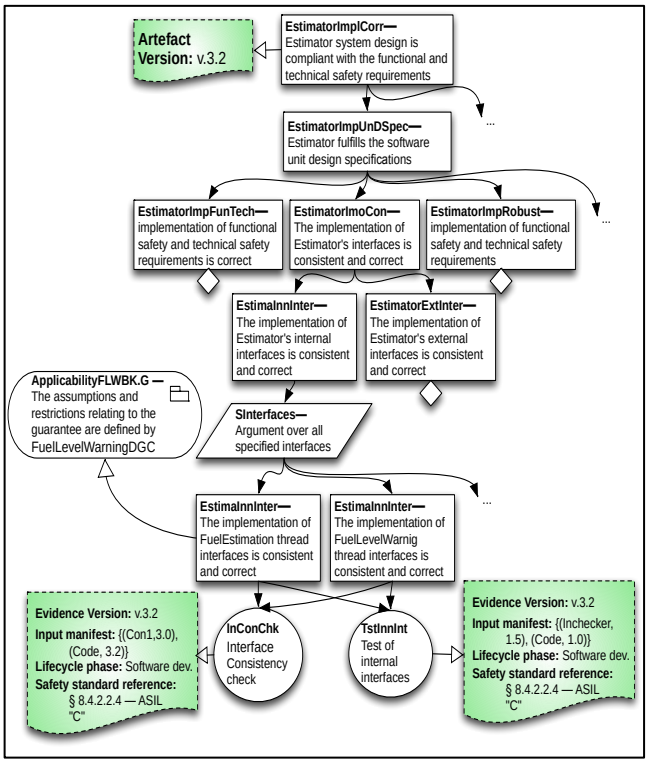

Figure 11. An argument fragment of SW Integration test safety case module

Table 5. Results of change impact analysis

No.	Module Name	Element affected	Explanation
1	*SW Safety Requirements*	DecF1010TSR1	The decomposition of this requirement has been changed
2	*SW Safety Requirements*	DecF2010TSR1	The decomposition of this requirement has been changed
3	*FuelLevelWarning.BK*	The entire module	Merged with another Module
4	*SW Integration test*	EstimaInnInter and all claims below	Argument about the estimation internal interfaces is suspect
5	*SW Integration test*	*InConChk*	Out of date implementation
6	*SW Integration test*	*TstInnInt*	Out of date implementation

The principal difference between our work and the existing approach proposed by the IAWG MSSC is that the MSSC approach contains changes at the level of a safety argument module and the corresponding system blocks. In contrast, our

approach provides the engineer to contain the changes at a lower-level where they feel that a tighter control over change is needed. More specifically, our approach means that changes can be contained within a safety argument module and within specific system blocks. It could be argued that this could have been handled in the existing approach by decomposing the system and its safety argument differently, however in practice it is better not to constrain system architects unnecessarily.

5. CONCLUSION AND FUTURE WORK

Applying changes to systems during their lifetime is inevitable task. In safety critical systems, system changes can be accompanied with changes to safety arguments. Maintaining those arguments is painstaking process because of the dependencies between their elements. The IAWG MSSC process was introduced as a response to safety cases maintenance difficulties. The process recommends applying changes as a series of relatively small increments rather than occasional major ones. However, The guidance of MSSC process does not show detailed information about how to follow some steps including the impact analysis part. In this paper, we applied the process to a real safety critical system to show how system engineers can identify the elements in a safety argument that might be impacted by a change. We showed that by extending the proposed DGC by IAWG to include additional information as annotations that is useful to highlight the impacted argument elements. Moreover, we provided starting points to maintain the affected parts of the argument as we described the reasons why they have become inadequate due to the change. The impact check based on the additional information is still manual as we have not yet studied the feasibility or value of developing a tool to automate the checks but we leave this effort to future work.

6. ACKNOWLEDGMENTS

We acknowledge the Swedish Foundation for Strategic Research (SSF) SYNOPSIS Project for supporting this work.

7. REFERENCES

[1] Jaradat. O, Graydon. P. J and Bate. I. (2014). "An Approach to Maintaining Safety Case Evidence after a System Change*". In Proceedings of the 10th European Dependable Computing Conference.*

[2] EUROCONTROL: *European Organisation for the Safety of Air Navigation, Preliminary Safety Case for Enhanced Traffic Situational Awareness During Flight Operations*, PSC ATSA-AIRB. Available at: www.eurocontrol.int/articles/cascade-documents, accessed: 20February 2015.

[3] Ewan, D. and Whiteside, I. (2012). "*Hierarchical Safety Cases*", Technical Report NASA/TM-2012-216481, NASA Ames Research Center

[4] Kelly, T. (2007*): "Modular Certification".* Lecture Note. Available at: http://webhost.laas.fr/TSF/IFIPWG/Workshops&Meetings/52/workshop/10%20Kelly.pdf, accessed: 20 February 2015.

[5] IAWG MSSC Process (2012). *Modular Software Safety Case Process Description*. Available at: https://www.amsderisc.com/wp-content/uploads/2013/01/MSSC_201_Issue_01_PD_2012_11_17.pdf, accessed: 20 February 2015.

[6] Kelly, T. (2006). "Using software architecture techniques to support the modular certification of safety-critical systems" *Eleventh Australian Workshop on Safety Critical Systems and Software*, Australia.

[7] ISO 26262 (2011). *Road Vehicles — Functional Safety. International Organization for Standardization*.

[8] Origin Consulting (2011). *GSN Community Standard*. Available at: http://www.goalstructuringnotation.info/, accessed 20 February 2015.

[9] Fenn. J. L, Hawkins. R. D, Williams. P, Kelly. T. P., Banner. M. G, and Oakshott. Y. (2007). "The who, where, how, why and when of modular and incremental certification". *In proceedings of the 2nd IET International Conference on System Safety*, pages 135–140.

[10] Kelly. T. (1995) "*Literature survey for work on evolvable safety cases*". Department of Computer Science, University of York.

[11] Wilson S. P, Kelly. T. P., and McDermid. J. A. (1997). "Safety case development: Current practice, future prospects". *In proceedings of Software Bases Systems - 12th Annual CSR Workshop*.

[12] Kelly.T and McDermid. J. (1999). "*A Systematic Approach to Safety Case Maintenance*". In M. Felici and K. Kanoun, editors, Computer Safety, Reliability and Security, volume 1698 of Lecture Notes in Computer Science, pages 13–26. Springer Berlin Heidelberg.

[13] Jaradat.O, Graydon. P, and Bate. I. (2013). "The Role of Architectural Model Checking in Conducting Preliminary Safety Assessment". *In Proceedings of the 31st International System Safety Conference.*

[14] Jaradat.O. (2012). "*Automated Architecture-Based Verification of Safety-Critical Systems*". Master Thesis. Mälardalen University, Sweden. Available at: www.diva-portal.org/smash/record.jsf?pid=diva2%3A723310&dswid=5193, accessed: 20 February 2015.

[15] Conmy, P. (2005). "*Safety Analysis of Computer Resource Management Software*". PhD Thesis. University of York. Available at: https://www.cs.york.ac.uk/ftpdir/reports/2006/YCST/07/YCST-2006-07.pdf, accessed: 5 March 2015.