Article:

Wei, Ran, Foster, Simon David orcid.org/0000-0002-9889-9514, Mei, Haitao et al. (7 more authors) (2024) ACCESS:Assurance Case Centric Engineering of Safety-critical Systems. Journal of Systems and Software. 112034. ISSN 0164-1212

https://doi.org/10.1016/j.jss.2024.112034

The Journal of Systems and Software 213 (2024) 112034

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

ACCESS: Assurance Case Centric Engineering of Safety–critical Systems[☆]

Ran Wei [a,b], Simon Foster [c,*], Haitao Mei [c], Fang Yan [c], Ruizhe Yang [d], Ibrahim Habli [c], Colin O'Halloran [e], Nick Tudor [e], Tim Kelly [c], Yakoub Nemouchi [c]

[a] Department of Engineering, University of Cambridge, Cambridge, CB3 0FA, UK
[b] School of Computing and Communications, University of Lancaster, Lancaster, LA1 4WA, UK
[c] Department of Computer Science, University of York, York, YO10 5GH, UK
[d] School of Computer Science and Technology, Dalian University of Technology, Dalian, 116024, China
[e] D-RisQ Ltd., Malvern, WR14 3SZ, UK

ARTICLE INFO

Keywords:
Safety–critical systems engineering
Assurance case development
Model based systems engineering
Formal methods
Robotics and autonomous systems

ABSTRACT

Assurance cases are used to communicate and assess confidence in critical system properties such as safety and security. Historically, assurance cases have been manually created documents, which are evaluated by system stakeholders through lengthy and complicated processes. In recent years, model-based system assurance approaches have gained popularity to improve the efficiency and quality of system assurance activities. This becomes increasingly important, as systems becomes more complex, it is a challenge to manage their development life-cycles, including coordination of development, verification and validation activities, and change impact analysis in inter-connected system assurance artifacts. Moreover, there is a need for assurance cases that support evolution during the operational life of the system, to enable continuous assurance in the face of an uncertain environment, as Robotics and Autonomous Systems (RAS) are adopted into society. In this paper, we contribute ACCESS — Assurance Case Centric Engineering of Safety–critical Systems, an engineering methodology, together with its tool support, for the development of safety–critical systems around evolving model-based assurance cases. We show how model-based system assurance cases can trace to heterogeneous engineering artifacts (e.g. system architectural models, system safety analysis, system behaviour models, etc.), and how formal methods can be integrated during the development process. We demonstrate how assurance cases can be automatically evaluated both at development and runtime. We apply our approach to a case study based on an Autonomous Underwater Vehicle (AUV).

1. Introduction

Safety–critical systems require justifications that they are acceptably safe to operate in their defined operational contexts. *Assurance cases* provide an explicit means for arguing, justifying and assessing the confidence in the safety of safety–critical systems. The submission of an assurance case is increasingly being required during system certification in many safety–critical industries, such as aviation (European Organisation for the Safety of Air Navigation (EUROCONTROL), 2006), nuclear power (International Atomic Energy Agency (IAEA), 2008), transportation (International Organization for Standardization (ISO), 2011; U.K. Rail Safety Standards Board, 2007), healthcare (Habli et al., 2018) and defence (U.K. Ministry of Defence (MOD), 2007). Prior to certification, an assurance case must be rigorously, and often independently, *evaluated*[1] to ensure that the arguments and evidence for safety is coherent and convincing.

Assurance cases are not self-contained documents. They usually depend on a variety of *engineering artifacts* that provide contextual and evidential information, including requirement documents, architecture designs, behaviour models, safety analyses, etc. These artifacts may originate from diverse languages and tools, and can be used for prototyping, analysis, formal verification, and the derivation of real-world artifacts. Therefore, assurance case *evaluation* involves the evaluation of the engineering artifacts an assurance case depends on, which is often an informal, manual, and error-prone process. In an *idealised*

[☆] Editor: Raffaela Mirandola.

[*] Corresponding author.

E-mail addresses: rw741@cam.ac.uk, r.wei5@lancaster.ac.uk (R. Wei), simon.foster@york.ac.uk (S. Foster), haitao.mei@icloud.com, haitaom@acm.org (H. Mei), fang.yan@york.ac.uk (F. Yan), ruizheyang@mail.dlut.edu.cn (R. Yang), Ibrahim.habli@york.ac.uk (I. Habli), coh@drsq.com (C. O'Halloran), njt@drsq.com (N. Tudor), tim.kelly@york.ac.uk (T. Kelly), y.nemouchi@gmail.com (Y. Nemouchi).

[1] In this work, we use *evaluation* to refer to both validation and verification activities involved in the development and the assessment processes of safety–critical systems and their assurance cases.

https://doi.org/10.1016/j.jss.2024.112034
Received 31 July 2022; Received in revised form 16 May 2023; Accepted 19 March 2024

development process, an assurance case is the central point of reference for all system stakeholders, to allow effective communication over diverse engineering artifacts. In addition, changes in engineering artifacts require the assurance case to be re-evaluated (Denney et al., 2015), which can significantly impact development efficiency. This challenge becomes more obvious as systems become more complex. Hence, there is a need to automate some (if not all) of the system assurance activities to efficiently manage assurance cases and their referenced engineering artifacts.

Over the past few years, system assurance practitioners have begun adopting Model Based Systems Engineering (MBSE). MBSE promises the interoperability and management of diverse artifacts/models in an automated manner, which provide the basis for automated, coherent and self-contained assurance cases. However, existing assurance case notations, such as the Goal Structuring Notation (GSN) (Kelly and Weaver, 2004) and Claim-Argument-Evidence (CAE) (Bishop and Bloomfield, 2000), do not have a sufficient model-based foundation to systematically support this kind of integration (Wei et al., 2019). Consequently, existing model-based assurance case approaches cannot provide the collective and automated evaluation of an assurance case, together with the engineering artifacts that it may depend on. The inspection, evaluation, and change management of engineering artifacts still remain manual.

In recent years, new applications for Robotics and Autonomous Systems (RAS) have emerged, which are often safety–critical. RASs are increasingly open (they inter-connect at runtime) and adaptive (they adapt to changing contexts at runtime), that render the current generation of safety assurance approaches insufficient (Trapp et al., 2013; Denney et al., 2015). Specifically, assurance cases for RAS need to be *living* documents that can evolve during the operational life of the system with minimal human intervention. As such, it is imperative to shift some system safety assurance activities from development time to runtime (Trapp et al., 2013; Wei et al., 2018). This is a significant challenge, though, as it requires automation of verification and validation activities that are both crucial parts of the evaluation process. On the one hand, we must demonstrate that each engineering artifact meets its requirements through verification, and on the other we must ensure that real-world system artifacts exhibit the behaviour predicted by a model through techniques like runtime safety monitoring (Machin et al., 2018).

To address the identified challenges, we introduce our model-based, assurance oriented methodology — Assurance Case Centric Engineering of Safety–critical Systems (ACCESS). ACCESS is underpinned by a combination of (1) design-time automated assurance-case-and-engineering-artifacts management and evaluation, and (2) runtime assurance case evaluation based on runtime data. We present a tooling prototype, Assurance Case Management Environment (ACME), which supports the creation and the management of assurance cases based on the Structured Assurance Case Metamodel (SACM) (Object Management Group, 2020), an international standard.

To demonstrate our approach, we provide a case study on the assurance case for an Autonomous Underwater Vehicle (AUV), including its safety requirements, arguments, and a formal model of the safety controller in the RoboChart language (Miyazawa et al., 2019) with formal verification evidence.

We discuss how we can apply our approach to develop critical systems around an evolving assurance case. In addition, we also demonstrate tool support for ACCESS at development time, so that we can: (1) perform a collective evaluation of an assurance case, via model-based traceability, with respect to the engineering artifacts it refers to; (2) automatically invoke evidence from formal methods by using Isabelle/HOL (Nipkow et al., 2002) as a verification service within an assurance case; (3) automatically generate and machine-check formalisation of an assurance case to verify its logical integrity; and (4) enable automated change impact analysis from engineering artifacts to assurance cases. We also discuss how we can turn a development time assurance case to a dynamic runtime assurance case and discuss how runtime assurance case evaluation (based on runtime data) can be achieved using our approach.

The main contributions of our paper are:

1. ACCESS — a critical systems engineering methodology around an evolving assurance case model;
2. Automated means to evaluate an assurance case with its referenced engineering artifacts at development time and runtime (with a prototypical dynamic assurance case management system to evaluate assurance cases based on runtime data);
3. Facilities to integrate diverse formal verification results into an assurance case and automatic generation of a formalised assurance case in Isabelle/SACM for analysis using theorem proving;
4. Automated change impact analysis from engineering artifacts to assurance cases;
5. The application of all of above to an AUV case study.

The rest of the paper is organised as follows. In Section 2 we provide some background information on assurance cases, GSN, model-based assurance case and formal methods. In Section 3 we describe the generic ACCESS methodology. In Section 4 we discuss our tool support to back the ACCESS methodology In Section 5 we evaluate the ACCESS methodology in depth with an case study for an Autonomous Underwater Vehicle (AUV). In Section 7 we discuss related work and in Section 8 we conclude this paper with a discussion and point out future research directions.

2. Preliminaries

This section provides background information on assurance cases, assurance case notations, model-based system assurance and formal methods, including the terminologies and concepts used in the rest of the paper.

2.1. Assurance cases

The practice of safety certification is increasingly *goal-oriented* rather than highly prescriptive (Denney et al., 2015). Goal-oriented certification places greater emphasis on explicitly stating safety claims, and supplying an argument along with supporting evidence to satisfy certification goals that regulators define (McDermid, 2001). Examples may include “All identified hazards have been mitigated” or “Necessary assumptions of the physical environment have been defined”. Such arguments and evidence are generally organised in the form of an *assurance case*. As defined in Kelly (1999), an assurance case is a document that *should communicate a clear, comprehensive and defensible argument that a system is acceptably safe to operate in a particular context*.

Conventionally, an assurance case is not a self-contained document. Definitions of assurance case (U.K. Ministry of Defence (MOD), 1996, 1997) indicate that an assurance case is a document (either as a logical concept or as a physical artifact) that can refer to, and pull together information regarding system safety (such as system requirements, system architectural design, safety analyses, etc.), to form a safety argument. The development of an assurance case involves communications among various stakeholders, one typical scenario is illustrated in Fig. 1. When a system concept is formed, system engineers define a set of requirements. Based on these requirements, safety engineers may perform safety analyses, from which hazards (and their associated risks) are identified. Identified hazards and risks are then used to derive safety goals, from which safety engineers can elicit safety requirements. Safety requirements are then considered in the system design, which impose constraints (e.g. acceptable failure rates), and mitigation measures (e.g. redundancy and monitoring). As an assurance case is developed, it may refer to all engineering artifacts above within its argument, for contextual and evidential information. Therefore, the *evaluation* of

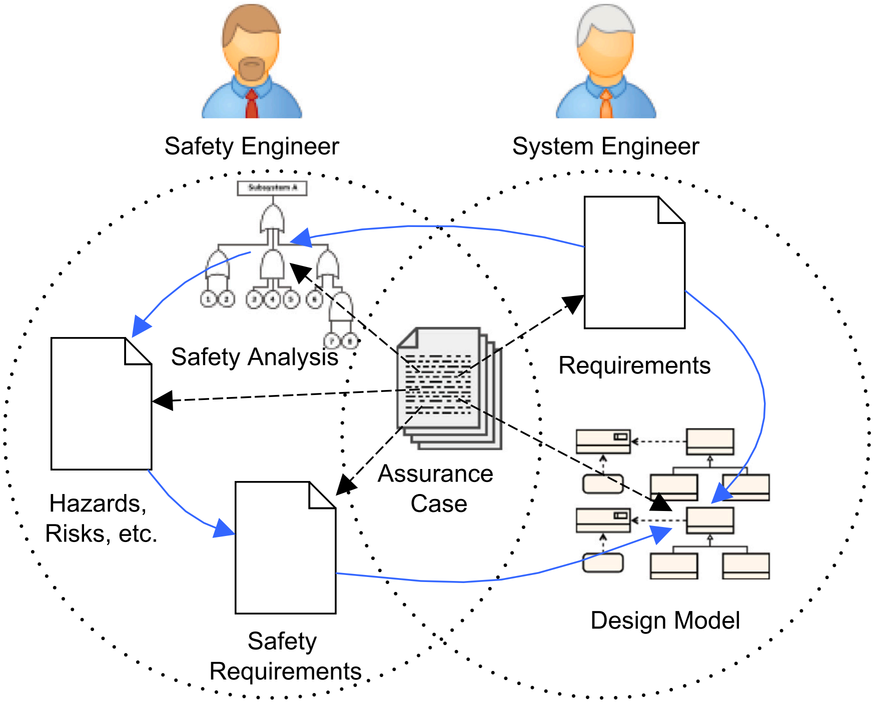


Fig. 1. Assurance cases and engineering artifacts.

an assurance case typically involves the validation and verification of engineering artifacts it refers to.

Assurance cases are subject to evolution, where engineering artifacts and arguments need to be adapted, potentially because of new requirements or upgrades. Changes in the engineering artifacts may invalidate the assurance case, and so its integrity must be checked through evaluation. To evaluate an assurance case, practitioners typically need to trace, navigate to, review, validate and verify the engineering artifacts it depends on Hawkins et al. (2015). This then informs the decision as to whether the system is acceptably safe to deploy and operate in its intended operational context. On the other hand, when an engineering artifact is changed during the development process, its impact in the assurance case needs to be identified, sometimes resulting the assurance case to be re-evaluated. This is observed in Nair et al. (2015), the authors of which report that evidence completeness and change impact for assurance cases are managed mostly manually using (sometimes even no) traceability information. Importantly, their study raises the question of how evolution and changes are identified, assessed and managed at the level of assurance case.

2.2. Goal structuring notation

In the current state of practice, assurance cases are typically communicated using graphical notations, among which the most widely used notation is the Goal Structuring Notation (GSN) (Kelly and Weaver, 2004). GSN is a well established graphical argumentation notation that is widely adopted within safety–critical industries for the presentation of safety arguments within safety cases. The core elements of GSN are shown in Fig. 2.

A *Goal* represents a safety claim within the argumentation. A *Strategy* is used to describe the nature of the inference that exists between a goal and its supporting goal(s). A *Solution* represents a reference to an evidence item or multiple evidence items. A *Context* represents a contextual artifact, which can be a statement, or a reference to contextual information. An *Assumption* represents an assumed statement made within the argumentation. A *Justification* represents a statement of rationale. An element can be *Undeveloped*, which means that a line of argument has not been developed yet (meaning it being abstract and needs to be instantiated). The *Undeveloped* notation can apply to *Goals* and *Strategies*. The *Undeveloped Goal* in Fig. 2 is an example.

Core elements of GSN are connected with two types of connectors, as shown in Fig. 3. The *SupportedBy* connector allows inferential or evidential relationships to be documented. The *InContextOf* relates contextual elements (i.e. *Context*, *Assumption* and *Justification*) to *Goals* and *Strategies*.

When elements of GSN are linked together in a network, they are often referred to as a *goal structure*. The purpose of a goal structure is to show how *Goals* are successively broken down into sub-*Goals* until a point is reached where *Goals* can be supported by direct reference to available evidence (*Solutions*). An example of a goal structure is shown in Fig. 14.

Goal structures can be organised in *Modules*. For example, for a system that consists of two components A and B, it is possible to organise the safety cases of component A and B in two Modules *MA* and *MB*. Modularity promotes re-use, so that safety cases for system components can be re-used when different components are integrated to form a system. Fig. 4 shows the GSN elements that enable modularity support. When integrating system safety cases, a *Contract Module* can be used to **bind** different *Modules* together.

Binding is done via the use of *Away Goals*, *Away Context*s and *Away Solution*, where *Goals*, *Context*s and *Solutions* from an external *Module* can be referenced. Like other GSN elements, *away* elements can be connected using *SupportedBy* and *InContextOf* connectors.

2.3. Model-based assurance cases

Model Based Systems Engineering (MBSE) (Brambilla et al., 2017) is a contemporary systems engineering approach. In MBSE, *models* are first class artifacts, therefore *driving* the development. MBSE has been proven to improve consistency and productivity significantly due to the automation provided by model management operations (Jaaksi, 2002; Kärnä et al., 2009).

Over the past few years, model-based assurance case approaches emerged due to the benefits introduced by MBSE. Studies have shown

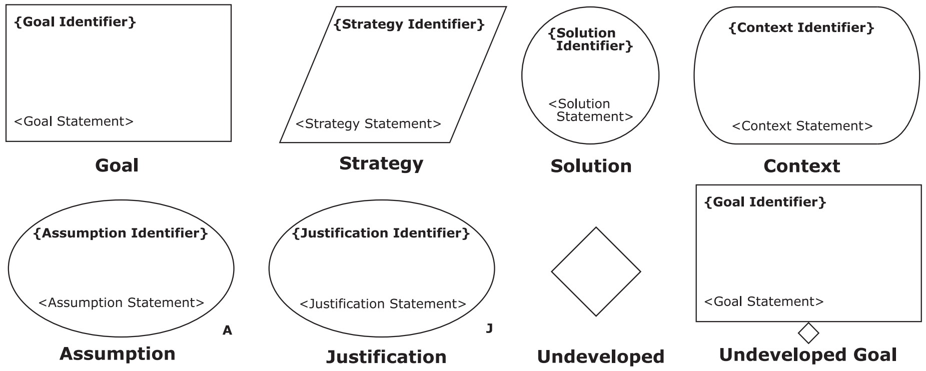


Fig. 2. Core GSN elements.

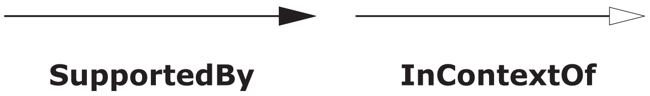


Fig. 3. GSN connectors.

how automated MBSE operations can be performed on model-based assurance cases (created using GSN) to check the well-formedness of assurance cases (Denney and Pai, 2017), generate and assemble structured argumentation within assurance cases (Hawkins et al., 2015), and automatically generate texts for assurance case reports (Denney and Pai, 2017). However, existing model-based assurance case approaches (GSN and CAE - Cliams-Arguments-Evidence (Bishop and Bloomfield, 2000)) do not provide sufficient support for traceability to engineering artifacts. This is partly caused by the fact that GSN and CAE permit only structured arguments and not external artifact traceability. This is a historical problem, as prior to model-based assurance case approaches, GSN and CAE are used to create physical documents, which naturally contain references to other (physical) engineering artifacts by their names.

2.4. Structured assurance case metamodel

Whilst graphical assurance case notations are powerful in expressing arguments regarding the safety of systems, they have their limitations. As discussed previously, an assurance case is not a self-contained document. That is, GSN elements (such as *Contexts* and *Solutions*) may refer to engineering artifacts that provide contextual and evidential information. Existing graphical notations (e.g. GSN and CAE) do not support such traceability.

To address this limitation, the Object Management Group (OMG) specified and issued the *Structured Assurance Case Metamodel* (SACM) (Object Management Group, 2020). SACM is developed by the specifiers of existing system assurance approaches (e.g. GSN and CAE), based on the collective knowledge and experiences of safety and/or security practitioners over the period of last two decades. Therefore, features that are not previously supported by GSN and CAE have been evaluated and included in SACM.

SACM organises model elements in *Packages* to promote modularity, as shown in Fig. 5. An *AssuranceCasePackage* may contain a number of *TerminologyPackages* (to store terms and expressions used in the assurance case), *ArtifactPackages* (to store artifacts, resources, events, etc. throughout the assurance case development process) and most importantly *ArgumentPackages* (to store safety/security arguments of a system or a component).

SACM provides essential concepts for complete model-based assurance cases (although currently there has not been approaches and tools to achieve it) in its *Base* component shown in Fig. 6. For a *ModelElement* in SACM, it can have a number of *UtilityElements*, in this work, we particularly focus on the *ImplementationConstraint* concept, using which we describe the validation rules against engineering models. In addition, it can also be seen that a *ModelElement* can "cite" another *SACMElement* via its *CitedElement* association. This is a powerful mechanism, as it allows the users of SACM to cite any *ModelElement* contained within one model.

Another SACM component worth mentioning is the *Artifact* component, as shown in Fig. 7. In this work, we make use of the *ArtifactAssets* (specifically, the *Artifact* class) to demonstrate how we could record information (such as location, format and meta information) of external engineering artifacts and then use such information to perform automated verification and validation of such artifacts.

2.5. Formal methods and RoboChart

Assurance cases can benefit from formal methods (Gleirscher et al., 2019b). Informal safety arguments and evidence can be difficult to automatically evaluate, and may be subject to some argumentation fallacies (Greenwell et al., 2006). Thus, formalisation of requirements, to allow the use of formal methods, can significantly improve both automation and the confidence (Foster et al., 2020). At the same time, assurance cases allow us to put formal methods results in context. For systems assurance it is never enough to simply prove properties of a formal model. For the results to be meaningful, the model must be linked to its corresponding real-world artifact (Lee and Sirjani, 2018), such as software and hardware by some form of validation argument. Consequently, a comprehensive demonstration of safety requires both assurance cases and formal methods.

An important development here is Isabelle (Nipkow et al., 2002), a verification framework for integrated formal methods (Wenzel and Wolff, 2007; Wenzel, 2019; Foster et al., 2019, 2021). Development centres around documents called Isabelle *theories*, which encode graphs of hyperlinked mathematical artifacts, such as definitions, theorems, and proofs. Formal method integration is supported by (1) a flexible front-end, which supports a variety of languages (Tuong and Wolff, 2019) and their translation into formal semantics; (2) an extensible plugin-oriented architecture where external tools, such as SMT solvers (Blanchette et al., 2011), can improve automation; and (3) incremental theory processing (Wenzel and Wolff, 2007). Moreover, Isabelle can be installed as a server component which other tools can make use of as a verification tool service. At the foundational level, Isabelle can be used for mechanisation of a variety of formal semantics (Nipkow and Klein, 2014), to support verification tools. A front-end for a programming language, such as C (Tuong and Wolff, 2019), can be developed to support concrete programme verification (Alkassar et al., 2008). Moreover, SACM was recently implemented in Isabelle to create

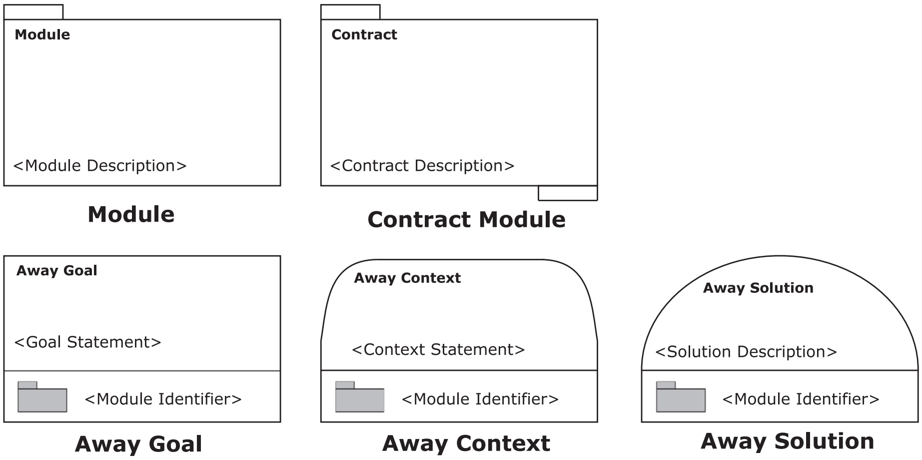


Fig. 4. Modular GSN elements.

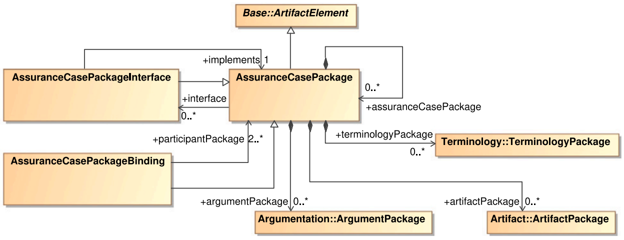


Fig. 5. The assurance case component of SACM (Object Management Group, 2020).

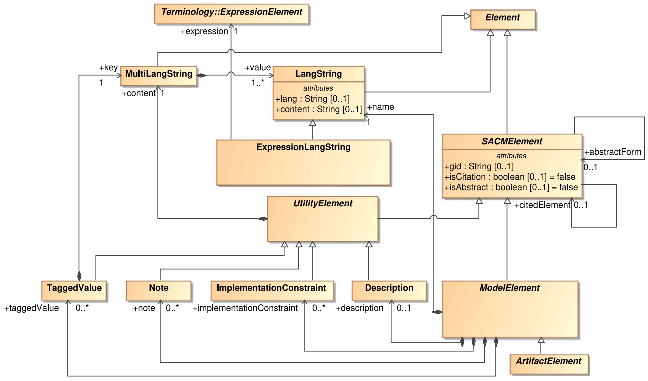


Fig. 6. The base component of SACM (Object Management Group, 2020).

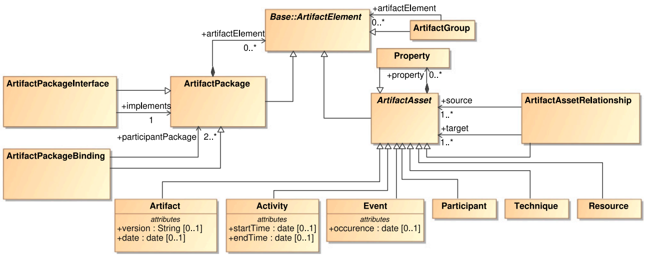


Fig. 7. The artifact component of SACM (Object Management Group, 2020).

Isabelle/SACM (Foster et al., 2019, 2021), which allows verification of assurance cases, integration with formal evidence, and generation of certification documents (Brucker and Wolff, 2019).

Formal methods can be difficult for non-experts to apply, and so there is a desire to use model-based graphical frontends. For example, the RoboChart language (Miyazawa et al., 2019) is a graphical language for the architectural and behavioural description of a robotic controller. It includes a formalised subset of the UML state machine notation with a complete formal semantics in the CSP process algebra (Brookes et al., 1984). The formal semantics allows RoboChart models to be subjected to formal analysis using model checking (Miyazawa et al., 2019) and theorem proving (Foster et al., 2018, 2020). RoboChart is therefore both accessible to practitioners, and at the same time uses formal methods to allow development at higher assurance levels.

A RoboChart model consists several elements:

1. a *data model*, consisting of data types and functions with pre- and postconditions;
2. *interfaces*, which collect together variables, events, operations, and clocks that can be used by other components;
3. *robotic platforms*, which abstract the hardware by providing variables and events, potentially through provided interfaces;
4. *controllers*, which describe different units that control the robot and communicate with other controllers and the robotic platform using required interfaces;
5. *state machines*, which are used to describe the behaviour of controllers and can communicate using events and shared variables;
6. an *architectural model*, which describes the connections between controllers and robotic platforms.

States and transitions in a RoboChart state machine can specify actions using a formal action language inspired by CSP. It includes primitives for receiving events $c?v$, sending them $d!e$, and assigning values to variables $x := e$. RoboChart also has a discrete time model, and each controller can share a number of clocks that can be used to observe the passage of time.

In this paper, we use RoboChart for modelling an Autonomous Underwater Vehicle (AUV). To support such languages, particularly where there is a diversity of such notations, an integrated approach is required. Model-based assurance cases allow such an integration of heterogeneous models, and support justification and traceability for these models, and any associated analysis results.

3. Approach overview

In this section, we propose the methodology of *Assurance Case Centric Engineering of Safety–critical Systems* –ACCESS, which is an engineering methodology for developing and assuring a system (including

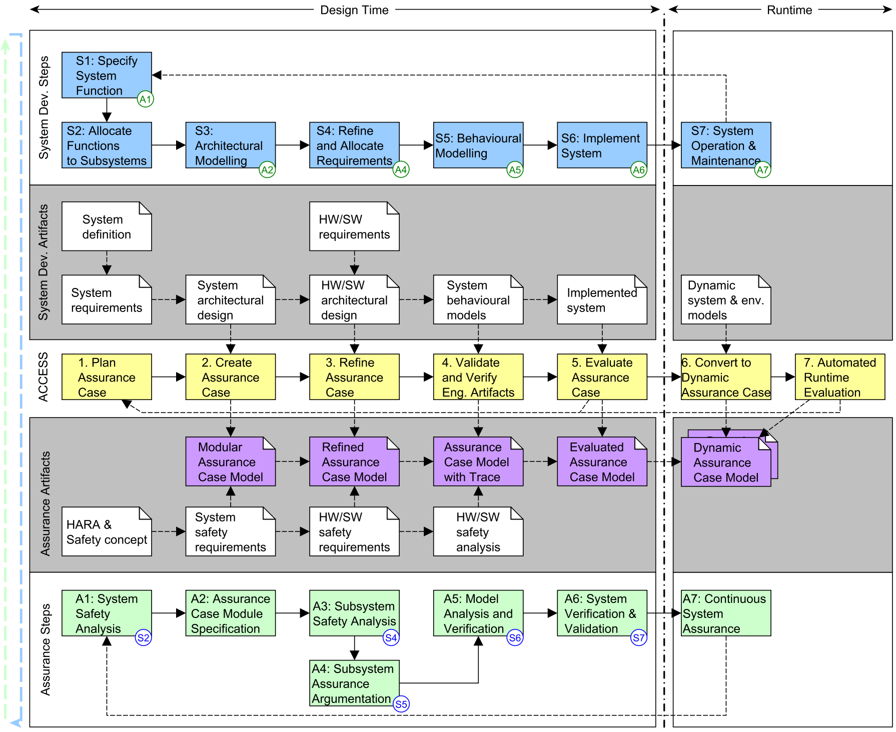


Fig. 8. ACCESS process overview. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

both its hardware and software) around an evolving Assurance Case, adopting principles of *Model Based Systems Engineering* (MBSE). We use the term "model" in a broad sense to encompass any structured machine-readable artifacts, which include resources like EMF-based models, XML files, spreadsheets, databases as well as models created using other technologies (e.g. Simulink).

The ACCESS process is illustrated in Fig. 8. We consider activities in both the System Development Process (boxes rendered in blue in the upper swim lane with white background) and the System Assurance Process (boxes rendered in green in the lower swim lane with white background). For each kind of the processes, we identify a key set of engineering artifacts (swim lanes with grey background), and we discuss the relationships between the engineering artifacts and the assurance case of the system.

There are 7 steps in the ACCESS methodology (rendered in yellow), each step coordinates System Development activities and System Assurance activities. The columns in Fig. 8 indicates the scopes of ACCESS steps. Within each ACCESS step, activities can be iterative (this is indicated by the circular dashed arrow lines on the left side of the swim lanes), that is, System Development and System Assurance activities can be repeated until each ACCESS step is deemed sufficiently executed. Activities in both groups are interleaved, in Fig. 8, there are circles at the bottom right corner of some of the activities, indicating that practitioners are advised to continue the development by conducting the activity identified in the circles (e.g. S1 to A1).

3.1. Step 1: Plan assurance case

In this step, the first task to perform is **S1: Specify System Function**, in which the high-level system functional requirements are defined, the hardware platform is chosen, and assumptions of the environment are specified. The function specification provides the top-level contract for the system: provided it is deployed in an environment satisfying the assumptions, it will perform the required functions. Once this task is complete, the assurance process begins in task (**A1: System Safety Analysis**), which includes activities such as Hazard Analysis and Risk Assessment (HARA), using analysis approaches such as Failure Mode and Effects Analysis, Fault Tree Analysis, etc. From the analysis, a preliminary list of *Safety Goals* can be derived, forming the *Safety Concept* of the system. Based on the safety concept, system development task **S2: Allocate Functions to Subsystems** shall be performed, defining subsystems, and the interface between them.

Outcome of this step may include: system definition, system requirements, HARA and the Safety concept.

3.2. Step 2: Create assurance case

In task **S3: Architectural Modelling** the architecture of the system is modelled, including subsystem blocks, functionalities provided by the hardware platform, and connections between the various components. Based on this architecture model, in task **A2: Assurance Case Module Specification**, corresponding assurance case modules shall be specified. This includes generation of a public claim for each requirement that has been allocated to a particular subsystem, each of which needs an associated argument, and also public assumptions that will need to be satisfied by peer subsystems.

Outcome of this step may include: system architectural design, system safety requirements and a modular assurance case model.

3.3. Step 3: Refine assurance case

The draft assurance case created in ACCESS step 2 is further refined. In this step, the first task to perform is **A3: Subsystem Safety Analysis**, in which safety analysis is performed for every subsystem identified in the system architectural design. Next, in task **S4: Refine and Allocate Requirements**, system requirements (as well as safety requirements) are allocated to subsystems, this requirement allocation shall be preferably traceable. Then, task **A4: Subsystem Assurance Argumentation** is performed, in which safety arguments are developed for each of the subsystem, it is to be noted that the argumentation shall correspond to the system requirements and safety requirements, and traceability shall be maintained. Tasks in this step shall be performed iteratively until the refined assurance case model is deemed sufficiently mature.

Outcome of this step may include: hardware/software requirements, architectural design, safety requirements and the refined assurance case model.

3.4. Step 4: Validate and verify engineering artifacts

With the assurance case model in place, the next step is to investigate how it could be verified and validated, and preferably in an automated manner. In our process, we introduce an additional task, **S5: Behavioural Modelling**, which is typical for software development process. This task can include the creation of state machines or sequence diagrams for each of the subsystem, conforming to the architectural model. We assume that the behavioural modelling notation will have a formal semantics, as is the case for example with the RoboChart language (Miyazawa et al., 2019). Next, we propose the task of **A5: Model Analysis and Verification**. The purpose here is to ensure that each of the requirements is represented in or satisfied by the behavioural model, and reflected in the assurance case. For example, we may wish to state that a particular technical requirement is implemented by a transition, or that a state machine satisfies a global safety requirement. We may also wish to verify the integrity of the model, for example by checking for the possibility of deadlock or livelock using a model checker. This task may expose flaws in the behavioural model, such as inadequately implemented requirements, and so there is iteration between **S5** and **A5**, which should in all circumstances be carried out by separate teams.

In this step it is possible to establish traceable links from the assurance case to its supporting models to form a self-contained assurance case model repository, this is where benefits of MBSE emerge — that some (if not all) of the verification and validation tasks can be automated from the assurance case as an entry point.

Outcome of this step may include: System behavioural model, hardware/software safety analysis and an assurance case model with trace.

3.5. Step 5: Evaluate assurance case

Once a robust behavioural model is created, where every requirement can be traced to a model element, or verification property, the design can be synthesised into an implementation by task **S6: Implement System**. This task can invoke a variety of techniques, including formal code and data refinement, code generation, and manual coding. In parallel with implementation, the assurance process executes task **A6: System Verification and Validation**, all hardware and software components are verified based on the requirements (for both system and safety) using techniques such as model-based testing and code verification, validation activities are also performed to support the validity of the assurance case. Again, these two tasks are iterative as verification may expose implementation or design issues that need addressing.

Unsuccessful verification results are reflected in the assurance case (presumably in an automated fashion), and shall be fixed by repeating previous ACCESS steps where necessary. The assurance case is “complete” once no unsupported claims remain. The assurance case will also need to be independently evaluated, though this is not shown in the process as evaluation is cross-cutting in all activities. Presumably, assurance case evaluation is automated provided that an MBSE approach is adopted and the traceablility links from the assurance case to its supporting engineering artifacts are established.

Outcome of this step may include: implemented system, system verification reports, system validation reports.

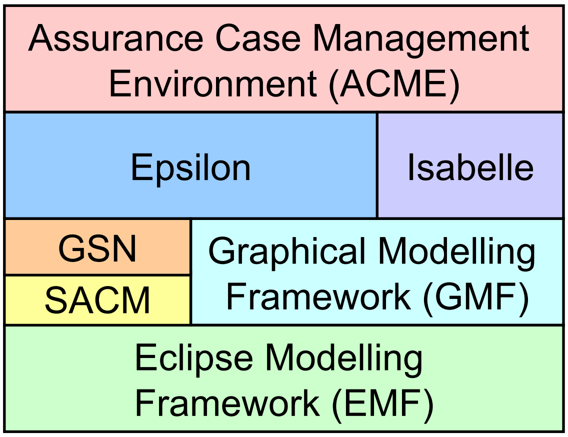


Fig. 9. Assurance Case Management Environment (ACME).

3.6. Step 6: Convert to dynamic assurance case

For RASs, it is becoming imperative to argue the safety at runtime. Therefore, this (optional) step targets system operation and maintenance. In task **S7: System Operation and Maintenance**, requirements for the safe operation and maintenance of the system are specified, including safety related requirements (e.g. safe operation protocols, safe maintenance procedures). In task **A7: Continuous System Assurance**, practitioners shall determine, based on system operation and maintenance context, which part of the assurance case shall be converted to dynamic (i.e. that runtime data are reflected to engineering models at runtime) one, so that assurance can still be carried out at system runtime.

Outcome of this step includes (but not limited to): Dynamic system models, system environment models and dynamic assurance case model.

3.7. Step 7: Automated runtime evaluation

Once the dynamic assurance case is obtained, in this step is to perform automated runtime evaluation, to automatically evaluate the parts of the assurance (and its supporting engineering artifacts) to constantly check the validity of the assurance case. It is to be noted that the assurance case evaluation shall be non-invasive, that it only provides the system with the validity of the assurance case, and shall not take control of the system in any form.

The evaluation results can be used by the system at runtime to yield system safety status at runtime, and take measures to get back to the safe state. Evaluation results shall also be recorded and reviewed for continuous improvement of the system, and ACCESS steps can be repeated for this purpose.

4. Tool support

The ACCESS methodology is backed by our tool support - *Assurance Case Management Environment* (ACME). ACME is an integrated model based assurance case modelling and management framework, that supports the creation and management of assurance case models that conform to the Structured Assurance Case Metamodel (SACM) (Object Management Group, 2020). Since SACM is relatively new and its graphical notations are being standardised, ACME also supports the Goal Structuring Notation (GSN), whose current model-based implementation extends the abstract syntax of SACM, explained in detail in Wei et al. (2019). In this way, ACME supports the creation of model-based GSN diagrams, and at the same time provides access to all other features of SACM.

The architecture of ACME is illustrated in Fig. 9. We implement SACM and GSN with the Eclipse Modelling Framework (EMF) (Steinberg et al., 2008), and use Graphical Modelling Framework (GMF) (Eclipse Foundation, 2003) to create graphical editors for SACM packages and GSN modules.s To enable automated model management and the checking of formal notations, we also integrate:

- The Eclipse Epsilon platform (Kolovos et al., 2008), which is an integrated model management platform, that provides task specific model management languages (model validation, model transformations, etc.) that operate on models defined in different modelling technologies (EMF, Excel spreadsheet, Simulink (Mathworks, 2020), UML, etc.);
- Isabelle (Nipkow et al., 2002), which is a generic proof assistant that allows mathematical formulas to be expressed in a formal language and provides tools for proving these formulas in a logical calculus, with a high degree of automation. Isabelle also provides an extensible document model which can support the encoding of different meta-models and associated parsers.

Using SACM's full potential and with the help of model management frameworks, ACME currently supports (1) fine-grained traceability from an assurance case to its referenced engineering artifacts (defined in mainstream modelling technologies) to the level of model element(s); (2) traceability to formal notations in Isabelle; and (3) automated means to validate/verify traced engineering artifacts. It is to be noted that ACME also support other high-level functionalities, but they are not within the scope of this paper.

To enable the traceability from an assurance case to its supporting engineering artifacts, we make use of SACM's *Artifact* component (discussed in Section 2.3) in ACME. With the help of the *Artifact* component of SACM, in an **Artifact** element, we are able to record inside it: **(1)** the type of an engineering artifact (e.g. EMF model, XML document, Excel spreadsheet, Isabelle theory file, etc.) to trace to; **(2)** the location of the engineering artifact; and **(3)** the meta-data of the engineering artifact (e.g. metamodel, XML metadata, etc.). To obtain a more fine-grained traceability, we make use of the **ImplementationConstraint** (discussed in Section 2.3) element, and record model querying/validation programmes written in a model querying language (in this work we support the Epsilon Object Language (EOL) (Kolovos et al., 2006), but any other languages can be supported) in **Artifact** model elements. In this way, when the programme is executed, we are able to obtain specific model element(s) (or information) from the engineering artifact that an *Artifact* element refers to, which can be used in assurance cases to provide contextual and evidential information. In summary, traceability from an assurance case to external engineering artifacts is achieved by referring to **Artifact** elements (organised in an **ArtifactPackage**) that contain traces to engineering models, from (in GSN terms) either an **Context** or an **Solution** element organised in a GSN **Module**.

With the traceability from an assurance case to engineering artifacts, we are also able to perform automated assurance case evaluations (i.e. validation and verification on referenced external engineering artifacts). For validation, we refer to external engineering artifacts using model elements defined in SACM's *Artifact* component, and embed programmes such as validation rules (which return *true* or *false*). ACME executes the validation rules and reflects the validation results to ACME editors so that the users can find out which part of an assurance case failed in the evaluation. For formal verification, we refer to Isabelle *theory documents*s. A theory document is a hierarchical structure consisting of formal artifacts, such as data types, functions, theorems, and proofs. Upon execution, ACME sends the *theory files* to the established Isabelle server (discussed in Wenzel (2019)), which can be communicated with using an RESTful API. When a theory file is sent to the Isabelle server, the server processes the file and returns JSON messages conveying the status of all artifacts contained within the theory file. If the processing of any artifacts fail, the JSON messages contain all the problems that Isabelle found. For example, a candidate proof could fail to prove a theorem, and this would raise an error. In ACME, we trace to an Isabelle theory file with an **Artifact** and perform formal notation checking in an automated manner. If an Artifact cannot be verified, ACME reflects this information in the model editor. Moreover, ACME also supports the translation of an assurance case to an Isabelle/SACM theory (Foster et al., 2019, 2021), which can be used for verifying its logical integrity. If there are any errors in other assurance case nodes, these are likewise reflected.

The support for the ACCESS methodology from ACME is illustrated in Fig. 10. Whilst in this section we illustrate the support from ACME, we argue that any model-based assurance case management framework may apply the ACCESS methodology to develop safety–critical systems.

For ACCESS **Step 2** and **Step 3**, we use ACME to create and manage a model-based assurance case, which may contain a number of **ArgumentPackage**s, **TerminologyPackage**s and **ArtifactPackage**s. In the figure we show how elements inside **ArgumentPackage**s can link to elements in **ArtifactPackage**s.

In ACCESS **Step 2**, **3** and **4**, various engineering artifacts (such as requirement models, architecture models, safety analysis models, and behavioural models) are produced, and reside alongside the assurance case (these are shown on the bottom right corner).[2]

In ACCESS **Step 4**, the evaluation of engineering artifacts can be automated within ACME, with the assurance case as the entry point. With SACM's **ImplementationConstraint** (IC) model element, we create validation rules written in EOL (*IC_1 and IC_2 in Fig.* 10) for the referenced artifacts. Based on the type of the engineering artifacts, ACME determines if it should invoke Epsilon or Isabelle. The results of the evaluations are processed by ACME, and if any problem occurs, they will be marked on the *Artifact*s that contain the evaluation rules in the ACME editor.

In ACCESS **Step 5**, we perform the evaluation of the entire assurance case, which includes the verification and validation on the system level. Using ACME, we perform two types of evaluation. The first type is to invoke evaluation on all referenced engineering artifacts from the assurance case. In ACME we provide an *evaluate* function which can be called on an assurance case, ACME then automatically looks into all **ArgumentPackage**s and looks for argument elements that refer to artifact elements in **ArtifactPackage**s, then ACME automatically calls Isabelle or Epsilon and determines if all evidence (that support the assurance case) are valid. The second type is to formalise the assurance case and check its logical integrity in an automated manner. We do this by transforming the assurance case to Isabelle/SACM notations with ACME's built-in model-to-text transformation (written in the Epsilon Generation Language (Rose et al., 2008) — EGL) and generate the Isabelle/SACM notation representation of the assurance case in the form of a *theory file*. The Isabelle/SACM notation is then sent to the Isabelle server for machine-checking. The evaluation result will be parsed by ACME, which then locates the model elements in the assurance case that fail the evaluation, and displays their corresponding error messages.

In ACCESS **Step 6**, elements of the assurance case can be converted to dynamic ones, whose validity depend on runtime data. In order to do this, ACME allows the creation of *Runtime Data Driver*s, which provide connections between runtime data and engineering artifacts. In this way, runtime data can be constantly reflected to engineering artifacts.

In ACCESS **Step 7**, ACME's runtime component — *Dynamic Safety Management System* (*DSMS*) is used to evaluate the assurance case. *DSMS* performs automated periodic evaluation on the assurance case, it does so by automatically invoking validation rules on dynamic parts of the assurance case (which refer to *Engineering Artifact*s that are updated

---

[2] In this paper we focus only on Engineering Artifacts that can be automatically validated with Epsilon, and Formalisations that can be automatically verified by Isabelle.

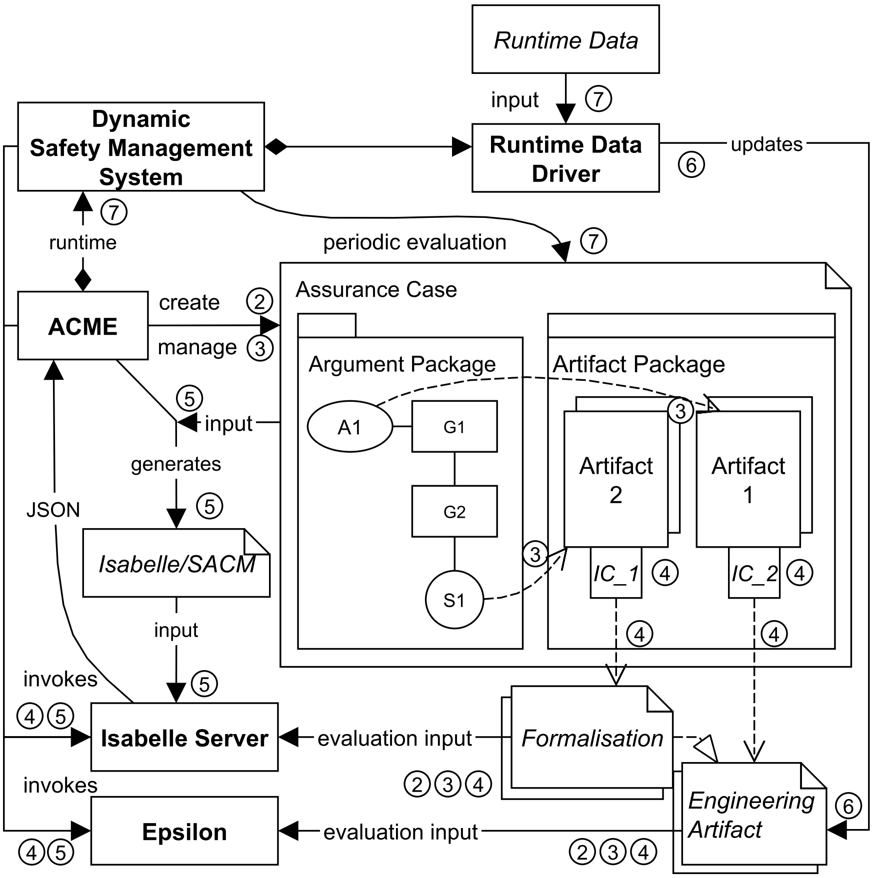


Fig. 10. ACME workflow in the context of ACCESS.

by *Runtime Data Drivers* at runtime). In this way, we can assure the assumptions about the system (e.g. its behaviour or its operational environment), check dynamic evidence, and determine operation contexts, which form the baseline for dynamic assurance case evaluation.

5. Case study

In this section, we evaluate the ACCESS methodology by applying it to a development process of an Autonomous Underwater Vehicle (AUV). The assurance case for the AUV is developed based on an integrated RoboChart (Miyazawa et al., 2019) model, which includes the architecture of the AUV and the behaviour of its controllers, discussed in Foster et al. (2020). We also show that it is possible, with the help of ACME, to automate the evaluation of model based assurance cases and have the assurance case *drive* the development of the system.

It is to be noted that due to confidentiality and the complexity of the AUV, in this case study we only demonstrate activities in the system development and system assurance of ACCESS where the benefits of automation can be reflected.

5.1. ACCESS Step 1

The AUV is a portable untethered remotely operated vehicle, equipped with a visual mapping system and verified on-board autonomy. The aim is to make it capable of conducting light intervention tasks, such as cathodic protection surveys (oil and gas) and simple coring (offshore), with potential to move to more complex interventions in a later phase, such as valve turning. The project brings together the UK expertise from: the National Oceanography Centre and Forth Engineering in Underwater Robotic Development; ROVCO on subsea operation, sensor development and subsea vision preception; and D-RisQ in Software Verification.

The National Oceanography Centre engages with regulators through their ongoing contribution to the Marine Autonomous Systems regulatory working group to ensure regulatory compliance. To this end, the use of a structured assurance case is vital to communicate the evidence of safety operation to non-specialists, especially in the aspect of software controlled autonomous behaviour.

In **ACCESS Step 1**, system functions are specified (task **S1**) and safety analysis on the system level is performed (task **A1**), and functions of the subsystems are allocated (task **S2**). We do not show the outputs of these tasks in detail due to confidentiality, but it is to be noted that the requirements are also model-based.

5.2. ACCESS Step 2

The overall architecture of the AUV is modelled (task **S3**) using the RoboChart language (Miyazawa et al., 2019), shown in Fig. 11. The robotic platform (*AUV_Platform*) acts as an abstraction layer for the hardware, and provides shared variables for sensors, actuators and events. The operator, which can be a human or navigation system, provides instructions to the LRE (*LRE_Ctrl* - Last Response Engine) to support execution of tasks, such as requesting a particular heading and velocity. The LRE sits between the operator (*AUV_Operator*) and the autopilot component (*AUV_Autopilot*). The LRE's job is to avoid hazardous behaviours, such as getting too close to an obstacle, or

Table 1
Fragment of the Failure Mode and Effect Diagnostic Analysis (FMEDA) for the AUV.

Component ID	Failure rate	Safety related	Failure mode	Failure mode distribution	Safety goal violation	Safety mechanism	Failure mode coverage by safety mechanism	SPF/RF
D1	10	Yes	Open	30%	Yes	None	0%	3
			Short	70%				
C1	2	Yes	Open	30%				
			Short	70%				
C2	2	Yes	Open	30%				
			Short	70%				
L1	15	Yes	Open	30%	Yes	None	0%	4.5
			Short	70%				
R1	1	No	Open	30%				
			Short	70%				
Lamp1	150	No	Open	100%				
U1	100	Yes	RAM	100%	Yes	ECC	99%	1

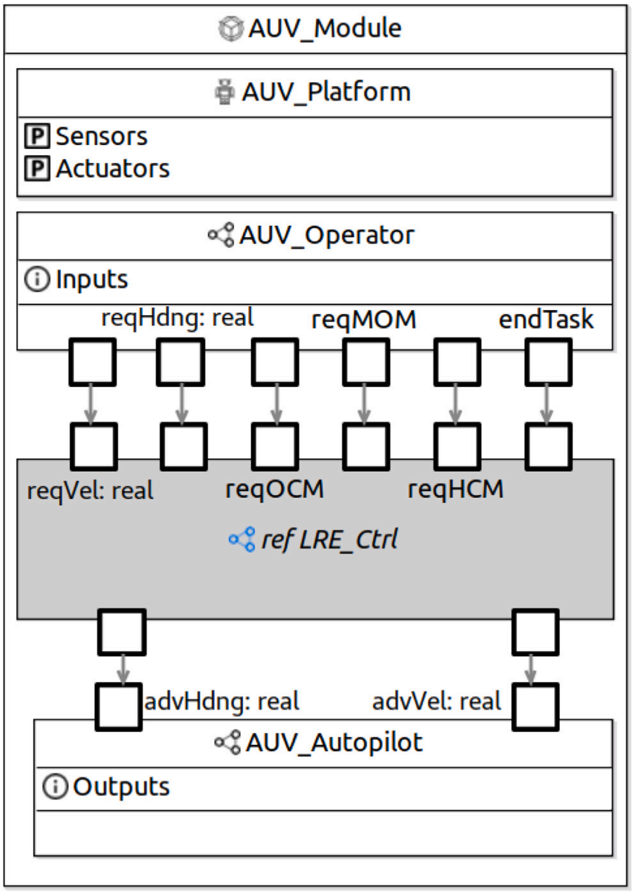


Fig. 11. Overall architecture of the AUV.

entering *Object Proximity Exclusion Zones* (OPEZs), and engaging evasive manoeuvres if necessary. The autopilot controls the AUV actuators, and takes advice only from the LRE.

Based on the architectural design, we create a modular assurance case (task **A2**) for the AUV, as shown in Fig. 12. It contains 5 *argument packages* (see Section 2.3), which are represented as GSN modules. The *AUV_system* module contains the argument of system level safety for the AUV, including hazard analysis and allocation of safety requirements. It is supported by modules *Platform_Argument*, *Operator_Argument*, *LRE_Argument* and *Autopilot_Argument*. This means that the validity of *AUV_System* depends on the validity of all 4 modules that support it. In addition, *LRE_Argument* depends on *Platform_Argument* and *Autopilot_Argument*.

5.3. ACCESS Step 3

We then perform safety analysis on subsystems (task **A3**). In critical systems development, it is often required that inductive and deductive safety analysis to be performed. One typical analysis that is performed frequently in the development of safety–critical systems is the Failure Mode and Effect Diagnostic Analysis (FMEDA). FMEDA looks at the failure mode, failure mode distribution and failure rate of system components (from simple components such as capacitors to more complex components such as Microcontrol Units), as well as the safety mechanisms to prevent failures, in order to compute hardware design metrics (e.g. Single Point Failure Metrics — SPFM) to determine the safety integrity levels of components. In this case study we demonstrate how we could establish traces to FMEDA from the assurance case in ACME, and automatically validate the assurance case using the FMEDA results as evidence.

ACME allows the traceability to engineering artifacts defined in arbitrary modelling technologies. One particular type of engineering artifact is Excel spreadsheets, which is often used in FMEDA. Table 1 shows a fragment of the FMEDA performed on the power supply of the proximity sensor (in which D1 is a Diode, C1 and C2 are capacitors, L1 is an inductance, R1 is a resistor, Lamp1 a lamp and U1 a Microcontrol Unit). Note that the unit of *Failure Rate* is *Failures in Time* (FIT), which is 10^{-9} times/h, where SPF stands for Single Point Faults and RF stands for Residual Faults (faults that are not covered by safety mechanisms). With the FMEDA, it is possible to compute the SPFM of a particular hardware component, using the formula:

$$SPFM = 1 - \frac{\sum_{SR}(\lambda_{SPF} + \lambda_{RF})}{\sum_{SR}\lambda}$$

where $\lambda_{S}PF$ is the failure rate associated with hardware element single-point faults, $\lambda_{R}F$ is the failure rate associated with hardware element residual faults, and λ is the failure rate associated too all faults.

```
var entries = FMEDA.all();
var safety_related = 0;
var spf_rf = 0;
for(e in entries) {
  if(e.SafetyRelated = "Yes") {
    safety_related += e.FailureRate.asReal();
  }
  if(e.SafetyGoalViolation = "Yes") {
    spf_rf += e.SPF_RF.asReal();
  }
}
var spfm = 1 − (spf_rf)/safety_related;
return spfm > 0.9;
```

Listing 1: Computing the SPFM for a hardware component.

To validate the FMEDA, we aim to achieve the target SPFM (assume we aim to achieve 90%). In order to do this automatically in ACME, in the assurance case, we create an **ArtifactPackage** (discussed in

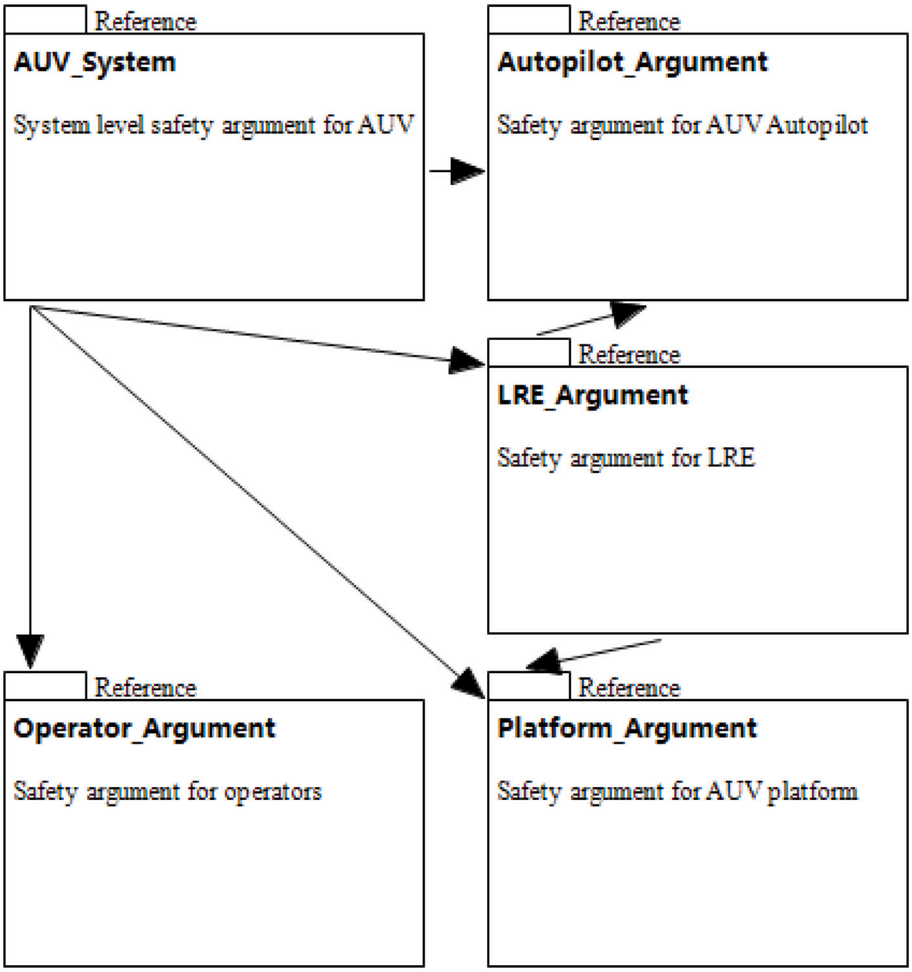


Fig. 12. Overall AUV safety argument structure.

Section 2.3), and in it create an **Artifact**, and we refer to the Excel spreadsheet from the **Artifact**, which can be later used as evidence to our safety argument in the assurance case. As shown in Fig. 13, in the *references* section, we configure the “Model Type” (as Excel spreadsheet), the “Document” (to the location of the Excel spreadsheet), and the “Metadata” (empty in this example). Within ACME, we use **ImplementationConstraint**s to store model validation rules in **Artifact** elements so that such rules can be executed when we evaluate the assurance case. In Fig. 13, in the “Constraint” section, we use the rule in Listing 1 (written in the Epsilon Object Language — EOL (Kolovos et al., 2006)) to check if the FMEDA fulfils our requirement for the target SPFM value. During the development process, when the FMEDA changes, ACME can automatically detect this change by the execution of the query and compute the SPFM automatically, then show the users if the hardware design fails to meet the target SPFM values.

With the safety analysis performed, we are able to further refine our assurance case by arguing the safety of subsystems (task **A4**), a fragment of the assurance case is shown in Fig. 14. However, we shall note that in this ACCESS step, only *C6_a* and *Autopilot* in Fig. 14 are developed (other elements are defined in the next ACCESS step). Also, at this stage, there are no traceability from the argument to the supporting evidence yet.

5.4. ACCESS Step 4

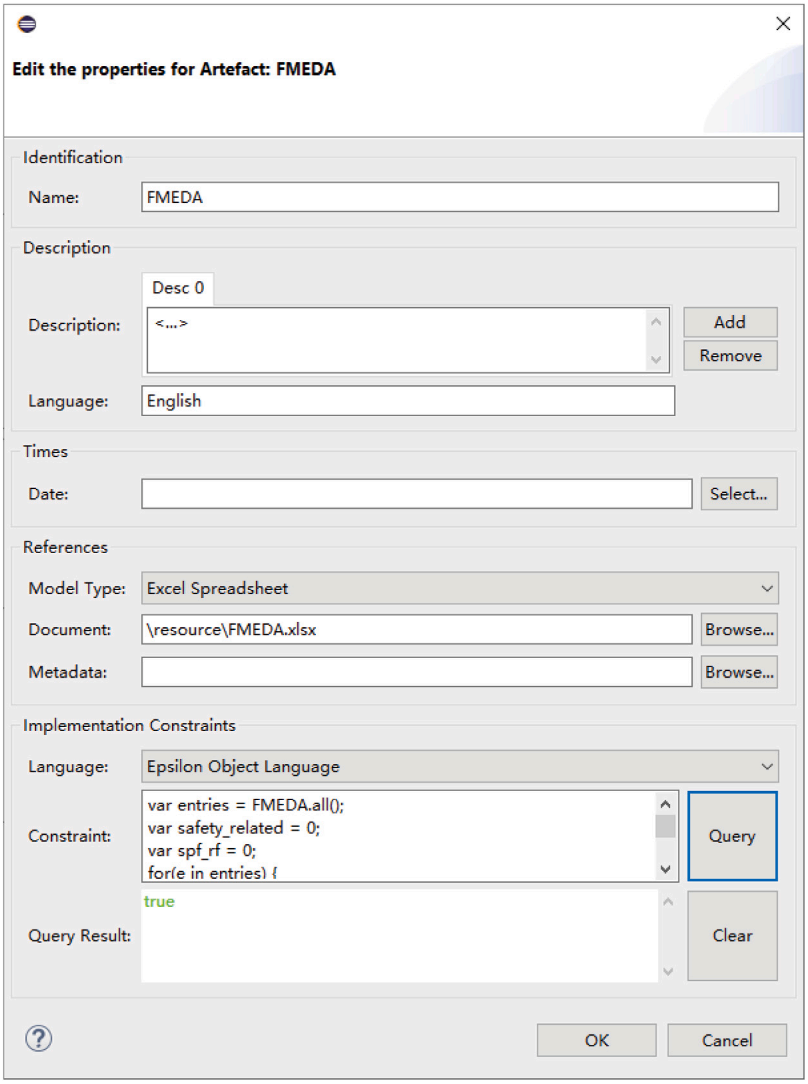
5.4.1. The last response engine

In this paper, we focus on the development of the LRE, which provides run-time safety assurance. We consider the case of the AUV navigating within an enclosed pond to perform maintenance tasks. There are two main hazards for the AUV that we consider: (1) collisions with static and dynamic obstacles and (2) causing a splash, which can be a hazard for workers and equipment around the pond. The AUV can either be under operator control, or running autonomously. If operating autonomously, the responsibility for satisfying the safety requirements lies with the LRE, which can engage evasive manoeuvres if necessary. There are also *Object Proximity Exclusion Zones* (OPEZs), which are designated areas where the AUV may not operate autonomously, and help with hazard mitigation. They include the area close to the pond wall, and also the area just below the water surface.

The LRE functions in four modes: Operator Control Mode (OCM), Main Operating Mode (MOM), High Caution Mode (HCM) and Collision Avoidance Mode (CAM). In OCM, the LRE passes control inputs from the operator to the autopilot. In MOM, the LRE takes control for normal behaviour at maximum speed. HCM is for the situation when the AUV is getting close to an obstacle, and so the LRE lowers the velocity. Finally, CAM is the mode where a potential unavoidable collision has been detected, and the AUV is manoeuvring away from the obstacle.

The LRE keeps an *obstacle register*, which stores identified obstacles, through sensor readings. In each behavioural cycle, the LRE calculates the closest obstacle and determines whether it should apply evasive manoeuvres or switch into high caution mode (HCM).

There are six event inputs: *reqVel*, with which the operator can request a velocity; *reqHdng*, to request a new heading; *reqOCM*, *reqMOM*, *reqHCM*, to request an operation mode; and *endTask*, to delineate tasks. The two output events are *advVel* and *advHdng*, with which the LRE can send instructions to change velocity or heading to the autopilot.



Edit the properties for Artefact: FMEDA

Identification

Name: FMEDA

Description

Desc 0

Description: <...>

Language: English

Times

Date:

References

Model Type: Excel Spreadsheet

Document: \resource\FMEDA.xlsx

Metadata:

Implementation Constraints

Language: Epsilon Object Language

Constraint:

```
var entries = FMEDA.all();
var safety_related = 0;
var spf_rf = 0;
for(e in entries) {
```

Query Result: true

Fig. 13. Reference to FMEDA in excel spreadsheet.

5.4.2. *Behaviour model for the LRE*

We now model the behaviour of the LRE (task **S5**). We create a state machine for the LRE, shown in Fig. 15. It implements the LRE's behavioural requirements and specifies the conditions on switching to different operation modes. The following definitions and functions appear in the state machine (Miyazawa et al., 2019): *vel* (velocity of the AUV), *inOPEZ* (if the AUV is in an OPEZ), *CDA* (Closest Distance of Approach), *StaticObsHorizDist* and *StaticObsVertDist* (shortest distance allowed to an obstacle horizontally and vertically), *MinSafeDist* (minimal overall safe distance), *cdyn* (closest dynamic obstacle), *cstc* (closest static obstacle), *hdist()* (horizontal distance to an obstacle), *vdist()* (vertical distance to an obstacle), *odist()* (overall distance to an obstacle).

The transitions give (1) events that trigger the transition; (2) the conditions under which they can fire, and (3) any action taken at that point. For example, the top-left most transition in Fig. 15 is

$$reqMOM \begin{bmatrix} vel \le 0.1 \wedge odist(cdyn) > 7.5 \\ \wedge odist(cstc) > 0.3 \wedge \neg inOPEZ \end{bmatrix}$$

It states that the LRE can move from OCM to MOM when the trigger event $reqMOM$ is received from the operator, and the set of conjoined conditions hold. Specifically, the AUV can only operate autonomously provided it has a low velocity, a minimum distance to static and dynamic obstacles, and the AUV is not in an OPEZ. The state MOM has an entry action, $advVel!1$, that is executed when the state is activated from any transition, and advises the autopilot to set the velocity to the maximum. The top-most transition from MOM to HCM has no trigger action, and only the condition

$$[hvel \ge 0.1 \wedge hdist(cstc) \le StaticObsHorizDist]$$

attached, meaning that it will activate as soon as the sensor values enter the characterised range.

5.4.3. *LRE argumentation*

With the behavioural model for the AUV defined, we discuss model analysis and verification (task **A5**) and further refined the assurance case (created using ACME) for the LRE,.

We focus on the scenario of static obstacle avoidance for the LRE, the safety argument fragment of which is shown in Fig. 14. The top level **Goal** *C6_a* states that upon detecting a close static obstacle, LRE should advise the autopilot to switch to HCM and reduce the velocity of the AUV to 0.1 m/s. *C6_a* is a public goal (indicated by the module icon on the top right corner) as it is used by the overall safety argument in the *AUV System* module, unlike other goals, which are private.

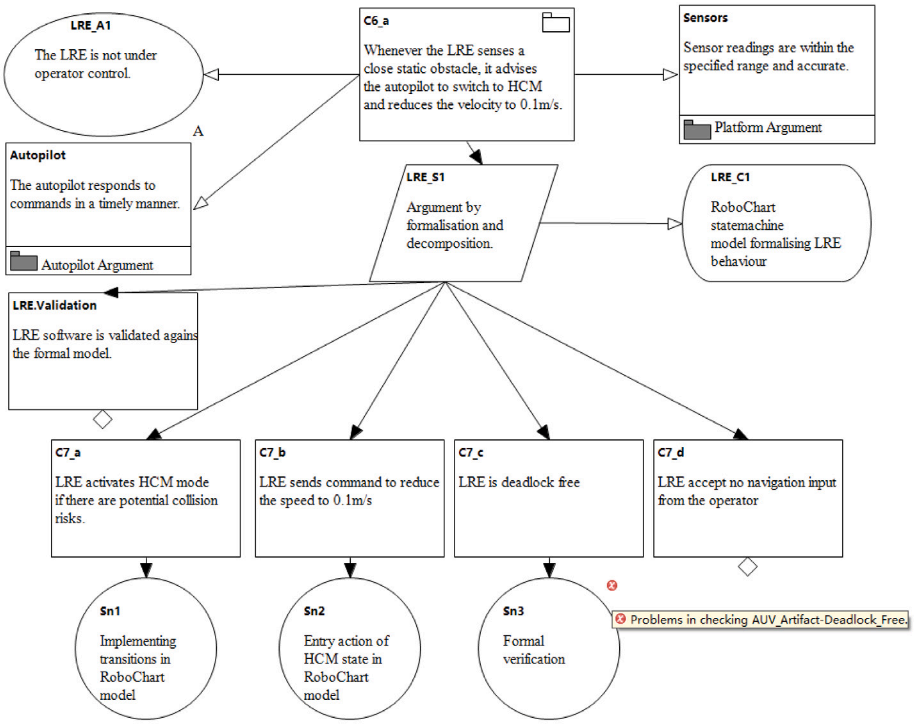


Fig. 14. Fragment in the LRE assurance case module to argue the safety of static obstacle avoidance.

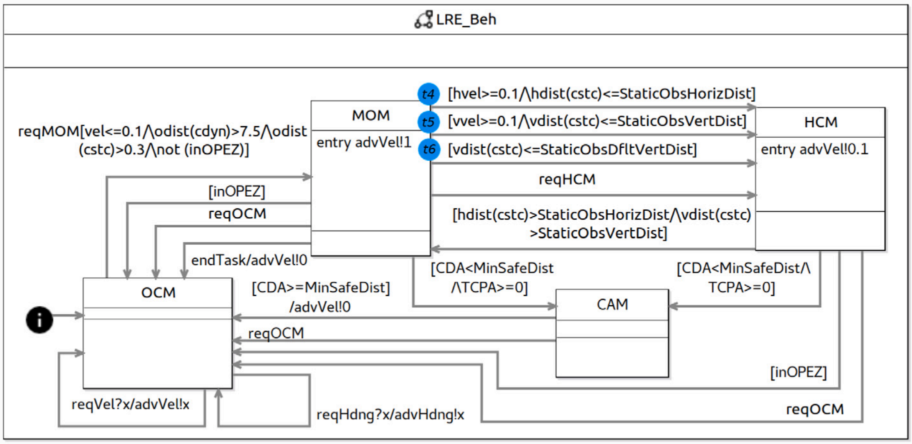


Fig. 15. LRE RoboChart state machine.

C6_a is in the context of, and thus contingent upon, **Assumption** *LRE_A1*, and **Away Goal**s *Autopilot* and *Sensors*. The away goals must be supported in the *Platform* and *Autopilot* modules for the LRE module to be valid. *LRE_A1* ensures that the argument need only hold when the operator is not in control; the alternative case is handled by the *Operator* module. We support *C6_a* by formalisation and decomposition. **Strategy** *LRE_S1* states our argument strategy, which is in the context of **Context** *LRE_C1*.

We focus on **Goals** *C7_a*, *C7_b* and *C7_c*. They use the RoboChart model to establish that the safety requirement is indeed satisfied. They are subject to a validation argument under *LRE.Validation*, which is left undeveloped for now, but should include activities like software testing. In *C7_a*, we state that the LRE should activate HCM if there are potential collision risks. We support this **Goal** with *Solution Sn1*, which states that transitions to HCM mode from MOM should be modelled by the behavioural model in Fig. 15. In *C7_b*, we state that the LRE should send a command to the autopilot to reduce the speed to 0.1 m/s, and we support this with **Solution** *Sn2*, which states that the entry action of HCM should reduce the speed to 0.1 m/s. In *C7_c*, we state that the LRE is deadlock free, and support this with **Solution** *Sn3* by formal verification.[3]

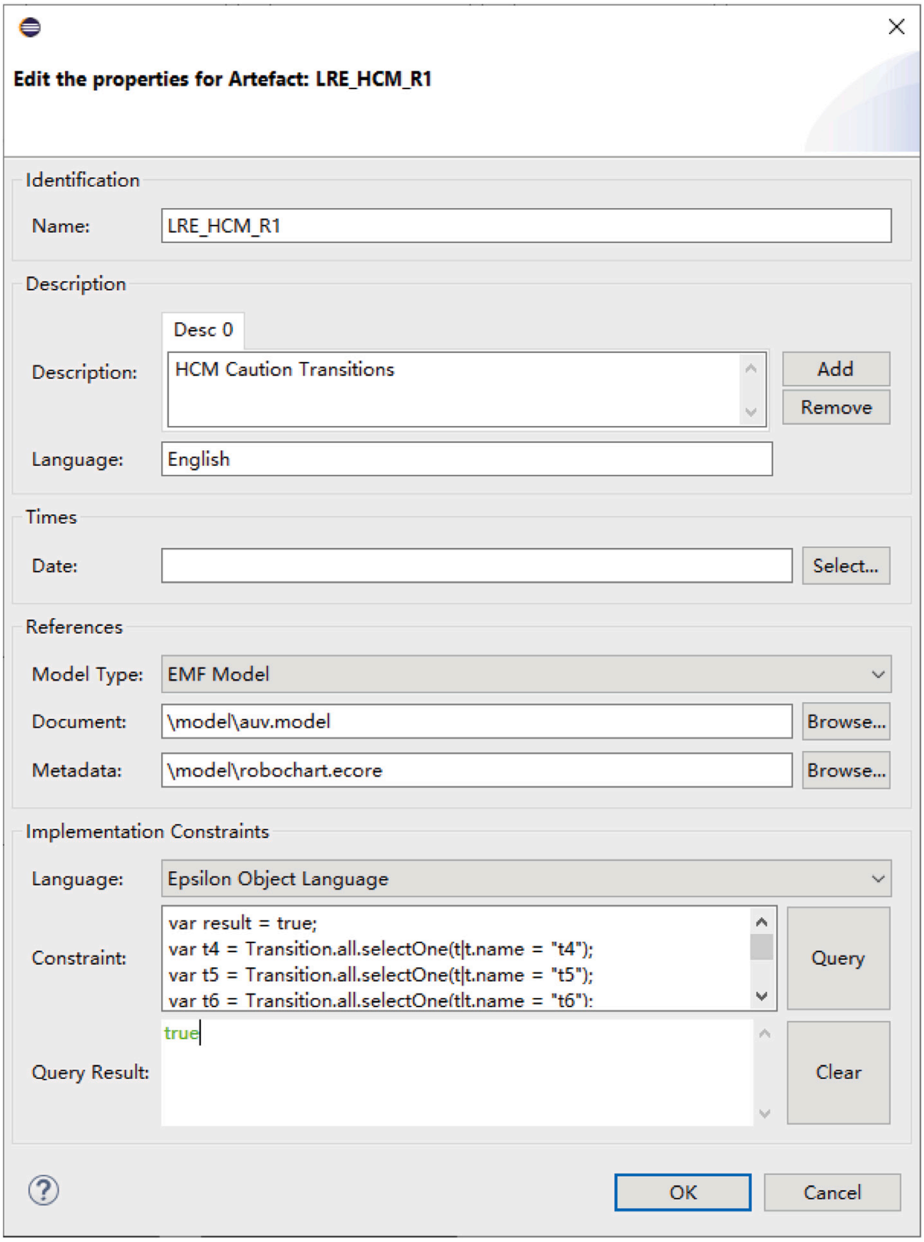
At this stage, we achieve traceability to formal verifications inside the assurance case. But the assurance case is by no means complete, since systematic verification and validation are yet to be performed.

5.5. ACCESS Step 5

In this step, the system shall be implemented (task **S6**), and we perform system verification and validation (task **A6**) from assurance

---

[3] We will explain the error marker on Sn3 later.



Name: LRE_HCM_R1

Description: HCM Caution Transitions

Language: English

Date:

Model Type: EMF Model

Document: \model\auv.model

Metadata: \model\robochart.ecore

Language: Epsilon Object Language

Constraint:

```
var result = true;
var t4 = Transition.all.selectOne(t|t.name = "t4");
var t5 = Transition.all.selectOne(t|t.name = "t5");
var t6 = Transition.all.selectOne(t|t.name = "t6");
```

Query Result: true

Fig. 16. ACME dialog to edit **Artifact** *LRE_HCM_R1*.

case. We do so by complete the traceability to all engineering artifacts from the assurance case, and automate the evaluation from ACME.

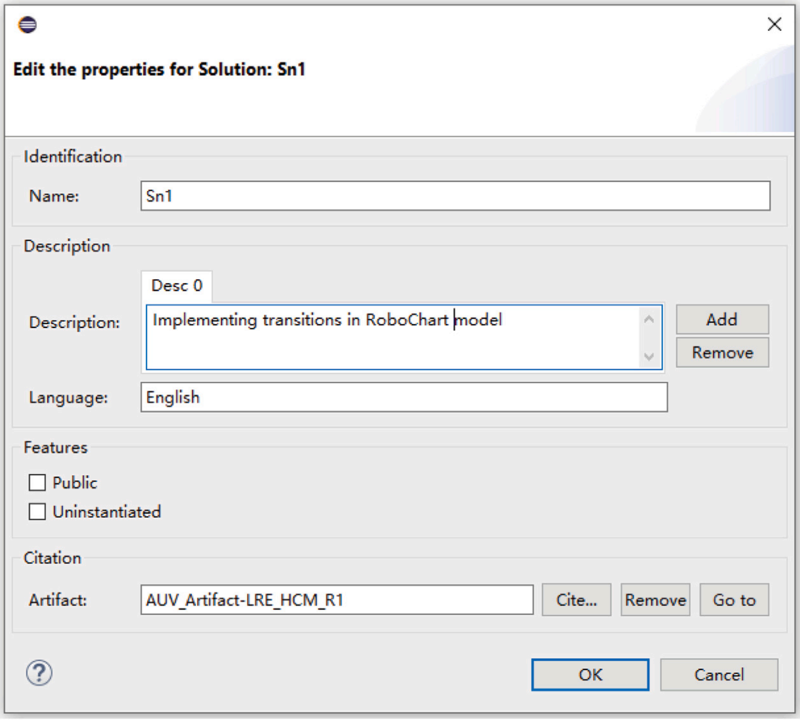
5.5.1. *Trace to EMF models*

GSN elements such as **Context**s and **Solution**s, can refer to models/documents external to the assurance case. With traditional GSN approaches, references to external models/documents are informal and their evaluation is often performed manually.

We illustrate traceability with **Goal** *C7_a* and its supporting **Solution** *Sn1* (in Fig. 14), which in turn is supported by several transitions in the RoboChart state machine. To be able to reference elements of the RoboChart model shown in Fig. 15, we create an **Artifact** named *LRE_HCM_R1* (in an **ArtifactPackage** named *LRE_Artifact*), which will be referenced by **Solution** *Sn1*. The properties of *LRE_HCM_R1* are shown in Fig. 16. In the properties view for an **Artifact**, we specify the "Model Type" (we currently support EMF models, Excel spreadsheets and plain text files), "Document" (location of the model) and the "Metadata" (metadata of the document, which can be metamodels, schemas, etc.), in the "References" section in Fig. 16 (Note that assurance cases and its referenced engineering models should reside in the same location).

We then attach the model validation rule in Listing 2 in the "Constraint" section. In this rule we check that there are at least 3 transitions from MOM, named "t4", "t5" and "t6" (shown in Fig. 15), which are triggered when there are potential collision risks. For readability we only show the queries for Transition "t4". The user can evaluate the query inside the dialog by pressing the "Query" button. ACME will load the model specified in the "Reference" section and execute the query, the result of which is displayed in the "Query Result" text field. It is to be noted that the validation rules do not have to be specified by EOL, in a separate publication (Wei et al., 2023b), we illustrated the use of constraint natural language with model-based approach.

```
var result = true;
```



Edit the properties for Solution: Sn1

Identification

Name: Sn1

Description

Desc 0

Description: Implementing transitions in RoboChart model

Language: English

Features

☐ Public

☐ Uninstantiated

Citation

Artifact: AUV_Artifact-LRE_HCM_R1

Fig. 17. ACME dialog to edit **Solution** *Sn1*.

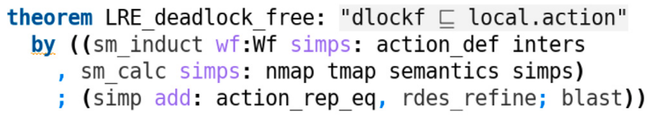
```
var t4 = Transition.all.selectOne(t|t.name = "t4");
var t5 = Transition.all.selectOne(t|t.name = "t6");
var t6 = Transition.all.selectOne(t|t.name = "t7");
var t4c = t4.condition;
var t4check = t4c.isTypeOf(And) and
t4c.left.isTypeOf(GreaterOrEqual) and
t4c.left.left.ref.name = "hvel" and
t4c.left.right.value = 0.1 and
t4c.right.isTypeOf(LessOrEqual) and
t4c.right.left.isTypeOf(CallExp) and
t4c.right.left.function.ref.name = "hdst" and
t4c.right.left.args.first.ref.name = "cstc" and
t4c.right.right.ref.name = "StaticObsHorizDist";
result = result and t4check;
return result;
```

Listing 2: Query the transitions in the RoboChart model.

Artifact *LRE_HCM_R1* can then be used as a supporting evidence for our assurance case (specifically to substantiate **Solution** *Sn1*). To do this, within *Sn1* we "cite" the *LRE_HCM_R1* (defined in the *AUV_Artifact* package) in the "Citation" section, shown in Fig. 17.

5.5.2. *Trace to Isabelle theory files*

In this work, we also support references to formal notations embedded in Isabelle theory files. The process of referring to an Isabelle theory file is the same described in Section 5.5.1, except that the users need to select "Isabelle Theory File" in the "Model Type" drop down menu (within the "References" section) in the property dialog of an **Artifact**, so that ACME knows to invoke the Isabelle server to check the referenced theory file. Upon assurance case evaluation, ACME sends the Isabelle theory file (*Formalisation*) to the *Isabelle Server*, which checks the theory file and returns a JSON string, ACME then parses the string and marks the errors in the **ArtifactPackage** editor. We illustrates this by injecting an error and shows this in ACME, as shown in lower part of Fig. 10. For this work, we create an **Artifact** called *Deadlock_Free* and refer to the theorem shown in Fig. 18. This uses automated proof tactics in Isabelle to prove deadlock freedom for the LRE state machine in Fig. 15. We then "cite" *Deadlock_Free* within **Solution** *Sn3* in Fig. 14.



```
theorem LRE_deadlock_free: "dlockf ⊑ local.action"
  by ((sm_induct wf:Wf simps: action_def inters
    , sm_calc simps: nmap tmap semantics simps)
    ; (simp add: action_rep_eq, rdes_refine; blast))
```

Fig. 18. Deadlock free theorem defined in Isabelle.

5.5.3. *Automated assurance case evaluation*

With engineering artifacts referenced from our assurance case, we can evaluate the assurance case by invoking the "evaluate" function, which users can select in the context menu provided by ACME. When we evaluate the assurance case in ACME, ACME starts the evaluation process from the assurance case diagram where the "evaluate" function is invoked. ACME automatically traces to **Artifacts** from **Solutions**, **Contexts**, and **Assumptions**. Then, depending on the types of the engineering artifacts (this information is associated to the **Artifacts**), ACME executes model queries (for model validation) or invokes the Isabelle server (for formal verification), respectively. Fig. 14 shows an error marker on **Solution** *Sn3*, which indicates that the Isabelle proof in Fig. 18 referenced from **Artifact** *Deadlock_Free* was unsuccessful.

This process of evaluation can be performed at regular intervals to ensure that updates to models and other artifacts do not invalidate the assurance case. For example, if one of the transitions from MOM to OCM is removed in the behavioural state machine, ACME will be able to pick up this change and flag an error. Moreover, if a change to the state machine introduces deadlock, this will also be flagged by the failure of the proof in Fig. 15. This is typically the benefits by adopting ACCESS, ACCESS assumes that model-based approaches are used in the development process, hence boosting the efficiency of developers and improving the correctness and coverage of assurance case evaluation activities.

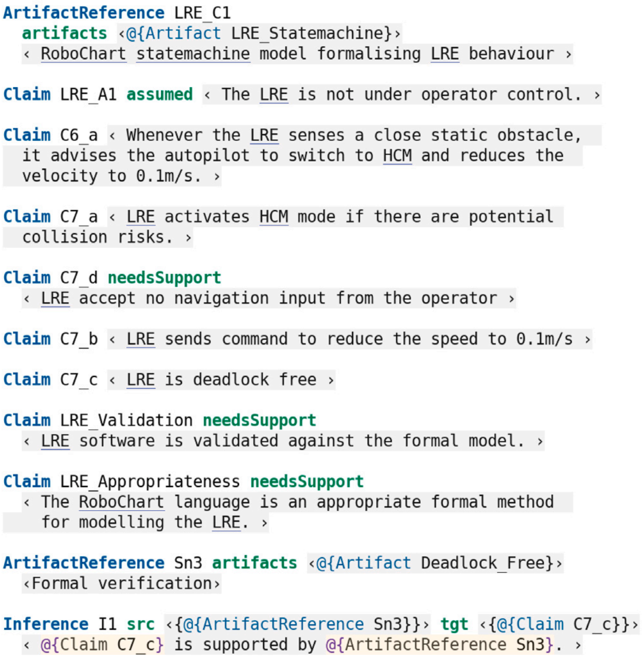
5.5.4. Transformation to Isabelle/SACM

Once a satisfactory assurance case is developed, a further step is to check the integrity of the overall assurance case formally using Isabelle/SACM. This is an optional step for ACCESS, but we would like to show that how formalism can be integrated with model-based assurance case to form a more convincing assurance case. To do this, we perform a model-to-text transformation to generate Isabelle/SACM notations from the assurance case automatically, which can be machine-check using the Isabelle server. In ACME, we use the Epsilon Generation Language (EGL) (Rose et al., 2008) to implement the transformation, but the transformation can be generalised. Algorithm 1 shows the pseudo-code for generating Isabelle/SACM notations from a GSN module.

Algorithm 1: Generating Isabelle from GSN.

```
for element in {all Contextual Elements} ∪ {all Goals} do
    let declarations = {“”, “axiomatic”, “assumed”, “needsSupport”, “asserted”};
    let declaration = determined based on the feature/type of element
    output “Claim ” + element.name + declaration + “‹” + element.description + “›”
end
for element in {all Solutions} do
    output “ArtifactReference” + element.name + declaration + “‹” + element.description + “›”
end
for element in {all Strategies} do
    let target = incoming SupportedBy
    let sources = outgoing SupportedBys
    let source_names = “@{Claim” + source names separated by “,” +“}”
    output “Inference ” + element.name + “ src ‹{” + source_names + “}› tgt ‹{@{Claim ” + target.name + “}}›” + “ ‹@{Claim ” + target.name + “} is supported by ” + source_names + “.›”;
end
for element in {all Relationships Not Processed} do
    let source_name = element.target.name;
    let target_name = element.source.name;
    if element isTypeOf(GSN!SupportedBy) then
        if source.isTypeOf(GSN!Solution) then
            output “Inference ” + element.name + “src ‹{@{ArtifactReference ” + source_name + “}}› tgt ‹{@{Claim” + target_name +"}}›" + “ ‹@{Claim ” + target_name + “ } is supported by @{ArtifactReference ” + source_name + “}.›”
        else
            output “Inference ” + element.name + “src ‹{@{Claim ” + source_name + “}}› tgt ‹{@{Claim” + target_name +“}}›” + “ ‹@{Claim ” + target_name + “ } is supported by @{ArtifactReference ” + source_name + “}.›”
        end
    else if element isTypeOf(GSN!InContextOf) then
        output “Context ” + element.name + “src ‹{@{Claim ” + source_name + “}}› tgt ‹{@{Claim” + target_name + “}}›” + “‹@{Claim ” + target_name + “ } is context for @{Claim ” + source_name + “}.›”
    end
end
```

A fragment of transformed Isabelle/SACM argument is shown in Fig. 19, with some elements reordered to aid readability. **Context** *LRE_C1* and **Solution** *Sn3* both give rise to references to formal artifacts included from another Isabelle theory file (Foster et al., 2020), including the formalised LRE state machine. Several of the claims are left open, and so have the keyword *needsSupport* attached, which can be checked to ensure that all branches of the argument are developed. An **Inference**, *I1*, connects the formalised *Sn3* (the source) to the **Claim** *C7_c* (the target). The remainder of the claims, inferences, and solutions are similarly mapped.



```
ArtifactReference LRE_C1
  artifacts ‹@{Artifact LRE_Statemachine}›
  ‹ RoboChart statemachine model formalising LRE behaviour ›

Claim LRE_A1 assumed ‹ The LRE is not under operator control. ›

Claim C6_a ‹ Whenever the LRE senses a close static obstacle,
  it advises the autopilot to switch to HCM and reduces the
  velocity to 0.1m/s. ›

Claim C7_a ‹ LRE activates HCM mode if there are potential
  collision risks. ›

Claim C7_d needsSupport
  ‹ LRE accept no navigation input from the operator ›

Claim C7_b ‹ LRE sends command to reduce the speed to 0.1m/s ›

Claim C7_c ‹ LRE is deadlock free ›

Claim LRE_Validation needsSupport
  ‹ LRE software is validated against the formal model. ›

Claim LRE_Appropriateness needsSupport
  ‹ The RoboChart language is an appropriate formal method
    for modelling the LRE. ›

ArtifactReference Sn3 artifacts ‹@{Artifact Deadlock_Free}›
  ‹Formal verification›

Inference I1 src ‹{@{ArtifactReference Sn3}}› tgt ‹{@{Claim C7_c}}›
  ‹ @{Claim C7_c} is supported by @{ArtifactReference Sn3}. ›
```

Fig. 19. LRE argument fragment in Isabelle/SACM.

The transformed Isabelle/SACM theory file is sent to the remote Isabelle server for machine-checking. Inside ACME we parse the JSON string returned by Isabelle and find out if there are any problems. If so we can locate model-elements in the assurance case that cause problems and display them in ACME editors.

5.6. ACCESS Step 6

5.6.1. Dynamic assurance case evaluation for AUV at runtime

As previously discussed, the assurance of open adaptive RAS requires safety evaluation to be performed at runtime when the system is operational. To take a first practical step in this direction, we show how we can achieve dynamic assurance case evaluation with dynamic runtime data, in order to support the ACCESS process.

In our LRE assurance case in Fig. 14, we state that **Goal** *C6_a* is valid within the context of **Away Goal** *Sensors* (top right element in Fig. 14), which is a top level **Goal** specified in the *Platform_Argument* module (hence the SupportedBy relation in Fig. 12).

In module *Platform_Argument*, we define a **Goal** *Sensor.G2.a* shown in Fig. 20 (we omit other parts of the argument in the *Platform_Argument* module to aid readability), which supports the top level **Goal** in the *Platform_Argument* module. In turn, two more **Goal**s that support *Sensor.G2.a*: **Goal** *Sensor.G3.a* states that sensors should be sufficiently reliable to provide accurate readings, this is in turn supported by **Solution** *Sensor.Sn1*, where the hardware design metrics is quantitatively analysed by FMEDA (we omit the details of the analysis for readability); **Goal** *Sensor.G3.b* states that obstacle data should be in the specified range, which is supported by **Solution** *Sensor.Sn2*, where runtime evaluation is performed.

This is where the assurance case starts to deviate into a dynamic assurance case. Conventionally, assurance cases have static evidence in the argument structure, which means, **Goal** *Sensor.G2.a* needs only to be supported by **Goal** *Sensor.G3.a*, because there was no notion of detecting random hardware faults at runtime. In our approach, we

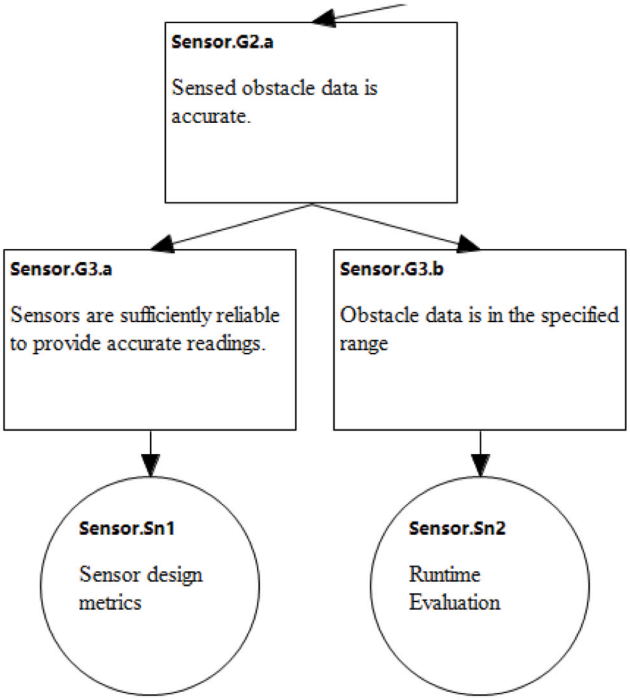


Fig. 20. **Goal** in the *Sensor* **Module** that requires runtime evaluation.

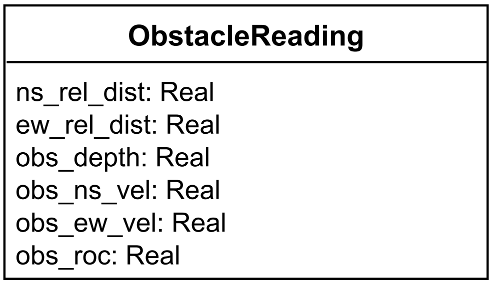
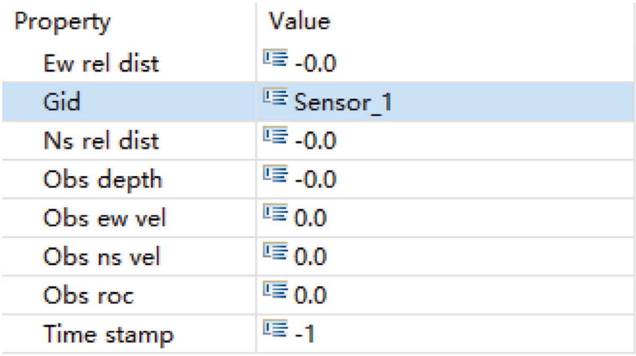


Fig. 21. ObstacleReading in the LRE runtime assurance metamodel.

argue that runtime random hardware fault can be detected by means of evaluating dynamic assurance cases. In this way, we can establish our confidence in the reliability of the entire system, due to the fact we could establish our confidence in the reliability of key components of the system (rather than trusting their manufacture specifications on failure rates). Therefore, **Solution** *Sensor.Sn2* is in place for ACME runtime component to perform dynamic validation (this corresponds to task **A7**).

To determine the reliability of the sensors, we need to synchronise their readings to models that we can refer to at runtime from our assurance case. For this purpose, we create an *LRE Runtime Assurance* metamodel. We are particularly interested in the *ObstacleReading* class in the metamodel, which is shown in Fig. 21. We are interested in 6 variables in an *ObstacleReading*, *ns_rel_dis* for relative north-east distance to the obstacle, *ew_rel_dist* for relative east–west distance to the obstacle, *obs_depth* for the depth of the obstacle, *obs_ns_vel* for north-east velocity of the obstacle, *obs_ew_vel* for east–west velocity of the obstacle, and then *obs_roc* for rate of climb of the obstacle. With the *LRE Runtime Assurance* metamodel, we create an *LRE Runtime Assurance* model, which contains default content before the AUV is deployed, shown in Fig. 22.



Property	Value
Ew rel dist	-0.0
Gid	Sensor_1
Ns rel dist	-0.0
Obs depth	-0.0
Obs ew vel	0.0
Obs ns vel	0.0
Obs roc	0.0
Time stamp	-1

Fig. 22. Example *ObstacleReading* in the sensor digital twin model.

In our **ArtifactPackage**, we create an **Artifact** *Obstacle_reading*, which refers to the *LRE Runtime Assurance* model. Inside *Obstacle_reading*, we attach the validation rule in Listing 3, which checks that the sensor readings are within the valid ranges for the AUV. By performing the above activities, we have converted some elements within the model-based assurance case into runtime elements, the validity of which would be determined by runtime data.

```
var r = M!ObstacleReading.all().first();
return (r.ns_rel_dist>=−50.0 and r.ns_rel_dist<=50.0)
and (r.ew_rel_dist>=−50.0 and r.ew_rel_dist<=50.0)
and (r.obs_depth>=−10.0 and r.obs_depth<=0.0)
and (r.obs_ns_vel>=−5.0 and r.obs_ns_vel<=5.0)
and (r.obs_ew_vel>=−5.0 and r.obs_ew_vel<=5.0)
and (r.obs_roc>=−5.0 and r.obs_roc<=5.0);
```

Listing 3: Evaluation rule for checking the well-formedness of obstacle readings.

5.7. ACCESS Step 7

For each class in the *LRE Runtime Assurance* metamodel, we use model-to-text transformation to generate a Java class *XXXDriver* (details omitted to aid readability). For example, for the *ObstacleReading* class, we generate *ObstacleReadingDriver*. To synchronise the sensor data, we create a Java class *ObstacleReadingDriver* which is a type of *Runtime Data Driver*, as discussed in Fig. 10. *ObstacleReadingDriver* provides an API which takes 6 parameters (*ns_rel_dist*, *ew_rel_dist*, *obs_depth*, *obs_ns_vel*, *obs_ew_vel* and *obs_roc*), for the AUV to write these values to the *AUV Sensor* model at runtime. For runtime evaluation, ACME provides a **Dynamic Safety Management System** (DSMS), as shown in Fig. 10.

In terms of model evaluation, we actively evaluate the assurance case by executing the "evaluate" function of the DSMS, which in turn executes all model validation rules embedded in the **Artifact**s (which support **Solution**s, **Context**s, **Assumption**s and **Justification**s) in the assurance case. In our prototype, the DSMS periodically evaluates the AUV assurance case (in this work we use a 50-millisecond intervals), which includes the evaluation of the **Solution** *Senso.G3.b*, which then triggers the evaluation of the **Artifact** *Obstacle_reading* by executing the evaluation rule in Listing 3 against the *AUV Sensor* model.

The runtime data is synchronised from the AUV to the *LRE Runtime Assurance* model through *ObstacleReadingDriver* (through simulation, as the AUV has not been developed for operational trials). If the readings are not within the range defined in the evaluation rule, the periodic evaluation of DSMS will fail, rendering the corresponding assurance case fragment invalid. In a sense, the development time assurance case is converted to a *model@runtime* with the DSMS framework. In this way, a novel means to perform additional checks at runtime (and reflect back to the assurance case) is proposed. A number of advantages comes with this approach:

- Random hardware faults is actively detected rather than analysed and calculated, this may result in a shorter *Time to Detect Fault*; in addition, this fault is propagated back to the assurance case, rendering it invalid immediately. At runtime, the system may consult DSMS, if the assurance case becomes invalid, it may make the transition to safe state (e.g. complete halt), result in shorter *Fault Reaction Time Interval*;
- The *LRE Runtime Assurance* model can be version-controlled, which makes it possible to accumulate history data for the corresponding sensor, therefore obtaining statistics on failure rates for the sensor, which can be further used in stochastic approaches approximating the fault rate of the sensors.
- Consequently, the safety analysis for the sensors can be updated accordingly, which we discussed in a recently published work (Wei et al., 2022, 2023a).

It is to be noted that **Context**s in the assurance case may also depend on sensor readings. For RAS, they may operate in different contexts, the safety arguments for such contexts may differ. Therefore, it is highly likely that dynamic assurance cases for RAS are no longer singular assurance case, but a repository of models, including a number of assurance cases for different operational contexts, engineering artifacts produced at development time, as well as *models@runtime* with synchronised sensor data. The repository of models should be monitored and version-controlled at runtime to aid automated assurance case evaluation and offline retrospection. To this point, we have concluded **ACCESS Step 7**. It is to be noted that ACCESS is an iterative process, development following ACCESS can start any ACCESS step in the development lifecycle, based on the justifications of the developers.

6. Evaluation

To evaluate the effectiveness and generality of ACCESS, we used our methodology to engineer the UAV (hereby referred to as System A) from Section 5, and a safety–critical autonomous robotic system (hereby referred to as System B) for earthquake aftermath search and rescue. The two systems were developed as described in Section 3, and were deployed in a realistic environment with changes in environment specific to their application domains. We examined the efficiency of the development process for both systems, and evaluate the generality of our methodology across different domains. In addition, we examine the coverage and the scalability of the supporting too, which provides insights on future research directions based on ACCESS and any model-based assurance case management environment. The aim of our evaluation was to answer the following research questions.

RQ1 (Efficiency): Does ACCESS, supported by model-based assurance case, increase efficiency of developers for safety–critical systems?

RQ2 (Generality): Does ACCESS support the development of safety–critical systems and model-based assurance cases across application domains?

Although the focus of research contribution is on the proposed ACCESS methodology, we also aim to answer the following research questions.

RQ3 (Coverage): Does ACCESS supported with ACME, cover the types of heterogeneous models that are produced throughout the development of safety–critical systems to achieve the highest degree of automation?

RQ4 (Scalability): Does ACCESS supported with ACME, support the development of complex safety–critical systems with a large number of model elements?

As the focus of our evaluation is the ACCESS methodology (primary) and its tool support (secondary), we necessarily made a number of assumptions. In particular, we assumed that the development activities identified in the ACCCESS methodology are (mostly) model based, which produce structured models that can be processed by model management frameworks. Secondly, we assumed that ACCESS could be used to construct assurance cases for all aspects of the target system, including their design, development, operation and maintenance. We further assumed that the runtime assurance case converted from development time assurance case, would be used by the target system as an non-invasive service (i.e. the runtime assurance case does not interact directly with the decision-making components of the target system), the target system would consult the runtime evaluation results of the assurance case and adapt accordingly.

Table 2
Comparative experiment for efficiency evaluation.

Subsystem	Participant	No. Elements in AC	WL1 (min)
A	A	102	505*
A	B	98	262
A	C	110	87
B	A	231	1143*
B	B	227	502
B	C	246	105

The experiments carried out to address the above research questions are described in Sections 6.1–6.4.

6.1. RQ1: Efficiency

For efficiency, we conducted comparative experiments on subcomponents of System A and System B due do their complexities. The evaluation set-up is as follows;

1. Evaluation subjects. For System A, we selected a power supply unit (Subsystem A) as the experiment subject. For System B, we selected a navigation unit (Subsystem B) which receives sensor data and plans the route for the system.
2. Participants. We asked three safety–critical systems engineer (with relatively same level of expertise) to participate in the experiments.

The experiment procedure is as follows. We asked Participant A to engineer Subsystem A and Subsystem B by following ACCESS with a complete manual approach with no model-based tool support (i.e. all artifacts produced are text-based documents). We then asked Participant B to engineer Subsystem A and Subsystem B (by following ACCESS) with an model-based approach but without the support from ACME, Participant B chose to use EMF (Steinberg et al., 2008) and Robochart (Miyazawa et al., 2019) as the modelling platform and chose Eclipse Epsilon (Kolovos et al., 2008) as the model management framework. We then asked Participant C to engineer Subsystem A and Subsystem B (by following ACCESS) with the tool support from ACME. We recorded the effort it took for all three participants to engineer Subsystem A and Subsystem B in terms of time. The experiment results are shown in Table 2.

In the experiment for Subsystem A, Participant A required approximately 505 min, and produced an assurance case with 102 elements; Participant B required approximately 262 min, and produced an assurance case with 98 elements; Participant C required approximately 87 min, and produced an assurance case with 110 elements. It is to be noted that the difference in number of elements in the assurance case is because that creating an assurance case is highly subjective, hence the difference in numbers were expected. It is also to be noted that since Participant A took a complete manual process, the assurance case produced was NOT a model-based one, therefore it could NOT be converted to runtime assurance case and could NOT be evaluated against runtime data. Therefore, the time taken for Participant A shown in Table 2 only reflected time taken following ACCESS steps 1 to 5.

The time breakdown for participants to engineer Subsystem A are:

- Participant A: 45 min for Step 1; 50 min for Step 2; 30 min for Step 3; 180 min for Step 4; 200 min for Step 5; N/A for Step 6 and 7.

- Participant B: 47 min for Step 1; 20 min for Step 2; 19 min for Step 3; 43 min for Step 4; 56 min for Step 5; 30 min for Step 6; and 47 min for Step 7.
- Participant C: 40 min for Step 1; 20 min for Step 2; 19 min for Step 3; 1 min for Step 4; 3 min for Step 5; 2 min for Step 6; and 2 min for Step 7.

We also observed that the time it took to follow ACCESS Step 1 for all participants are in the same order of magnitude, this was due to the fact that Participants B and C chose to create modelling languages for their system definition, system requirements and safety concept. Although it took them about the same time as Participant A, we argue that the creation of modelling languages is a one-off effort — in subsequent experiments, the modelling languages are re-used, further reducing the time taken.

We do not discuss the time taken to engineer Subsystem B in detail, except that Participant B and C took significantly less time in ACCESS Steps 1 and 2 due to the reuse of modelling languages they created in engineering Subsystem A. We could therefore draw the conclusion that by adopting ACCESS with model-based support, it improves development efficiency comparing to manual effort, even more so when ACME support is available.

6.2. RQ2: Generality

As previously mentioned, we used ACCESS to develop the AUV and a safety–critical autonomous robotic system. The model-based tool support for ACCESS (either it being ACME or other model-based assurance case management environment) is underpinned by two fundamental aspects of Model Driven Engineering:

- Domain Specific Modelling, which allows experts in different domains to create modelling languages that best describe their applications;
- Model Management Operations, which enable model transformations and model validation in an automated manner.

In addition, throughout the development process using ACCESS with model-based support, we found that it was often necessary to “link” models defined using different modelling technologies. In our work, this is achieved by exploiting the facilities provided by the Structured Assurance Case Metamodel (SACM). However, other model-based approaches, for example, the use of a *weaving model* (Hawkins et al., 2015) may also be adopted to realise the links among heterogeneous models.

In addition, we would like to point out that, some components of System B were developed in conformance to ISO 26262 as SEooCs (Safety Element out of Context) (International Organization for Standardization (ISO), 2011) following the development process defined in ISO 26262. From our experience the ACCCESS methodology integrated seamlessly with the development of SEooCs which would be certified against ISO 26262.

Hence, we draw the conclusion that the ACCESS methodology is a generic approach across different domains in the context of safety–critical systems, as long as an assurance case is required for the certification of the target system.

6.3. RQ3: Coverage

During the development of systems following ACCESS, we needed to deal with models defined using different tools and technologies. The types of models included:

- Model defined using Eclipse Modelling Framework (EMF), EMF models are most commonly seen as EMF is the de-facto modelling framework for most open source tools, and are supported by most open source modelling platforms. We also used EMF to model our system requirements, safety concepts, system architectures, etc.;
- Simulink models, Simulink is a modelling environment under MATLAB, it provides a graphical block-based modelling framework that supports the design, simulation and analysis of systems; we made use of Simulink models to design some of the schematics of our EE systems;
- Excel spreadsheets, spreadsheets are typically used to store Failure Mode and Effect Analysis (FMEA), which is an inductive safety analysis method used in identifying failure modes of components and their effects in the system;
- UML models, standardised by the Object Management Group, most UML tools (commercial or open-source) uses XMI (XML Metadata Interchange) format to store models;
- Formal models, in our work, Isabelle models are used to verify software behaviour;
- JSON models, structured models with no metadata for rapid modelling.

In our work, due to the fact that we adopted Eclipse Epsilon as the supporting model management framework, we are able to support the above model formats by either using Epsilon’s Model Connectivity (EMC) layer, or directly using Epsilon’s existing model driver. If there are new types of models defined using new modelling technology, we may also support it by extending the EMC by creating a dedicated model driver for it. We therefore draw the conclusion that ACCESS supported by ACME achieve a high degree of coverage in managing heterogeneous models.

Table 3
Normalised efficiency experiment.

Model	No. of model elements	Time taken for evaluation (s)
Set0	109	0.1
Set1	269	0.2
Set2	1369	0.8
Set3	5689	4.1
Set4	5 689 000	48.3
Set5	568 990 000	N/A

6.4. RQ4: Scalability

Our last evaluation is on the scalability of ACME, although ACME is the secondary contribution of our work, we report our findings in the scalability of the tools. Our evaluation was performed on the premise that the majority of the models used in our development process are EMF models, with a mixture of Simulink models, Excel spreadsheets, formal models and JSON models, which are identified in Section 6.2. To evaluate the scalability of ACME, we selected 5 data sets as shown in Table 3. It is to be noted that the in our end result systems, the maximum number of model elements we have in our collection of models was 5689 (Set3). We made duplicates of our models and put them together to form Set4 and Set5 to evaluate the scalability of ACME in different order of magnitudes. We found that ACME suffered from scalability issues from Set4 and would not load Set5 due to memory overflow. This is typically caused by the fact that ACME need to load EMF models in their entirety before any queries can be performed on them, which is an existing issue discovered in various studies (Barmpis and Kolovos, 2013; Wei et al., 2016; Shah et al., 2014).

7. Related work

There are a number of assurance case works and tools that promote automation by adopting MDE, such as AdvoCATE (Denney and Pai, 2017), D-Case Editor (Matsuno et al., 2010), ASCE (Netkachova et al., 2014), Astah GSN (Larrucea et al., 2017), and CertWare (Barry, 2011).

MDE is applied from different perspectives of Assurance Case process including

1. to generate AC following the process of "predefined pattern" and "pattern instantiation". The pattern can be modelled by extending the syntax of graphical notations, and the relationship between system data and AC elements can be modelled for automatic instantiation.
2. to verify by formal verification for evidence generation.
3. to check the correctness of AC structure by structure modelling.

NASA has developed a powerful graphical tool AdvoCATE (Denney and Pai, 2017) based on Eclipse EMF for AC generation, management, and evaluation. MDE is applied in two aspects; GSN is extended with a formal syntax to support the syntactic checks; and the evidence is generated by exploiting FM; it also uses a formal foundation for a lightweight semantic checking based on the metadata attached to the nodes though it is not to encode the whole argument in a formal machine-checkable language. The work follows the process of pattern design, instantiation, and formal verification. The AC pattern designed is generic without specific application constraints to the systems, and is split into two levels, from hazard mitigation to safety requirements, and from safety requirements down to the evidence. The claims at the second level, i.e. the safety requirements, are formalised manually, and verified by invoking the AUTOCERT tool (Denney and Trac, 2008). The instantiation requires engineers to identify the logic relationship among system data and between AC nodes and system data. The mapping is represented as a table to facilitate the automatic instantiation.

AUTOCERT invoked by AdvoCATE further invokes an automatic theorem prover to verify that the code satisfies the safety properties. However, no means for evaluating the referenced engineering artifacts has been provided. During operation, the change of the system design and the invalidation of assumptions, etc. can be identified, then ACs are automatically instantiated with updated system data for evolution. However, the AC update process will be partially manual if the safety claim is changed which requires the manual formalisation.

Hawkins et al. proposed a model-based approach for generating Assurance Cases (Hawkins et al., 2015). Model-based GSN assurance case patterns are used as a basis. The work exploits the concept of weaving models (Del Fabro et al., 2006) that represent the links between metamodels. With the weaving models, the approach allows the instantiation of GSN patterns, which automatically instantiate *weaves* (by means of model transformation) system information into an assurance case based on the links in the weaving models. The links between assurance case elements and system data can be updated automatically when the system design changes because the links are built between the metamodels instead of specific system data. However, apart from injecting system information into an assurance case, no traceability support from an assurance case to engineering artifacts is provided. Consequently, the validation and verification of assurance case is not covered.

Lin et al. (2016) is also based on a pre-defined pattern and pre-organised data. The software development process is modelled. The process activities' results are software artifacts whose relationships are shown through the input and output of processes. For example, the contribution of software to the hazard derives the software safety requirements. Thus, the relationships of different classes of data can be extracted from the development process models. The pattern is designed specifically for software safety with consideration of software contribution to the hazard. The work adds the syntax of GSN to allow automatic instantiation of the software artifacts. With the help of a relationship model, the impact of artifact change, e.g. the change of the software contribution to a hazard, can be identified automatically in a AC structure for a convenient review. However, since the software is not required to be developed with MDE, the automatic verification of AC is not considered in this work.

Utsunomiya et al. (2018) developed a tool for constructing assurance cases by reading architecture models, quality properties, and risk measurements written in Extensible Markup Language (XML). Assurance cases are generated which conform to SACM v1.0 XML Metadata Interchange schema definition. The tool does not cover the evidence generation of assurance cases. The effectiveness of the method has been insufficiently evaluated to show whether it can be generalised.

Prokhorova et al. (2015) provide a solution for formalising subclaims and verifying with formal method tools. The system properties represented by sub-claims are categorised into eight classes such as temporal properties, timing properties, etc. Patterns are established for each class with different formal method verification solutions. The combination of verification tools shows the necessity of multiple formal methods, also referred to as integrated formal methods (Paige, 1997).

Gleirscher et al. (2019a) proposed to model assurance cases formally for autonomous robot systems following the pattern and instantiation process. The purpose is to cope with the AC evolution during the system development instead of the operation. Two assurance case patterns are designed, that cover the construction pattern for the system specification phase, and the extension pattern for the system implementation/refinement phase. Both are instantiated with the system models. Since the system is required to be modelled in a formal language, there is no need for a process of formalisation of system models, and formal method is a natural choice for assurance case verification. However, the two-phase patterns are specific to the assumption/guarantee (A/G)-style reasoning, therefore not suitable for other RAS systems that do not follow A/G-style reasoning.

Calinescu et al. (2018) proposes the dynamic verification of self-adaptive systems at runtime, but no tool support has been provided. The work proposed the assurance case pattern for self-adaptive systems which is instantiated with formalised system requirements, and further verified by model checkers, which facilitates the automatic co-evolution of system and assurance case by avoiding the manual process of system model formalisation.

Gacek et al. (2014) proposed a new domain-specific language, Resolute, which is also a tool for building assurance cases based on AADL (AS5506, 2004) models. The generation of assurance cases consists of two steps. Firstly, the top-level claim is defined formally by the engineers in first-order predicates where AADL models are queried. Secondly, the engineers decompose the top-level claim quantitatively. An example could be that the sum of message delay of threads is bounded while the top-level claim is the message delay of the process is bounded. Then the AADL model will be queried through the assurance case and checked automatically by model checking. The claims and rules for two steps are recorded in the Resolute library for reuse. The way to integrate Resolute assurance case with AADL model is to add a "prove" statement within AADL models. As the assurance case is written in the same developing environment of AADL models, system model to support assurance case claims will be queried by Resolute models, not transformed or instantiated. So the Resolute assurance case is directly integrated into the system architecture model. This integration enables the traceability and consistency between the system model and its assurance cases, and facilitates the automatic co-evolution of system and assurance case. But this integration on the other hand limits the solution application to other system modelling languages. Since ACME is based on the EMF and the Epsilon framework, it is possible to support various modelling languages (which is demonstrated in Section 5). It is noted that Resolute can only be applied to the system architecture level, rather than the implementation, otherwise it will incur state explosion. Also, the claim of the use case is security properties, where the architectural data are sufficient as assurance case inputs. However, it will require more data (e.g. hazard models, safety-related function models) for safety property argumentation; and Resolute may not be capable of describing those properties properly.

Rushby conceived of an evidential tool bus (Rushby, 2005) that would allow integration of various verification tools to provide evidence to an assurance case, an idea that was later realised by Cruanes et al. (2013). Isabelle is also an evidential tool bus, and its connection to ACME allows linking to formal evidence.

8. Summary and future work

In this work, we present ACCESS, a development methodology that promotes the development of safety–critical systems around an evolving model-based assurance case. We present ACME, along with ACCESS, and illustrate how ACCESS steps can be followed to develop a critical system from the beginning.

ACME is an integrated model-based assurance case development tool, that supports fine-grained traceability to engineering artifacts such as EMF models, Excel spreadsheets and Isabelle theory files. For model-based engineering artifacts, ACME provides the support that enables the users to attach model queries to SACM elements, which can be automatically executed to validate the engineering artifacts. For external Isabelle theory files, we provide support which makes use of an Isabelle server to process the theory files and reflect the result back to ACME. In this case, assurance cases can be automatically evaluated, which includes the traceability and validation of engineering artifacts, that significantly reduces development time and improves development efficiency, comparing to human efforts. In addition, assurance cases can co-evolve with system development, for changes in the system engineering artifacts can be validated by ACME and problems can be found rapidly, to allow a rapid turn-around between system development and system assurance. We also showed how a development time assurance case can be converted to a dynamic assurance case, with traceability to runtime data. With the help of *Dynamic Safety Management System* and *Runtime Data Drivers*, we are able to monitor the validity of the dynamic assurance case in a real time manner.

It is to be noted that we could not show activities and outcomes in each ACCESS step entirely in this paper, due to space limitation. As such, in the future we would continuously improve the ACCESS methodology and perhaps publish guidelines based on ACCESS to explain the assurance case centric development process in greater details.

With regard to ACME, we have shown that we can trace engineering artifacts such as EMF models, Excel spreadsheets and Isabelle theory files. In the future, we plan to support models defined in other modelling technologies such as Simulink and PTC Integrity Modeller. Currently, ACME only support EOL for model validation rules, which practitioners may not be acquainted with. In the future, we plan to lower the technical barrier by providing support to *Constrained Natural Language* as query language, so that it requires minimal effort to learn. Also, the impact of system engineering artifacts to assurance case currently requires the practitioners to evaluate the assurance case to detect invalidities. In the future we plan to provide support for passive impact analysis with a version control system, so that we can support version control on both assurance cases and engineering artifacts. Finally, we plan to support SACM's argument notation alongside with GSN notation, to prompt the wider adoption of SACM.

CRediT authorship contribution statement

Ran Wei: Assurance case model development, ACME development. **Simon Foster:** Formal specification development, Software behavioural modelling. **Haitao Mei:** Runtime assurance case conversion, FMEDA example contribution. **Fang Yan:** Robochart case study creator. **Ruizhe Yang:** Transformation rules from Simulink models to EMF models. **Ibrahim Habli:** Related work. **Colin O'Halloran:** Related work, AUV use case provider. **Nick Tudor:** AUV use case provider. **Tim Kelly:** Advisor, Assurance case model development. **Yakoub Nemouchi:** Coding.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The data that has been used is confidential.

Acknowledgments

Simon Foster's contributions are funded by the EPSRC-UKRI project CyPhyAssure (https://www.cs.york.ac.uk/circus/CyPhyAssure/), grant reference EP/S001190/1. Fang Yan's contributions are funded by the European Union's Horizon 2020 research and innovation programme under the Marie SkĺodowskaCurie grant agreement No 812.788 (MSCA-ETN SAS).

References

Alkassar, E., Hillebrand, M., Leinenbach, D., Schirmer, N., Starostin, A., 2008. The Verisoft approach to systems verification. In: VSTTE 2008. In: LNCS, vol. 5295, Springer, pp. 209–224.

AS5506, S., 2004. Architecture analysis and design language (AADL). Embed. Comput. Syst. Comm. SAE.

Barmpis, K., Kolovos, D., 2013. Hawk: Towards a scalable model indexing architecture. In: Proceedings of the Workshop on Scalability in Model Driven Engineering. pp. 1–9.

Barry, M.R., 2011. CertWare: A workbench for safety case production and analysis. In: Aerospace Conference, 2011 IEEE. IEEE, pp. 1–10.

Bishop, P., Bloomfield, R., 2000. A methodology for safety case development. In: Safety and Reliability. volume 20, Taylor & Francis, pp. 34–42.

Blanchette, J.C., Bulwahn, L., Nipkow, T., 2011. Automatic proof and disproof in Isabelle/HOL. In: FroCoS. In: LNCS, vol. 6989, Springer, pp. 12–27.

Brambilla, M., Cabot, J., Wimmer, M., 2017. Model-driven software engineering in practice. Synth. Lect. Softw. Eng. 3 (1), 1–207.

Brookes, S.D., Hoare, C.A.R., Roscoe, A.W., 1984. A theory of communicating sequential processes. J. ACM 31 (3), 560–599.

Brucker, A., Wolff, B., 2019. Using ontologies in formal developments targeting certification. In: Integrated Formal Methods. IFM, In: LNCS, vol. 11918, Springer, pp. 65–82.

Calinescu, R., Weyns, D., Gerasimou, S., Iftikhar, M.U., Habli, I., Kelly, T., 2018. Engineering trustworthy self-adaptive software with dynamic assurance cases. IEEE Trans. Softw. Eng. 44 (11), 1039–1069.

Cruanes, S., Hamon, G., Owre, S., Shankar, N., 2013. Tool integration with the evidential tool bus. In: VMCAI. In: LNCS, vol. 7737, Springer, pp. 275–294.

Del Fabro, M.D., Bézivin, J., Valduriez, P., 2006. Weaving models with the eclipse AMW plugin. In: Eclipse Modeling Symposium, Eclipse Summit Europe. volume 2006, pp. 37–44.

Denney, E., Pai, G., 2017. Tool support for assurance case development. Autom. Softw. Eng. 1–65.

Denney, E., Pai, G., Habli, I., 2015. Dynamic safety cases for through-life safety assurance. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. volume 2, IEEE, pp. 587–590.

Denney, E., Trac, S., 2008. A software safety certification tool for automatically generated guidance, navigation and control code. IEEE Aerosp. Conf. Proc..

Eclipse Foundation, 2003. Eclipse modelling framework (GMF). https://www.eclipse.org/modeling/gmp/.

European Organisation for the Safety of Air Navigation (EUROCONTROL), 2006. Safety case development manual.

Foster, S., Baxter, J., Cavalcanti, A., Miyazawa, A., Woodcock, J., 2018. Automating verification of state machines with reactive designs and Isabelle/UTP. In: Proc. 15th. Intl. Conf. on Formal Aspects of Component Software. In: LNCS, vol. 11222, Springer.

Foster, S., Nemouchi, Y., Gleirscher, M., Kelly, T., 2019. Isabelle/SACM: Computer-assisted assurance cases with integrated formal methods. In: IFM. In: LNCS 11918, Springer, pp. 379–398.

Foster, S., Nemouchi, Y., Gleirscher, M., Wei, R., Kelly, T., 2021. Integration of formal proof into unified assurance cases with Isabelle/SACM. Form. Asp. Comput..

Foster, S.D., Nemouchi, Y., O'Halloran, C., Tudor, N., Stephenson, K., 2020. Formal model-based assurance cases in Isabelle/SACM: An autonomous underwater vehicle case study. In: Formal Methods in Software Engineering (FormaliSE 2020): Proceedings of the 8th International Conference. ACM.

Gacek, A., Backes, J., Cofer, D., Slind, K., Whalen, M., 2014. Resolute: an assurance case language for architecture models. ACM SIGAda Ada Lett. 34 (3), 19–28.

Gleirscher, M., Foster, S., Nemouchi, Y., 2019a. Evolution of formal model-based assurance cases for autonomous robots. In: International Conference on Software Engineering and Formal Methods. Springer, pp. 87–104.

Gleirscher, M., Foster, S., Woodcock, J., 2019b. New opportunities for integrated formal methods. ACM Comput. Surv. 52 (6).

Greenwell, W.S., Knight, J.C., Holloway, C.M., Pease, J.J., 2006. A taxonomy of fallacies in system safety arguments.

Habli, I., White, S., Sujan, M., Harrison, S., Ugarte, M., 2018. What is the safety case for health IT? A study of assurance practices in England. Saf. Sci. 110, 324–335.

Hawkins, R., Habli, I., Kolovos, D., Paige, R., Kelly, T., 2015. Weaving an assurance case from design: a model-based approach. In: High Assurance Systems Engineering (HASE), 2015 IEEE 16th International Symposium on. IEEE, pp. 110–117.

International Atomic Energy Agency (IAEA), 2008. IAEA safety glossary: Terminology used in nuclear safety and radiation protection.

International Organization for Standardization (ISO), 2011. ISO 26262: Road vehicles - functional safety.

Jaaksi, A., 2002. Developing mobile browsers in a product line. IEEE Softw. 19 (4), 73–80.

Kärnä, J., Tolvanen, J.-P., Kelly, S., 2009. Evaluating the use of domain-specific modeling in practice. In: Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling.

Kelly, T.P., 1999. Arguing Safety: a Systematic Approach to Managing Safety Cases (Ph.D. thesis). University of York York, UK.

Kelly, T., Weaver, R., 2004. The goal structuring notation–a safety argument notation. In: Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases. Citeseer, p. 6.

Kolovos, D.S., Paige, R.F., Polack, F.A., 2006. The epsilon object language (EOL). In: European Conference on Model Driven Architecture-Foundations and Applications. Springer, pp. 128–142.

Kolovos, D.S., Paige, R.F., Polack, F.A., 2008. The epsilon transformation language. In: International Conference on Theory and Practice of Model Transformations. Springer, pp. 46–60.

Larrucea, X., Walker, A., Colomo-Palacios, R., 2017. Supporting the management of reusable automotive software. IEEE Softw. (3), 40–47.

Lee, E.A., Sirjani, M., 2018. What good are models? In: FACS. In: LNCS, vol. 11222, Springer.

Lin, C.-L., Shen, W., Drager, S., 2016. A framework to support generation and maintenance of an assurance case. In: 2016 IEEE International Symposium on Software Reliability Engineering Workshops. ISSREW, IEEE, pp. 21–24.

Machin, M., Guiochet, J., Waeselynck, H., Blanquart, J.-P., Roy, M., Masson, L., 2018. SMOF: A safety monitoring framework for autonomous systems. IEEE Trans. Syst. Man Cybern. 48 (5).

Mathworks, 2020. Simulink. https://www.mathworks.com/products/simulink.html. Online; accessed 6th June, 2020.

Matsuno, Y., Takamura, H., Ishikawa, Y., 2010. A dependability case editor with pattern library. In: High-Assurance Systems Engineering (HASE), 2010 IEEE 12th International Symposium on. IEEE, pp. 170–171.

McDermid, J.A., 2001. Software safety: where's the evidence? In: Proceedings of the Sixth Australian Workshop on Safety Critical Systems and Software-Volume 3. Australian Computer Society, Inc., pp. 1–6.

Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A., Timmis, J., Woodcock, J., 2019. RoboChart: modelling and verification of the functional behaviour of robotic applications. Softw. Syst. Model..

Nair, S., de la Vara, J.L., Sabetzadeh, M., Falessi, D., 2015. Evidence management for compliance of critical systems with safety standards: A survey on the state of practice. Inf. Softw. Technol. 60, 1–15.

Netkachova, K., Netkachov, O., Bloomfield, R., 2014. Tool support for assurance case building blocks. In: International Conference on Computer Safety, Reliability, and Security. Springer, pp. 62–71.

Nipkow, T., Klein, G., 2014. Concrete Semantics with Isabelle/HOL. Springer.

Nipkow, T., Wenzel, M., Paulson, L.C., 2002. Isabelle/HOL: A proof assistant for higher-order logic. In: LNCS, vol. 2283, Springer.

Object Management Group, 2020. Structured assurance case metamodel. https://www.omg.org/spec/SACM. Online; accessed 6th June, 2020.

Paige, R.F., 1997. A meta-method for formal method integration. In: International Symposium of Formal Methods Europe. Springer, pp. 473–494.

Prokhorova, Y., Laibinis, L., Troubitsyna, E., 2015. Facilitating construction of safety cases from formal models in event-B. Inf. Softw. Technol. 60, 51–76.

Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A., 2008. The epsilon generation language. In: European Conference on Model Driven Architecture-Foundations and Applications. Springer, pp. 1–16.

Rushby, J., 2005. An evidential tool bus. In: Formal Methods and Software Engineering. ICFEM, In: LNCS, vol. 3785, Springer.

Shah, S.M., Wei, R., Kolovos, D.S., Rose, L.M., Paige, R.F., Barmpis, K., 2014. A framework to benchmark NoSQL data stores for large-scale model persistence. In: International Conference on Model Driven Engineering Languages and Systems. Springer, pp. 586–601.

Steinberg, D., Budinsky, F., Merks, E., Paternostro, M., 2008. EMF: Eclipse Modeling Framework. Pearson Education.

Trapp, M., Schneider, D., Liggesmeyer, P., 2013. A safety roadmap to cyber-physical systems. In: Perspectives on the Future of Software Engineering. Springer, pp. 81–94.

Tuong, F., Wolff, B., 2019. Deeply integrating C11 code support into Isabelle/PIDE. In: Formal Integrated Development Environment (F-IDE). In: EPTCS, vol. 310, pp. 13–28.

U.K. Ministry of Defence (MOD), 1996. JSP 430 - ship safety management system handbook.

U.K. Ministry of Defence (MOD), 1997. 00-55 Requirements of safety related software in defence equipment.

U.K. Ministry of Defence (MOD), 2007. Safety management requirements for defence systems.

U.K. Rail Safety Standards Board, 2007. Engineering safety management issue 4.

Utsunomiya, H., Kobayashi, N., Morisaki, S., Yamamoto, S., 2018. A tool to create assurance case through models. Trans. Mach. Learn. Artif. Intell. 6 (2), 46.

Wei, R., Jiang, Z., Guo, X., Mei, H., Zolotas, A., Kelly, T., 2022. Designing critical systems with iterative automated safety analysis. In: Proceedings of the 59th ACM/IEEE Design Automation Conference. pp. 181–186.

Wei, R., Jiang, Z., Guo, X., Yang, R., Mei, H., Zolotas, A., Kelly, T., 2023a. DECISIVE: Designing critical systems with iterative automated safety analysis. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst..

Wei, R., Jiang, Z., Mei, H., Barmpis, K., Foster, S., Kelly, T., Zhuang, Y., 2023b. Automated model based assurance case management using constrained natural language. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst..

Wei, R., Kelly, T.P., Dai, X., Zhao, S., Hawkins, R., 2019. Model based system assurance using the structured assurance case metamodel. J. Syst. Softw. 154, 211–233.

Wei, R., Kolovos, D.S., Garcia-Dominguez, A., Barmpis, K., Paige, R.F., 2016. Partial loading of XMI models. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems. pp. 329–339.

Wei, R., Reich, J., Kelly, T., Gerasimou, S., 2018. On the transition from design time to runtime model-based assurance cases. In: 13th International Workshop on Models@Runtime, ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2018).

Wenzel, M., 2019. Interaction with formal mathematical documents in Isabelle/PIDE. In: CICM. In: LNCS 11617, Springer, pp. 1–15.

Wenzel, M., Wolff, B., 2007. Building formal method tools in the Isabelle/Isar framework. In: TPHOLs. In: LNCS, vol. 4732, Springer.

Ran Wei is a Senior Research Fellow in the Department of Engineering of the University of Cambridge. His research interests include model driven engineering, digital twins, model-based systems assurance, high integrity systems engineering, real time systems, automated assurance of robotic and autonomous systems.

Simon Foster is a Lecturer in Computer Science at the University of York in the UK. His interests lie in theorem proving, formal semantics, cyber–physical systems, hybrid systems, process algebra, denotational semantics, algebraic methods, and functional programming.

Haitao Mei is an ACM Member. His research lie in scalable cloud architecture, real time systems, operating systems, high integrity systems engineering and automated assurance of robotic and autonomous systems.

Fang Yan is a researcher in Computer Science at the University of York, UK. Her research interests are in the Model-based Assurance Cases process for Robotic Autonomous Systems. She is currently working specifically on the topic of AC generation and verification technical solutions based on FM.

Ruizhe Yang is a MSc student in Artificial Intelligence of Dalian University of Technology (DUT), China. His current research interests include Model Driven Engineering, High Integrity Systems Engineering and Model-Based Digital Twin.

Ibrahim Habli is a professor in Computer Science at the University of York, UK. His interests lies in the design and assurance of safety–critical systems, with a particular focus on AI systems (e.g. for clinical diagnosis and autonomous and connected driving) and Digital Health (e.g. ePrescribing and self-management apps).

Colin O'Halloran has over 30 years of experience in high integrity systems, operating as a scientist, consultant and business group manager with RSRE, DERA and QinetiQ. He is also a visiting professor at the University of York and was previously a full professor of computer science at the University of Oxford. Colin is an internationally recognised expert in the validation and verification of software-based systems using formal methods.

Nick Tudor has spent nearly two further decades within the aerospace, autonomous systems and automotive industries in various roles in large business and SMEs. Nick is an active supporter of the international effort to refresh DO178 and is a member of the Forum for Aeronautical Software.

Tim Kelly is an honorary professor of Computer Science at the University of York, UK. He was one of the main contributors for the Goal Structuring Notation (GSN) and the Structured Assurance Case Metamodel (SACM), both are international standards used in arguing the safety of high-integrity systems. His research lie in the rigorous justification and development of safety–critical systems; architectural and pattern-oriented approaches to software design and development.

Yakoub Nemouchi is a former postdoctoral researcher at the University of York. His research interests include Formal Verfication and Model Based Testing.