Model Based System Assurance Using the Structured Assurance Case Metamodel

Ran Wei, Tim P. Kelly, and Richard Hawkins

University of York, United Kingdom
{ran.wei,tim.kelly,richard.hawkins}@york.ac.uk

Abstract. Assurance cases are used to demonstrate confidence in system properties of interest (e.g. safety and/or security). A number of system assurance approaches are adopted by industries in the safety-critical domain. However, the task of constructing assurance cases remains a manual, lengthy and informal process. The Structured Assurance Case Metamodel (SACM) is a standard specified by the Object Management Group (OMG). SACM is designed to be used to construct assurance case models, and is more powerful than existing system assurance approaches. However, the intended usage of SACM has not been sufficiently explained. In addition, there has not been support to interoperate between existing assurance case models and SACM models. SACM provides a solid foundation for model-based system assurance, which bears great application potentials in growing technology domains such as Cyber-Physical Systems and Internet of Things.
In this paper, we explain the intended usage of SACM. In addition, to promote model based approach, we provide SACM compliant metamodels for existing system assurance approaches (i.e GSN and CAE), and the transformations from them to SACM. We also briefly discuss the tool support for model-based system assurance which help practitioners make the transition from existing system assurance approaches to model-based system assurance using SACM.

Keywords: Mode Driven Engineering; Structured Assurance Case Metamodel; Model Based System Assurance; Goal Structuring Notation; Claims, Arguments and Evidence

1 Introduction

Systems/services used to perform critical functions require justifications that they exhibit necessary properties (i.e. safety and/or security). *Assurance case*s provide an explicit means for justifying and assessing confidence in these critical properties. In certain industries, typically in the safety-critical domain (e.g. defence, avaiation, automotive and nuclear), it is a regulatory requirement that an assurance case is developed and reviewed as part of the certification process [1]. An assurance case report is a document that facilitates information exchange between various system stakeholders (e.g. between operator and regulator), where the knowledge related to the safety and security of the system is communicated in a clear and defendable way [2].

Assurance case reports are typically represented textually - using natural languages; or graphically - using structured graphical notations such as the Goal Structuring Notation (GSN) [3] or Claims, Arguments and Evidence (CAE) [4]. A number of tools exist which implement GSN anc CAE to produce argumentation diagrams. Some tools adopt model-based approach to produce models that conform to their own versions of GSN and CAE metamodels.

To improve standardisation and interoperability, the Object Management Group (OMG) specified and issued a Structured Assurance Case Metamodel (SACM), which can be used to construct model-based assurance cases. SACM is more powerful than GSN and CAE in the sense that it captures not only the argumentation regarding system properties, but also the inter-relationship between the argumentation and its supporting artefacts. However, neither detailed explanation has been provided in the OMG specification to demonstrate how to use SACM, nor the relationships between existing assurance case approaches (e.g. GSN and CAE) and SACM has been discussed. This brings challenges to the adoption of SACM due to the complex structure of SACM and the sophistication of its intended usage.

Model-based system assurance attracts significant amount of interests in recent years due to the benefits provided by Model-Driven Engineering (MDE) such as automation, consistency and efficiency. With the development of Cyber-Physical Systems (CPS) and Internet of Things (IoT), model based system assurance seems particularly important where automated run-time system assurance is needed (e.g. for collaborative CPSs that form temporary system of systems at runtime). SACM provides a solid foundation for such scenarios.

In this paper, we provide a detailed explanation of SACM and discuss its relationship with existing system assurance approaches. We first discuss the motivation of our work. We then provide detailed discussions about the facilities provided by SACM[1]. We demonstrate how SACM can be used to construct an assurance case using examples, and how to maintain the traceability between the argumentation of assurance cases and their supporting evidence/artefacts. We finally discuss briefly tool support for model-based system assurance, namely the Assurance Case Modelling Environment (ACME).

The contributions of this paper are as follows:

- A definitive exposition of SACM version 2.0;
- A detailed discussion on how to use SACM to create a complete system assurance case.
- GSN and CAE metamodels that are compliant with SACM;
- Comprehensive mappings from GSN/CAE to SACM;

This paper is organised as follows. In Section 2 and Section 3, we provide the motivation and the background of our work. In Section 4 we provide detailed discussions about SACM. We also provide examples on how to use SACM to

[1] Version 2.0 as of April, 2019

construct an assurance case. In Section 6, we discuss the relationship between existing notations and SACM. We provide SACM compliant metamodels for GSN and CAE and their mappings to SACM. In Section 7, we briefly discuss tool support for model-based system assurance. We finally conclude the paper in Section 8.

2 Background and Motivation

2.1 Safety Cases

The concept of assurance cases has been long established in the safety-related domain, where the term *safety case* is normally used. For many industries, the development, review and acceptance of a safety case forms a key element of regulatory processes. This includes the nuclear [5], defence [6], civil aviation [7] and railway [8] industries. Safety cases are defined in [3] as follows: *A safety case should communicate a clear, comprehensible and defensible argument that a system is acceptably safe to operate in a particular context.*

Historically, safety arguments were typically communicated in safety cases through free text. However, there are problems experienced when text is the only medium available for expressing complex arguments. One problem of using free text is that the language used in the text can be unclear and poorly structured, there is no guarantee that system engineers would produce safety cases with clear and well-structured language. Also, the capability of expressing cross-references for free text is very limited, multiple cross-references can also disrupt the flow of the main argument. Most importantly, the problem with using free text is in ensuring that all stakeholders involved share the same understanding of the argument to development, agree and maintain the safety arguments within the safety case [3].

To overcome the problems of expressing safety arguments in free text, graphical argumentation notations have been developed. Graphical argumentation notations are capable of explicitly representing the elements that form a safety argument (i.e. requirements, claims, evidence and context), and the relationships between these elements (i.e. how individual requirements are supported by specific claims, how claims are supported by evidence and the assumed context that is defined for the argument). Amongst the graphical notations, the Goal Structuring Notation (GSN) [3] has been widely accepted and adopted [9]. The key benefit experienced by companies/organisations adopting GSN is that it improves the comprehension of the safety argument amongst all of the key project stakeholders (e.g. system developers, safety engineers, independent assessors and certification authorities), therefore improving the quality of the debate and discussion amongst the stakeholders and reducing the time taken to reach agreements on the argument approaches being adopted.

A number of drawing tools have been developed [10–14] to produce GSN diagrams. Although GSN diagrams produced by the majority of the tools are valuable in communicating safety argumentations amongst stakeholders, these diagrams cannot be consumed and interpreted by computers (e.g. automated validation of safety argumentation, automated generation of safety case reports).

2.2 Safety Cases and Model Driven Engineering

Model-Driven Engineering (MDE) is a contemporary software engineering approach. In MDE, *model*s are first class artefacts, therefore driving the development. There are two important aspects in MDE: domain specific modelling and model management. Domain specific modelling enables domain experts to capture the concepts in their systems in the form of *metamodel*s, which are then used to create models of their systems (that conform to the defined *metamodel*s). Model management enables a series of operations to be performed on models in an automated manner, which include, but not limited to: Model Validation, Model-to-Model Transformation, Model-to-Text Transformation. MDE has been proven to improve productivity significantly due to the automation that model management provides [15, 16].

There is a tendency for tools to adopt MDE, it is no exception for GSN tools. Some GSN tools support exporting GSN diagrams into machine consumable models [13, 14]. However, these tools implement their own versions of the GSN metamodels, which do not consider the links from safety argumentations to their supporting evidence. Therefore, there is little value in performing model management operations on them.

2.3 Assurance Cases and the Structured Assurance Case Metamodel

There has been increasing interest in the use of structured argumentation in other domains, particularly for demonstrating system security [17]. Such argumentations are typically referred to as *security cases*. The similarities between safety and security cases have been highlighted in [18]. Therefore, the term *assurance case* is a broader definition: *An assurance case should communicate a clear, comprehensible and defensible argument that a system/service is acceptably safe and/or secure to operate in a particular context.*

To promote standardisation and interoperability, the Object Management Group (OMG) specified and issued the *Structured Assurance Case Metamodel* [19]. In version 2.0 of SACM[2], a considerable amount of changes have been introduced to promote modularity and openness of SACM. SACM provides a complete solution for model based system assurance case construction. In addition, supporting evidence and related information can also be referenced via reference mechanisms provided in SACM. This makes SACM more powerful than existing techniques such as GSN and CAE.

However, in the SACM specification there is limited information on the intended usage of SACM. To exploit SACM's full potential, and to promote the adoption of SACM, it is necessary to explain SACM in detail so that safety

[2] Released in September, 2017

and security engineers can fully use SACM to achieve higher level goals (e.g. automated model-to-text transformation to generate assurance case reports).

In the current state of practice, graphical notations such as GSN remain the most popular approach for system assurance. SACM is designed to support GSN, but the OMG specification does not provide a mapping between GSN elements and SACM elements. This is due to the fact there has not been a SACM aligned GSN metamodel. Thus, in this paper we provide a GSN metamodel which aligns to SACM. There is also a need to interoperate from GSN to SACM. The reason behind this is two fold. First of all, the OMG has not defined a concrete syntax (i.e. graphical notation) for SACM elementswhich makes creating SACM models a tedious and error-prone process. Thus, to make the transition from GSN to SACM, it is a good practice to use GSN notations to construct arguments and then transform from to SACM using model-to-model transformation. Secondly, since GSN has been widely adopted in industry, practitioners can convert their legacy diagrams into GSN models, and then transform to SACM to enable model-based system assurance. Due the reasons above, in this paper, we will provide a mapping (model-to-model transformation) from GSN to SACM.

2.4 Model-based System Assurance and Technology Trends

In recent years, the term Cyber-Physical Systems (CPS) has emerged to characterise a new generation of embedded systems. CPSs are open and adaptive systems, in the sense that they dynamically interconnect with each other (and other systems) at runtime and they are able to dynamically adapt to changing contexts.

CPS has attracted a significant amount of research interest in recent years for its enormous potential for economic and societal impact in domains such as mobility, home automation and health care [20]. The open and adaptive nature of CPS poses a significant new challenge in assuring dependability. For example, CPSs typically collaborate with each other by temporarily forming a network of CPSs. It is necessary to ensure the dependability of the temporary network of CPS. In [21], the authors identify the importance of system assurance at runtime for CPS and propose the idea of Models@Runtime and Assurance Case@Runtime, in the sense that system assurance information is exchanged when CPSs interconnect with each other to reason the dependability of the to-be-formed system of systems.

We envision that SACM is the best candidate for AssuranceCase@Runtime in this context, for its extensive support for argumentation and the ability to link to supporting evidence within the model. SACM also provides multi-language support, in the sense that argumentations can be expressed using computer languages, which is the premise for automated argument reasoning. This is particularly beneficial for CPSs. This scenario is also applicable to Internet of Things (IoT) where systems that collaborate with each other are connected to the Internet (where security is also a concern).

Therefore, model-based assurance cases are more than 'nice diagrams', they are important artefacts which can be used at design time for system integration, as well as at runtime for automated reasoning when systems collaborate with each other and form temporary networks.

3 The Goal Structuring Notation

It is necessary to discuss the Goal Structuring Notation (GSN) before discussing SACM as SACM was developed based on the concepts in GSN. GSN is a well established graphical argumentation notation to represent safety arguments in a structured way. GSN captures elements that form safety argument, and the relationships between the elements. GSN is widely adopted within safety-critical industries for the presentation of safety arguments within safety cases. The core elements of GSN are shown in Figure 1.

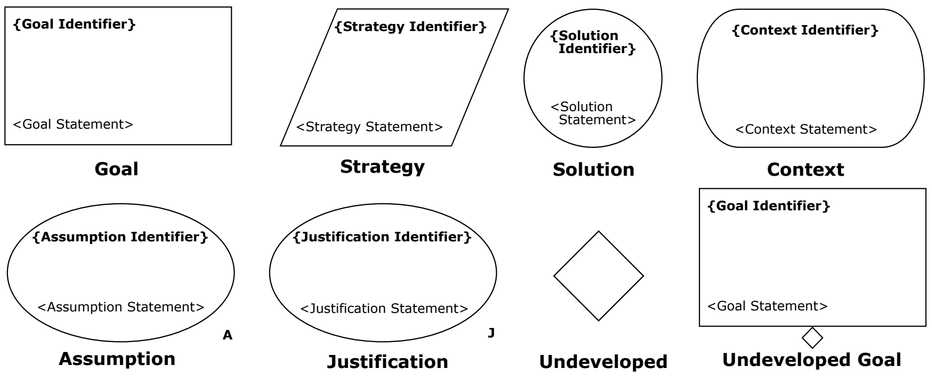


Fig. 1: Core GSN elements.

A *Goal* represents a 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 artefact, 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, it can apply to *Goal*s and *Strategies*, the *Undeveloped Goal* in Figure 1 provides an example.

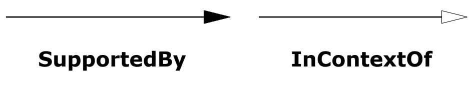
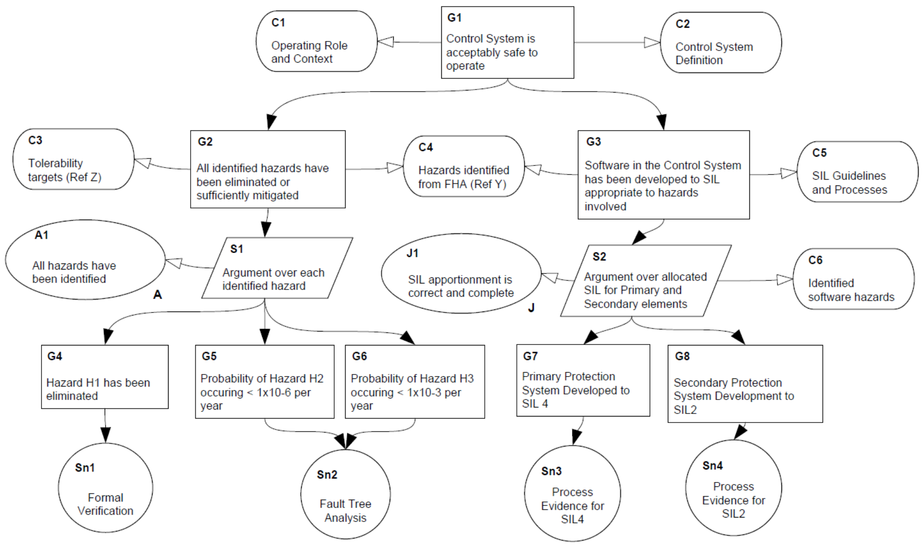


Fig. 2: GSN connectors.

Core elements of GSN are connected with two types of connectors, as shown in Figure 2. The *SupportedBy* edge allows inferential or evidential relationships to be documented. The *InContextOf* relates contextual elements (i.e. *Context*, *Assumption* and *Justification*) to *Goals* and *Strategies*.

When the elements of the GSN are linked together in a network, they are described as a *goal structure*. The purpose of any goal structure is to show how *Goals* are successively broken down into sub-*Goals* until a point is reached where claims can be supported by direct reference to available evidence (*Solutions*). An example goal structure is shown in Figure 3.



G1 Control System is acceptably safe to operate
C1 Operating Role and Context
C2 Control System Definition
G2 All identified hazards have been eliminated or sufficiently mitigated
C3 Tolerability targets (Ref Z)
C4 Hazards identified from FHA (Ref Y)
G3 Software in the Control System has been developed to SIL appropriate to hazards involved
C5 SIL Guidelines and Processes
A1 All hazards have been identified
A
S1 Argument over each identified hazard
J1 SIL apportionment is correct and complete
J
S2 Argument over allocated SIL for Primary and Secondary elements
C6 Identified software hazards
G4 Hazard H1 has been eliminated
G5 Probability of Hazard H2 occuring < 1x10-6 per year
G6 Probability of Hazard H3 occuring < 1x10-3 per year
G7 Primary Protection System Developed to SIL 4
G8 Secondary Protection System Development to SIL2
Sn1 Formal Verification
Sn2 Fault Tree Analysis
Sn3 Process Evidence for SIL4
Sn4 Process Evidence for SIL2

Fig. 3: An example goal structure.

Figure 4 shows how GSN supports modularity. Goal structures can be organised in *Modules*. For example, for a system that consists two components A and B, it is possible to organise the safety case of component A in Module A and safety case of component B in Module B. Modularity promotes re-use, so that safety cases for system components can be re-used when different components integrate to form a system. When integrating system safety case, a *Contract Module* can be used to *bind* different *Modules* together. Binding is done via the use of *Away Goals*, *Away Contexts* and *Away Solution*, where *Goals*, *Contexts* and *Solutions* from an external *Module* can be referenced. Like other GSN elements, away elements can be connected using *SupportedBy* and *InContextOf* edges.

GSN also provides the extension for users to define *GSN Patterns* to re-use good practice of safety cases. In [22], the use of patterns are discussed, patterns are a means of documenting and reusing successful assurance argument struc-

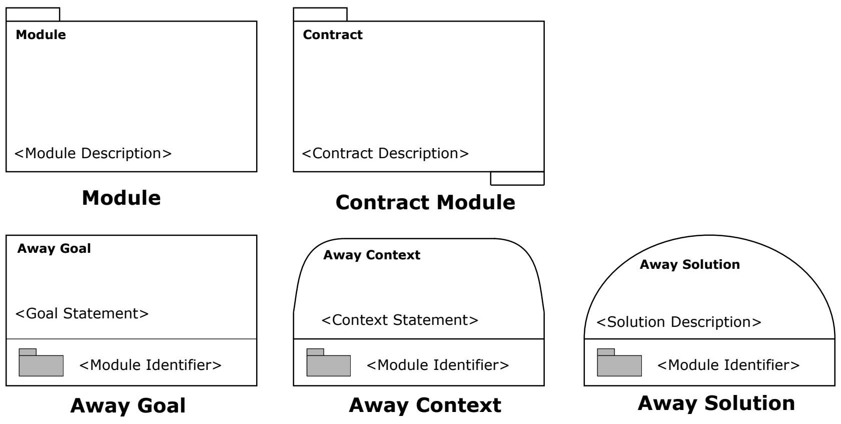


Fig. 4: Modular GSN elements.

tures. Safety case argument patterns provide a way of capturing the required form of a safety argument in a manner that is abstract from the details of a particular argument. It is then possible to use the patterns to create specific arguments by *instantiating* the patterns in a manner appropriate to the application. Pattern instantiation refers to the process of constructing concrete system safety cases by filling in the templates provided in the GSN pattern with actual system information. Figure 5 shows the elements in GSN which enables the users to create patterns.

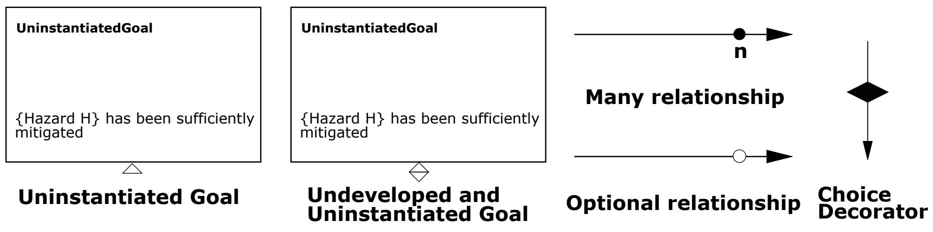

UninstantiatedGoal
{Hazard H} has been sufficiently mitigated
Uninstantiated Goal
UninstantiatedGoal
{Hazard H} has been sufficiently mitigated
Undeveloped and Uninstantiated Goal
n
Many relationship
Optional relationship
Choice Decorator

Fig. 5: GSN pattern elements.

The *Uninstantiated* indicator marks that an element is abstract and to be instantiated, at some later stage, the abstract element needs to be replaced with a more concrete instance. *Uninstantiated indicator* can be associated to any GSN element, Figure 5 demonstrates how it can be associated to an *Goal*. The *Undeveloped and Uninstantiated* indicator marks an element (in particular, *Goal*s and *Strategies*) is both abstract (to be instantiated) and needs more development (needs supporting argument), Figure 5 demonstrates its usage on a Goal. In GSN patterns, the *SupportedBy* and *InContextOf* connector can bear more information, the *Many* decorator on a connector indicates that when the pattern is instantiated, the connector can be multiplied n times (expressed in the label) based on the actual system information provided. The *Optional* decorator indicates that when the pattern is instantiated, the connector can connect to one or zero element. The *Choice* decorator on a connector can be used to denote possible alternatives in satisfying a relationship. It can represent 1-of-n and m-of-n selection.

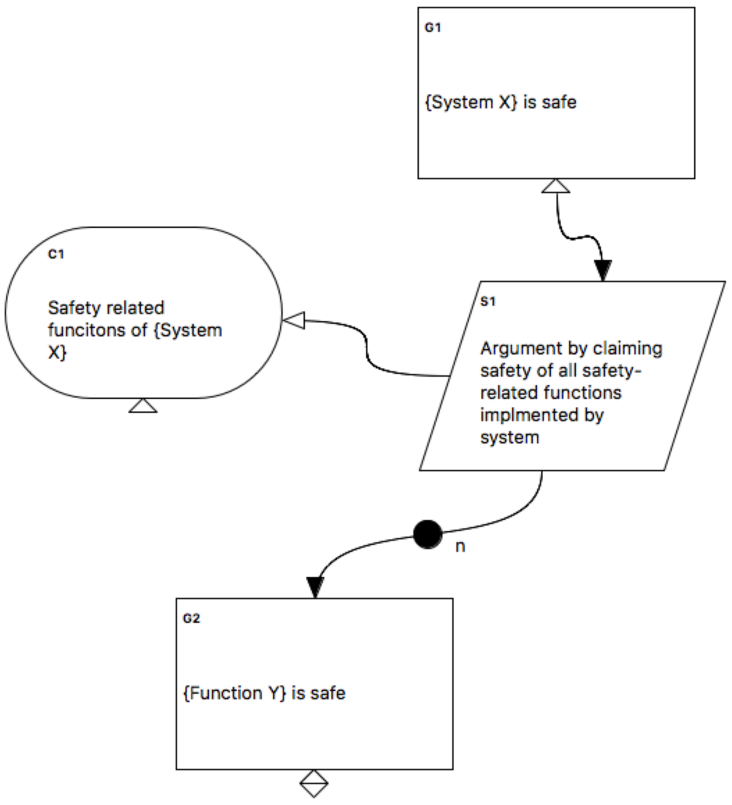

G1
{System X} is safe
C1
Safety related funcitons of {System X}
S1
Argument by claiming safety of all safety-related functions implmented by system
n
G2
{Function Y} is safe

Fig. 6: An example of GSN Pattern [22]

Figure 6 shows an example of GSN pattern (adopted from [22]). The contents in the curly brackets are *role*s in GSN terms, they are place holders which when the pattern is instantiated, will be replaced by actual system information. {System X} in *G1* will be replaced with the actual name of the system when the pattern is instantiated. Similarly, the *SupportedBy* connector between *S1* and *G2* specifies that when the pattern is instantiated, there should be n (the number of safety-related functions implemented by the system) *Goal*s and n *SupportedBy* connected to *S1*. Pattern instantiation if often an manual process that involves comprehension of the GSN patterns and replacing the *role*s with actual system information. However, there has been work on automating the pattern instantiation process using MDE with the use of a *weaving model* [23].

In this section, we discussed briefly the elements provided by GSN. GSN is powerful in representing arguments in a structured way, which enables better comprehension of system safety arguments. GSN promotes modularity and abstraction in the sense that good practice in safety case construction can be captured and re-used. In the following sections, we will discuss SACM and its relationship with GSN.

4 Structured Assurance Case Metamodel

The *Structured Assurance Case Metamodel* (SACM) is standardised by the Object Management Group (OMG). The intention of the metamodel is to promote model-based approach in the process of *System Assurance*, which is currently a manual approach that produces artefacts (i.e. Assurance Cases) that are (mostly) not consumable by computers. SACM is created to support structural argumentation approaches such as the *Goal Structuring Notation* (GSN) and *Claims, Arguments and Evidence* (CAE).

SACM captures not only fundamental concepts in the process of *System Assurance* such as *Claim*s and the relationships between *Claim*s, but also concepts such as *Artifact*s and *Terminologi*es, in the sense that supporting evidence and information involved in the argument can be modelled in greater precision. In addition, SACM promotes *modularity*, in the sense that assurance cases are organised in packages, which in turn organise argumentations, evidence and terminologies in corresponding packages. SACM also promotes *openness* in the sense that external information (such as external models and/or documents) can be linked via the facilities provided.

In this section, we discuss the packages in SACM and explain their intended usage. We will provide two types of examples to demonstrate how SACM can be used: for simple concepts to explain without further context we will use in-place examples; and for complex concepts which requires the context of understanding the entirety of SACM we will use dedicated examples, provided at the end of this section. Since OMG has not standardised the concrete syntax of SACM, we will use object diagrams in the examples.

4.1 SACM Overview

In overview, SACM is organised in five packages, as illustrated in Figure 7. The *Base* package provides the foundation of SACM, which will be discussed in Section 4.3. The *Argumentation* package captures the concepts used in arguing system properties (such as safety and/or security)[3], which will be discussed in Section 4.6. The *Terminology* package captures the concepts used in expressing

[3] System properties refer to the safety and/or security in the context of this paper, hereafter.

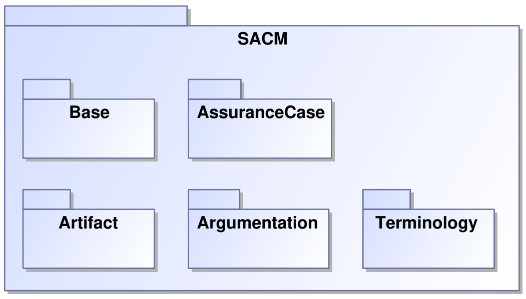


Fig. 7: Packages of SACM

the arguments regarding system properties, which will be discussed in Section 4.5. The *Artifact* package captures the concepts used in providing evidence for the arguments made for system properties. The *Artifact* package will be discussed in Section 4.4. Finally, the *AssuranceCase* package captures the concepts in *System Assurance*, which combines all the elements in other SACM packages to form a *System Assurance Case*. The *AssuranceCase* package will be discussed in Section 4.2.

4.2 SACM AssuranceCase Package

Although the *Base* package provides the foundation of SACM, it is necessary to discuss the *AssuranceCase* package first as it provides an insight on how an *Assurance Case* in SACM is organised. The structure of the *AssuranceCase* package is shown in Figure 8.

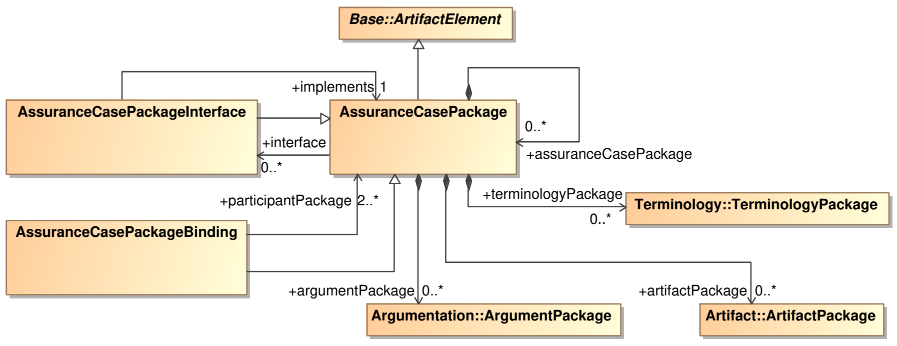


Fig. 8: SACM AssuranceCase Package.

The core element in the *AssuranceCase* package is the *AssuranceCasePackage* element, which extends the *ArtifactElement* in the *Base* package. The implication is that an *AssuranceCasePackage* can be considered to be an artefact. An *AssuranceCasePackage* can hold a number of *ArgumentPackages*, *TerminologyPackages* and *ArtifactPackages*, which are modeled in the *Argumentation*, *Terminology* and *Artifact* packages respectively.

Sometimes, the developer of an *AssuranceCasePackage* may want to make part of the *AssuranceCasePackage* available externally so that they can be reused. Consider the scenario where a system is composed of components A and B and *AsssuranceCasePackages* ACPA and ACPB are created respectively for A and B (which contain structured argumentations with regard to safety and/or security properties for A and B). The developer may want to make parts of the argumentations public so that during system integration, where A and B are integrated to form a system, their assurance cases ACPA and ACPB can also be integrated to form a new *AssuranceCasePackage*. To disclose only necessary contents externally, one may make use of the *AssuranceCasePackageInterface* to do so.

The premise of system integration in safety related domains is to integrate assurance cases of systems to form an overall assurance case. SACM handles this scenario with the *AssuranceCasePackageBinding*, which binds two or more *AssuranceCasePackageInterfaces* together to form an overall *AssuranceCasePackage*. This particular scenario is discussed in Section 5.4.

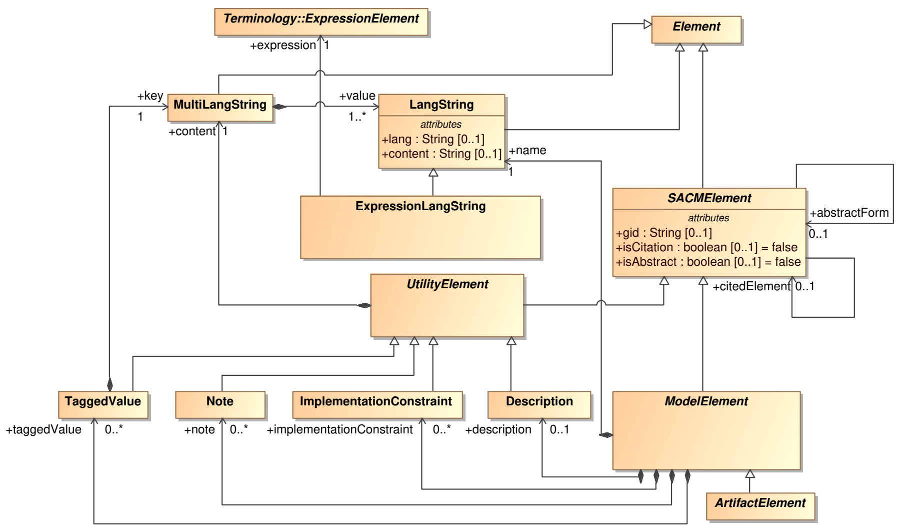


Fig. 9: SACM Base Package.

4.3 SACM Base Package

The *Base* package captures the foundational concepts of SACM, the structure of the *Base* package is shown in Figure 9. The base element of all SACM elements is *Element*. Its direct children are *LangString*, *MultiLangString* and *SACMElement*.

LangString is used as an equivalence to *String* (the value of which is held in the *+content* feature), except it captures an additional feature *+lang* which allows the users to define what language is used in the *LangString*. *ExpressionLangString* is used to not only record a *String* in SACM, but also refer to its corresponding *Expression* organised in a *TerminologyPackage*. The usage of *ExpressionLangString* is discussed in Section 4.5 *MultiLangString*, as its name suggests, is used to express the same semantics using different languages. For example, to express 'hazard' in both English and German, the user can create a *MultiLangString* with two *LangString*, as shown in Figure 10.

The *MultiLangString* can then be associated to other SACM elements to denote the same meaning. What is more important than multiple natural language support is the support for computer languages. We envision that in the future, system assurance can benefit from automation, so that system assurance can be partially automated. In this case, *MultiLangString* can be used to hold both natural languages and computer languages (e.g. formal languages) to support automated reasoning of argumentations.

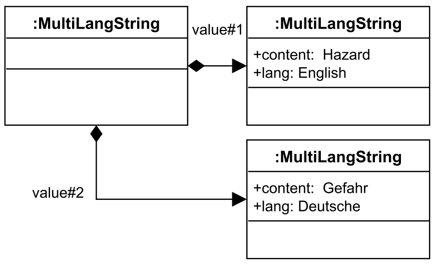


Fig. 10: Example *MultiLangString*.

SACMElement is an abstract element which lays the foundation of all SACM elements. *SACMElement* can record a *+gid* (*global identification*). This means that within the same *AssuranceCasePackage*, each *SACMElement* should have an unique ID. *SACMElement* is also able to refer to (or 'cite') other *SACMElements*, which is useful for implicit references discussed in Section 5.1. The *citedElement* and *isCitation* properties are used for this purpose. A *SACMElement* can also be abstract, denoted by the *isAbstract* property. A *SACMElement* can also be an *abstractForm* of another, which is discussed in Section 5.3.

ModelElement further refines *SACMElement* which contains a *name* (of type *LangString*) and a set of *UtilityElements*. A *ModelElement* can contain a *Description* which describes its contents. Like previously mentioned, a *Description* can be expressed in any language via its usage of *MultiLangString*. A *ModelElement* can also contain an *ImplementationConstraint*, which is used to specify the instantiation rules for assurance case templates (that of similar to safety case patterns). A *ModelElement* can also contain a number of *Note*s, to hold additional information rather than descriptions. Finally, a *ModelElement* can also contain a number of *TaggedValue*s, which are essentially {key, value} pairs. *TaggedValue* can be considered as an extension mechanism to allow the users to associate additional features to a *ModelElement* (other than the features modelled in the current version of SACM).

The *Base* package also defines the *ArtifactElement*, in the sense that all elements that extend *ArtifactElement* are considered to be *Artifact*s. The reason for this is discussed in Section 5.2.

In summary, the *Base* package lays the foundation for SACM, it provides facilities not to only express assurance cases (to be precise, assurance case models) in natural language, but also in computer languages. The *Base* package also provides a number of *UtilityElement*s so that the user can use them to describe *ModelElement*s as precisely as possible.

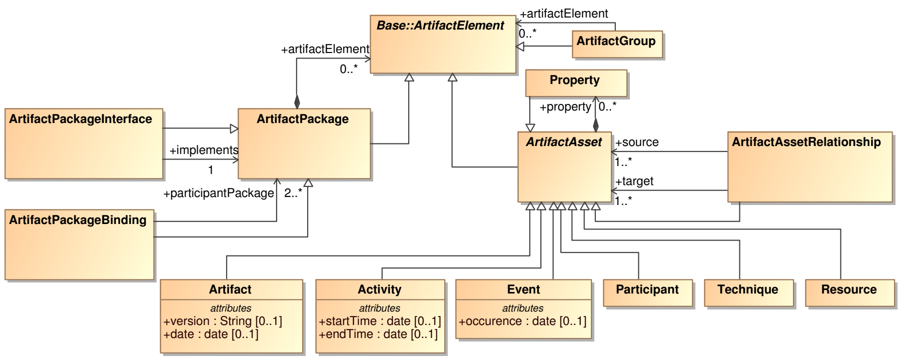


Fig. 11: SACM Artifact Package.

4.4 SACM Artifact Package

Before delving into the *Argumentation* package, it is necessary to discuss the *Artifact* package. The structure of the *Artifact* package is shown in Figure 11. All elements in the *Artifact* package extend *ArtifactElement* in the *Base* package. *ArtifactElement*s are organised in *ArtifactPackages* to promote modularity. System assurance case integration at the level of *ArtifactPackage*. Thus, *ArtifactPackageInterface* and *ArtifactPackageBinding* are used for this purpose. *ArtifactGroup* is a new concept introduced in SACM 2.0. As *ArtifactPackage* can contain rather a large number of *ArtifactElement*s, the *ArtifactGroup* provides the user with a means to selectively group *ArtifactElements*, so that the user can group/view *ArtifactElements* with their defined criteria.

ArtifactAsset allows the users to create corresponding artefact elements in SACM, it can contain a number of *Property*-ies to hold user-defined properties. *Artifact* is used to record a piece of information (e.g. hazard log, failure logic models, etc). *Activity* is used to record an activity (e.g. specification of requirements). *Event* is used to record an event (e.g. creation/modification of *Artifacts*). *Participant* is used to record participants involved in *ArtifactAssets*. *Technique* is used to record the techniques used in *Activities*. *Resource* is used to record a piece of resource, usually in the form of some electronic file. And finally, *ArtifactAssetRelationship* is used to link *ArtifactAssets* (e.g. connecting *Activity* to *Participants*). Note that the *ArtifactAssetRelationship* is a generic relationship, however, the user can choose to use *Property* to specify the purpose of an *ArtifactAssetRelationship*.

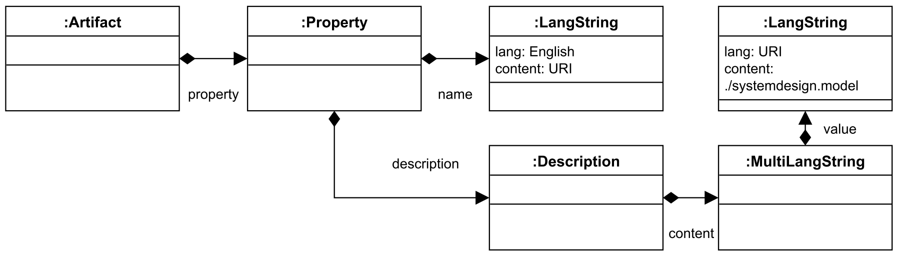


Fig. 12: External reference using *Property*.

One open question regarding the *Artifact* package is how to refer to external materials (such as system requirement, system design model, system failure logic model, etc.). To achieve automation, it is necessary to have a means to refer to external materials (especially models) so that all information can be gathered together to form an assurance case. The user may make use of the *Property* element, as shown in Figure 12 (consider elements filled in *white* first).

In Figure 12, a *Property* is associated to an *Artifact*[4], which has a *name* (of type *LangString*): 'URI' and a *Description*, which in turn specifies a file on local disk (systemdesign.model). In this way, the *Property* element is used as a reference to a local file.

SACM does not restrict how external materials should be referenced, the description provided above is one way of achieving it. It also depends on tool implementations on how external references should be handled.

[4] The *Artifact* should have its own features such as name and description, which is omitted here.

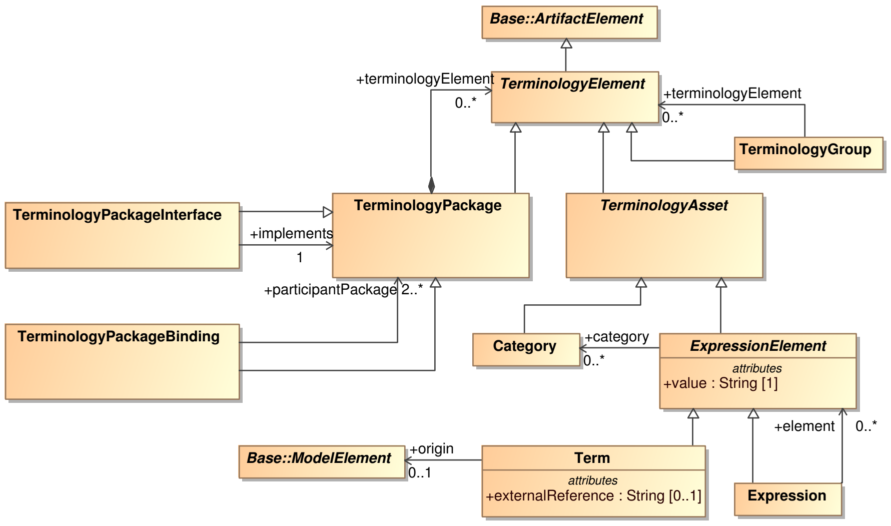


Fig. 13: SACM Terminology Package.

4.5 SACM Terminology Package

The *Terminology* package captures concepts that enable the users of SACM to describe their argumentations in greater precision. The structure of the *Terminology* package is shown in Figure 13.

The root element of the *Terminology* package is *TerminologyElement* which also extends *ArtifactElement* in the *Base* package. *TerminologyGroup*, *TerminologyAsset* and *TerminologyPackge* are direct children of *TerminologyElement*. *TerminologyAsset*s are organised in *TerminologyPackage*s to promote modularity. System assurance case integration can be done at the level of *TerminologyPackage*. *TerminologyInterface* and *TerminologyPackageBinding* are used for this purpose. As *TerminologyPackage* can also get large in size, *TerminologyGroup* provides the user with a means to selectively group *TerminologyAsset*s, so that the user of SACM can group/view *TerminologAsset*s as they wish.

ExpressionElement captures the concept of two kinds of expressions used in argumentations. *ExpressionElement* holds a *value* (of type String). *ExpressionElement* can either be an *Expression* or a *Term*. An *Expression* is used to model phrases and sentences. For example, when stating a sentence: **All identified hazards have been sufficiently mitigated.**, the *Expression* element can be used to to capture this sentence as shown in Figure 14 (element filled with cyan colour). Alternatively, the users of SACM may choose to capture arbitrary phrases in the sentence as an *Expression*, as shown in Figure 14 (element filled with yellow colour). The idea behind *Expression* is to store phrases/sentences in the *TerminologyPackage* like a set of defined grammar, so that the *Expressions* can be re-used. Of course the example in Figure 14 does not add too much value

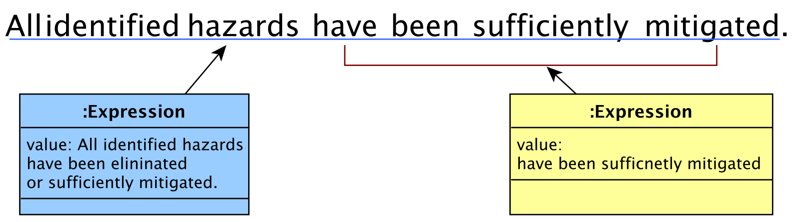

All identified hazards have been sufficiently mitigated.
:Expression
value: All identified hazards have been elininated or sufficiently mitigated.
:Expression
value: have been sufficnetly mitigated

Fig. 14: Example use of *Expression*.

to the assurance case, in realistic scenarios, only phrases that can be re-used should be captured in *Expressions*. This is completely the decision to make for the users and tool developers of SACM.

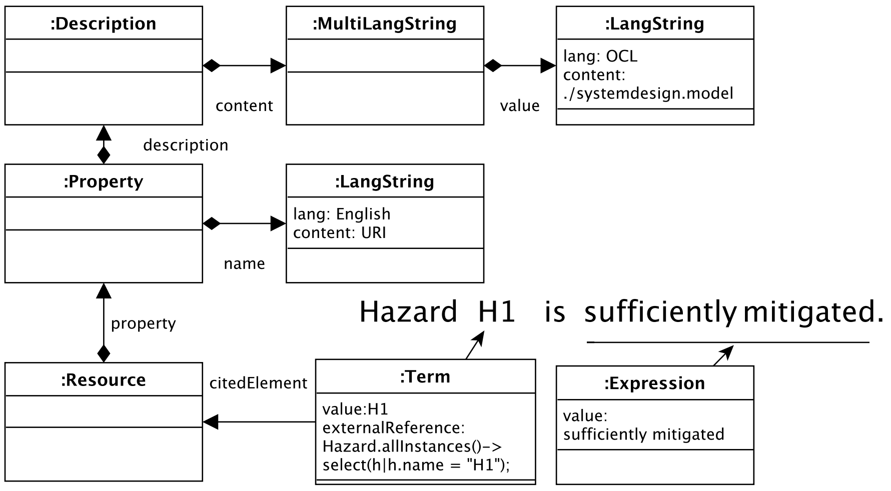


Fig. 15: Example use of *Expression* and *Term*.

The *Term* element is used to capture a terminology used in the process of system assurance. For example, in Figure 15 (first consider elements filled in white), a statement is made: **Hazard *H1* is sufficiently mitigated.** Within the sentence, *H1* in fact may cross-references to an identified hazard in a hazard log. Thus, *H1* can be captured using a *Term*. The location of the hazard log can be captured using a *Resource*[5] in the *Artifact* package, where the location of the *Resource* is captured using *Property*. In order to point to a specific model element (in this case, model element H1 in the hazardLog.model), the user may use OCL

[5] The details of the *Resource* is omitted.

in the +externalReference feature of the *Term*. This is illustrated in Figure 15. Again, SACM does not restrict how external references should be handled, the description provided above is one way of achieving it. It also depends on tool implementations on how external references are handled.

Sometimes, a term may refer to a *ModelElement* within the assurance case (via the +origin feature). This will be discussed in Section 4.6.

The users of SACM may also organise *ExpressionElement* into *Categories*. For example, the users may create an *Category* of Hazard, and associate the *Term* H1 to it.

4.6 SACM Argumentation Package

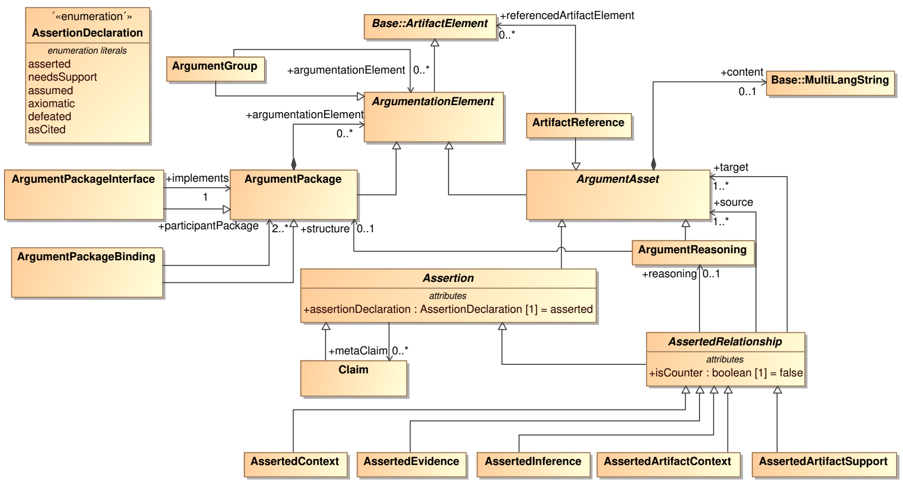


Fig. 16: SACM Argumentation Package.

The *Argumentation* Package captures the concepts required to model structured arguments regarding system properties. The structure of the *Argumentation* package is shown in Figure 16. The root element of the *Argumentation* package is the *ArgumentationElement*, which is a direct child of *ArtifactElement* in the *Base* package. This implies that all elements in the *Argumentation* package are also considered to be artifacts.

To promote modularity, argumentations are organised in *ArgumentPackage*s. *ArgumentationPackage* can contain a number of *ArgumentationElement*, which can either be *ArgumentPackage*s (and their children) or *ArgumentAsset*s. *ArgumentAsset* can store a *content*, as discussed in Section 4.3, the content can be in any language[6].

[6] Via the usage of *MultiLangString*

ArgumentAsset and its children are the elements that form the structured argumentation in the *Argumentation* package. An *Assertion* has an *AssertionDeclaration* to distinguish different kinds of *Assertion*s. The *AssertionDeclaration*s are as follows:

- **asserted** - the default declaration, means that the *Assertion* is made and is supported by evidence;
- **needsSupport** - a flag indicating that the *Assertion* is not supported yet (needs further development);
- **assumed** - a flag indicating that the truth of the *Assertion* is assumed although no supporting evidence is provided;
- **axiomatic** - a flag indicating that the truth of the *Assertion* is axiomatically true without further supporting evidence;
- **defeated** - a flag indicating that the truth of the *Assertion* is invalidated by a counter-evidence and/or argumentation;
- **asCited** - when an *Assertion* 'cite's another *Assertion* (via the +citedElement property in *SACMElement*), the truth of the *Assertion* is transitively derived from the value of the cited *Assertion*.

The use of *AssertionDeclaration* will be illustrated in Section 5.1.

Sometimes it is necessary to argue the confidence in the arguments provided in the assurance case. This can be achieved using the *+metaClaim* feature of the *Assertion* element. *+metaClaim* as its name suggests, is a *Claim* about an *Assertion* to argue the soundness of the *Assertion*. Within the meta *Claim*, one may write "full confidence in Claim C1 is achieved via..." in its description, within which the *Term* G1 refers to another *Claim* in the same SACM model. This is where the *+origin* feature of the *Term* is used, in the sense that the *Term* G1 refers to a *Claim* G1 within the same SACM model.

Claim and *AssertedRelationship* are the core elements of the *Argumentation* package. *AssertedRelationship* are used to connect *ArgumentAsset*s to form structured argumentation (*AssertedRelationship*s can also be *counter* relationships, indicated by the *+isCounter* feature):

- **AssertedContext** - this relationship is used to connect contextual *Assertion*s to an *Assertion*;
- **AssertedEvidence** - this relationship is used to connect evidence (referenced via *ArtifactReference*) to an *Assertion*;
- **AssertedInference** - this relationship records the inference between on or more *Assertion* and another *Assertion*;
- **AssertedArtefactContext** - this relationship is used to connect contextual artefacts (via *ArtifactReference*) to an artefact (via *ArtifactReference*);
- **AssertedArtefactSupport** - this relationship is used to connect supporting artefacts (via *ArtifactReference*) to an artefact (via *ArtifactReference*).

ArtifactReference is a type of *ArgumentAsset*, which is able to refer to an *ArtifactElement*. *ArtifactReference* is typically used to refer to evidence stored in an *Artifact* package. In addition, it can refer to any element that extend

ArtifactElement (all elements in the *AssuranceCase*, *Argumentation*, *Artifact* and *Terminology* packages are *ArtifactElements*).

ArgumentReasoning is also a type of *ArgumentAsset*, it is used to provide a explanatory description for an *AssertedRelationship*. An detailed discussion of *ArgumentReasoning* is in Section 5.2.

4.7 Summary

In this section, we discussed the packages defined in SACM. We explain the intended semantics of the elements in the packages and we used some in-place examples to illustrate how SACM can be used to construct a system assurance case with argumentations regarding system properties (i.e. safety and/or security) and its supporting evidence/artefact (using the *Artifact* and *Terminology* packages of SACM). In the next section, we will illustrate how SACM can be used with more concrete examples.

5 SACM: Examples

In this section, we provide concrete examples on how SACM is used to construct structured argumentation, and how SACM can be used to form argumentation patterns (similar to GSN argument patterns). Since GSN notations are widely accepted and understood, we compare GSN depictions with their equivalent model elements in SACM to illustrate how SACM elements can be used to denote the same meaning carried by their GSN counterparts.

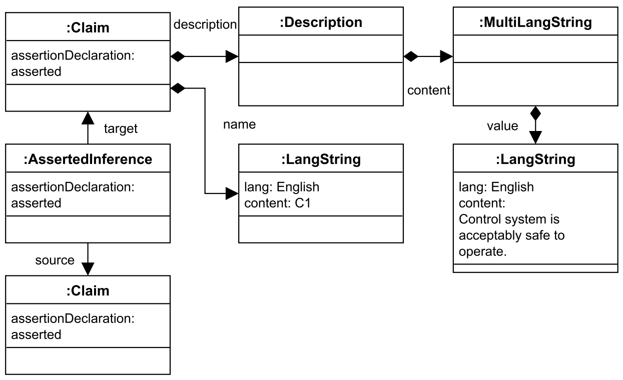


Fig. 17: An **asserted** *Claim* in SACM.

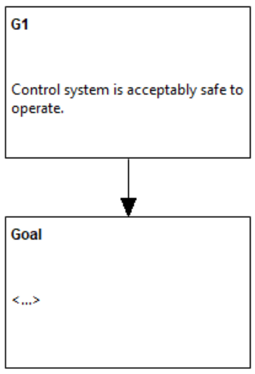


Fig. 18: A *Goal* structure in GSN.

5.1 Example: Making Claims and Citations

Figure 17 provides an example on how to construct a *Claim*. A *Claim* can have a name and a description, captured by *LangString* and *Description* respectively. A *Claim* is **asserted** if it is supported by other *Claims*. In this example, it is connected by an *AssertedInference* with another *Claim* (details ommitted). The equivalence of *Claim* C1 Figure 17 in GSN is provided in Figure 18, where the *Goal* G1 bears the same semantics with C1 (and how *Goal* G1 is supported by another *Goal*[7]).

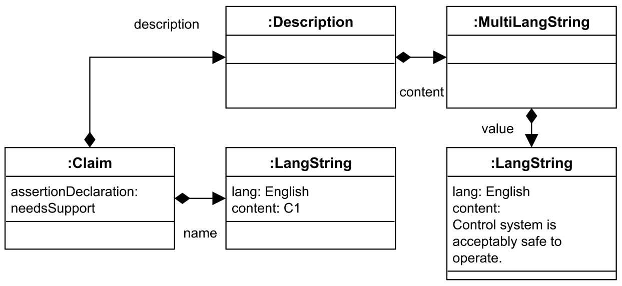


Fig. 19: A **needsSupport** *Claim* in SACM.

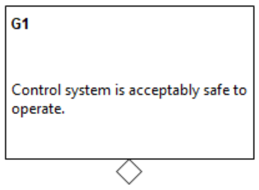


Fig. 20: A undeveloped *Goal* in GSN.

Like previously mentioned, a *Claim* can be marked as **needsSupport**. Figure 19 illustrates the use of this declaration. A *Claim* that is not supported by any argument/evidence should be marked as **needsSupport**. The equivalence of this *Claim* in GSN is shown in Figure 20, where *Goal* G1 bears the same semantics of *Claim* C1. When a *Goal* is not supported by argument/evidence, it is marked as *undeveloped*.

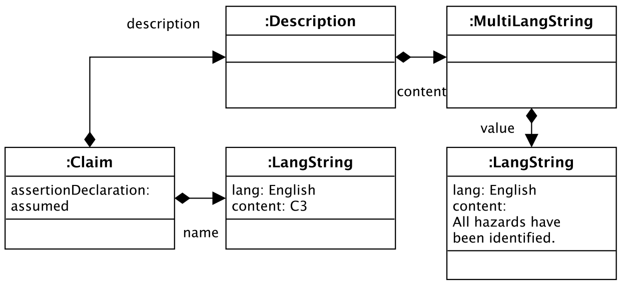


Fig. 21: An **assumed** *Claim* in SACM.

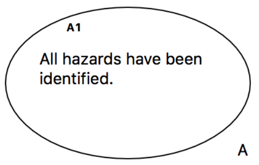


Fig. 22: An *Assumption* in GSN.

[7] Details of subsequent *Goals* are omitted.

A *Claim* marked as **assumed** is used to state an assumption in the argumentation. Figure 21 illustrates an **assumed** *Claim* in SACM (where the user assumes that *all hazards have been identified*). The users are responsible to declare that a *Claim* is **assumed** when they make assumptions about the system. The equivalence for **assumed** *Claim* is an *Assumption* in GSN, as shown in Figure 22.

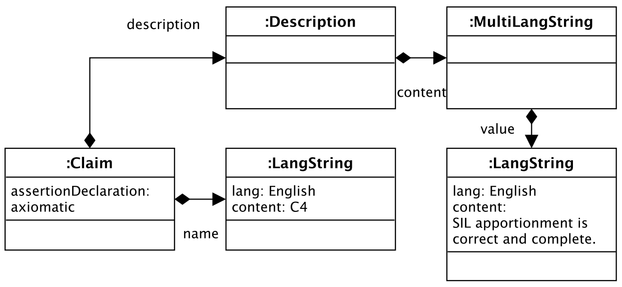


Fig. 23: An **axiomatic** *Claim* in SACM.

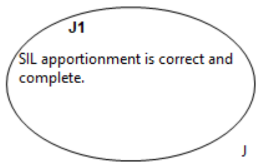


Fig. 24: A *Justification* in GSN.

A *Claim* marked as **axiomatic** is used to state a well agreed fact (presumably among stakeholders), which does not need any further support by arguments/evidence. The equivalent of an **axiomatic** *Claim* in GSN is a *Justification*, which does not need further supporting argument/evidence to support its content. Figure 23 and Figure 24 illustrates the use of **axiomatic** *Claim* and *Justification* respectively.

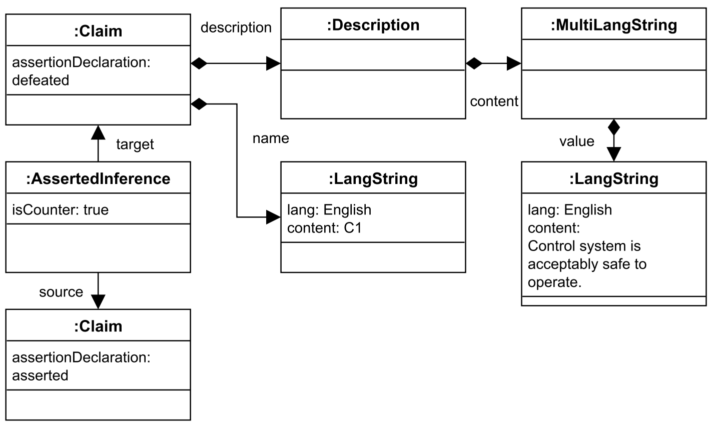


Fig. 25: A **defeated** *Claim*.

A *Claim* is marked as *defeated* if the truth of the claim is proven to be false by supporting argument/evidence. Figure 25 provides an example of a **defeated** *Claim*, the *Claim* C1 is connected with another *Claim* (we do not consider the detail of this *Claim* for this example) by an *AssertedInference*, but with its +isCounter feature to be *true*. This means that the *AssertedInference* is a counter-argument, which negates the truth of *Claim*, if the *+source* of the *AssertedInference* is *true*.

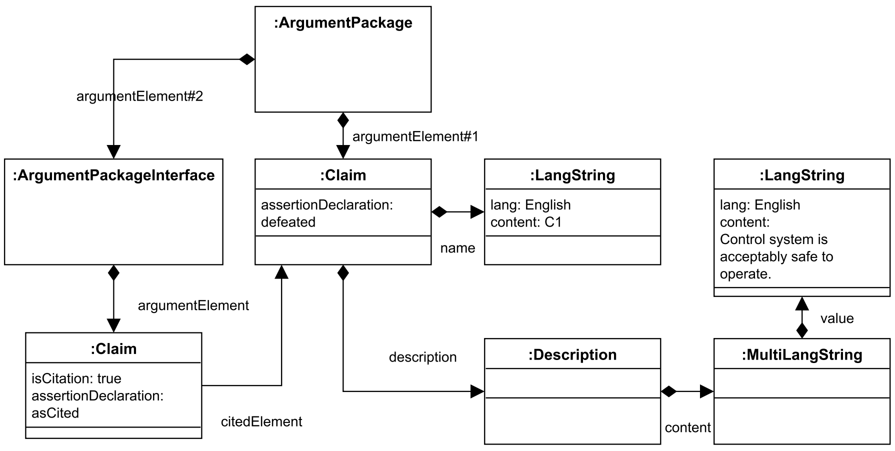


Fig. 26: An **asCited** *Claim*.

As previously discussed, the users of SACM are able to selectively disclose the content of an *ArgumentPackage* via the use of *ArgumentInterface*. Figure 26 illustrates how an *ArgumentPackageInterface* can be used. In this example, a *Claim* C1 is held within an *ArgumentPackage*, which contains an *ArgumentPackageInterface* that in turn contains another *Claim*, which is a citation (its *+isCitation* is true). The *Claim* '**cites**' C1 in the *ArgumentPackage* via the *+citedElement* feature (which is defined in the *SACMElement* element in the *Base* package). Hence, the **asCited** declaration on the *Claim* should be used, which means that the truth of the *Claim* as a citation, depends on the *Claim* that it cites (in this case, *Claim* C1).

The *+isCitation* and *+citedElement* features can be used in the same way in *ArtifactPackageInterface* and *TerminologyInterface*, should the developers of assurance cases wish to disclose information in their corresponding packages.

5.2 Example: AssertedRelationships and ArgumentReasoning

The intent of *Claims* are distinguished using *AssertionDeclarations*. Different types of *Claims* are connected using different *AssertedRelationships* to form a structured argumentation.

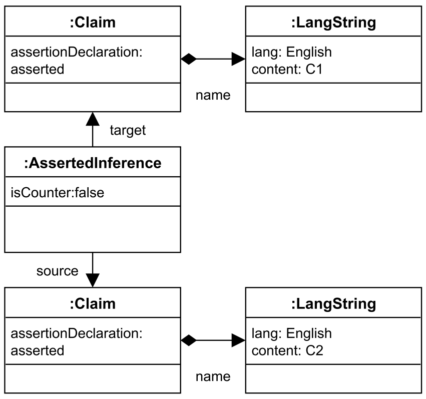


Fig. 27: An example of **AssertedInference**.

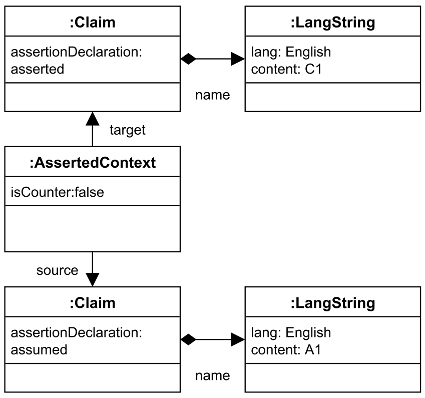


Fig. 28: An example of **AssertedContext**.

The *AssertedInference* denotes the inference between one or more *Assertion*s an another *Assertion*, figure 27 provides an example, where the truth of *Claim* C1[8] is inferred from the truth of Claim C2.

The *AssertedContext* connects contextual *Assertion*s to an *Assertion*, Figure 28 provides an example, where the *Claim* A1 provides contextual information for *Claim* C1.

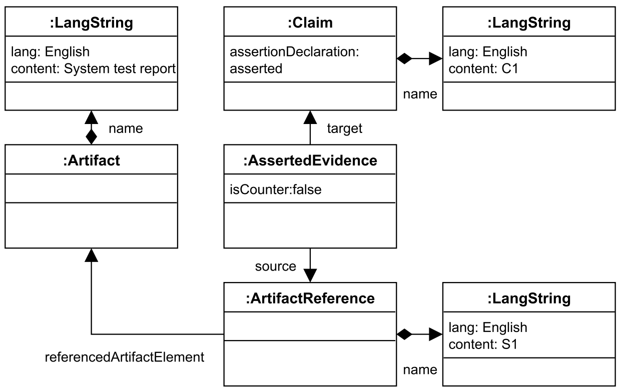


Fig. 29: An example of **AssertedEvidence**.

The *AssertedEvidence* connects evidence to an *Assertion*, Figure 29 provides an example, where the *ArtifactReference* S1 (which refers to an *Artifact* or-

[8] description details are omitted due to space limit

ganised in an *ArtifactPackage*, details omitted) provides evidence for the *Claim* C1.

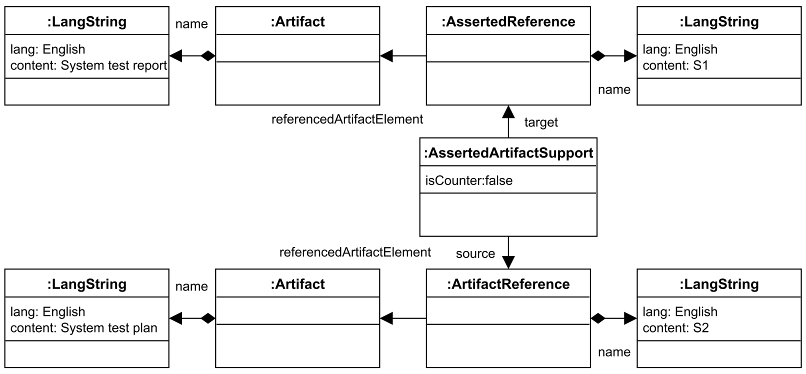


Fig. 30: An example of **AssertedArtifactSupport**.

The *AssertedArtifactSupport* connects supporting artefacts to an artefact, Figure 30 provides an example, where *ArtifactRefernce* S2 (which cites an *Artifact System test plan*) supports the *ArtifactReference* S1 (which cites the *Artifact System test report*), in the sense that the system test plan supports the system text report. The *AssertedArtifactContext* is also used on *ArtifactReferences*, except that it connects contextual artefacts to another artefact.

As previously mentioned, all elements that extend *ArtifactElement* can be considered an artifact, this is due to the fact that there are *supporting* and *contextual* relationships between elements rather than those in the *Artefact* package. For example, assurance case developer may want to express that one *ArgumentPackage* bears supporting arguments to another *ArgumentPackage*, in this case, the user may want to use *AssertedArtifactSupport* to connect two *ArtifactReference* which reference these two *ArgumentPackage*s.

Sometimes, a justification of *AssertedRelationship*s can be attached to further clarify the reasons of the *AssertedRelationship*, *ArgumentReasoning* is used for this purpose. Figure 31 illustrates the use of *ArgumentReasoning*, where top level *Claim* C1 is backed by its sub-*Claim* C2, which are connected by an *AssertedInference*, the user may choose to give the *AssertedInference* a reason with the use of *ArgumentReasoning*. In this case, an *ArgumentReasoning* S1 is attached to the *AssertedInference*, which states that the argument is made by *Argument over all identified hazards*. Supportive and contextual information can also be associated to *ArgumentReasoning* via the use of *AssertedSupport* and *AssertedContext*. Since an *ArgumentReasoning* is not an *Assertion*, there is little value in arguing the soundness of the *ArgumentReasoning*.

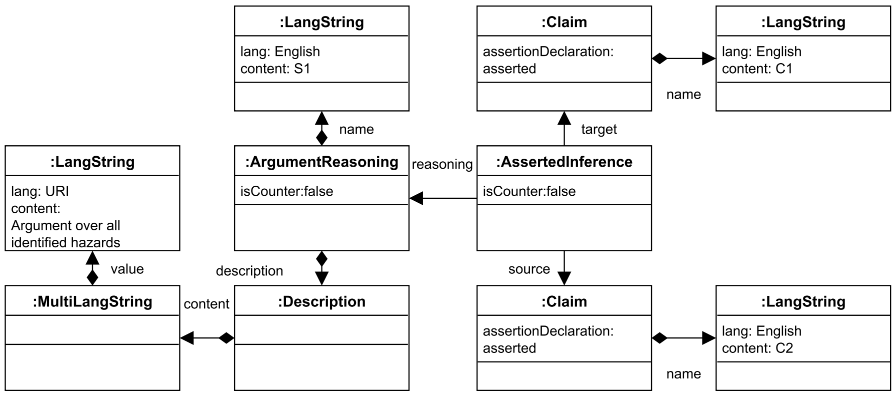


Fig. 31: An example of **ArgumentReasoning**.

5.3 Example: Argumentation Patterns

In GSN, there is a concept of *GSN Pattern*s [22], which are essentially templates that enable the re-use of good practice in safety cases. SACM provides similar concepts so that the user can construct templates in SACM, and at a later stage, to *instantiate* the templates by populating actual system information in the templates. Figure 32 shows an argumentation pattern in SACM. For comparison, the content of the pattern is identical to the GSN pattern shown in Figure 6. Note that some details are omitted due to the complexity of the SACM model.

There is a *TerminologyPackage* in the upper part of Figure 32 named TP1, within which contains an *Expression* and a *Term*[9]. In the lower part of the figure, there is an *ArgumentPackage* AP1, which contains the structured argumentation pattern (N.B. the packages are placed in this way to make the figure more readable and does not denote the priority of the packages).

The top level *Claim* of this pattern is **G1**, which maps to the *Goal* **G1** in Figure 6. Note that **G1**'s *+isAbstract* is set to *true* since this is an abstract *Claim* (a template). Now we focus on the *description* of **G1**, within which an *ExpressionLangString* is used. The *ExpressionLangString* refers to the *Expression* in *TerminologyPackage* TP1 (which contains value: {*System X*} *is safe*). The curly brackets are a convention in GSN, which are called *role*s in GSN. *Role*s are essentially place holders in the pattern, the contents enclosed in the curly brackets will be replaced by actual system information if the pattern gets instantiated. In this case, when the pattern is instantiated, {System X} will be replaced with the actual system name.

In the *TerminologyPackage* TP1, the *Expression* refers to the *Term System X*. This is a typical use of *Term* in structured argumentation. When the pattern is *instantiated*, the *Term* should have its *+externalReference* feature populated to import actual system information.

[9] The containment is described in this way to improve comprehensibility.

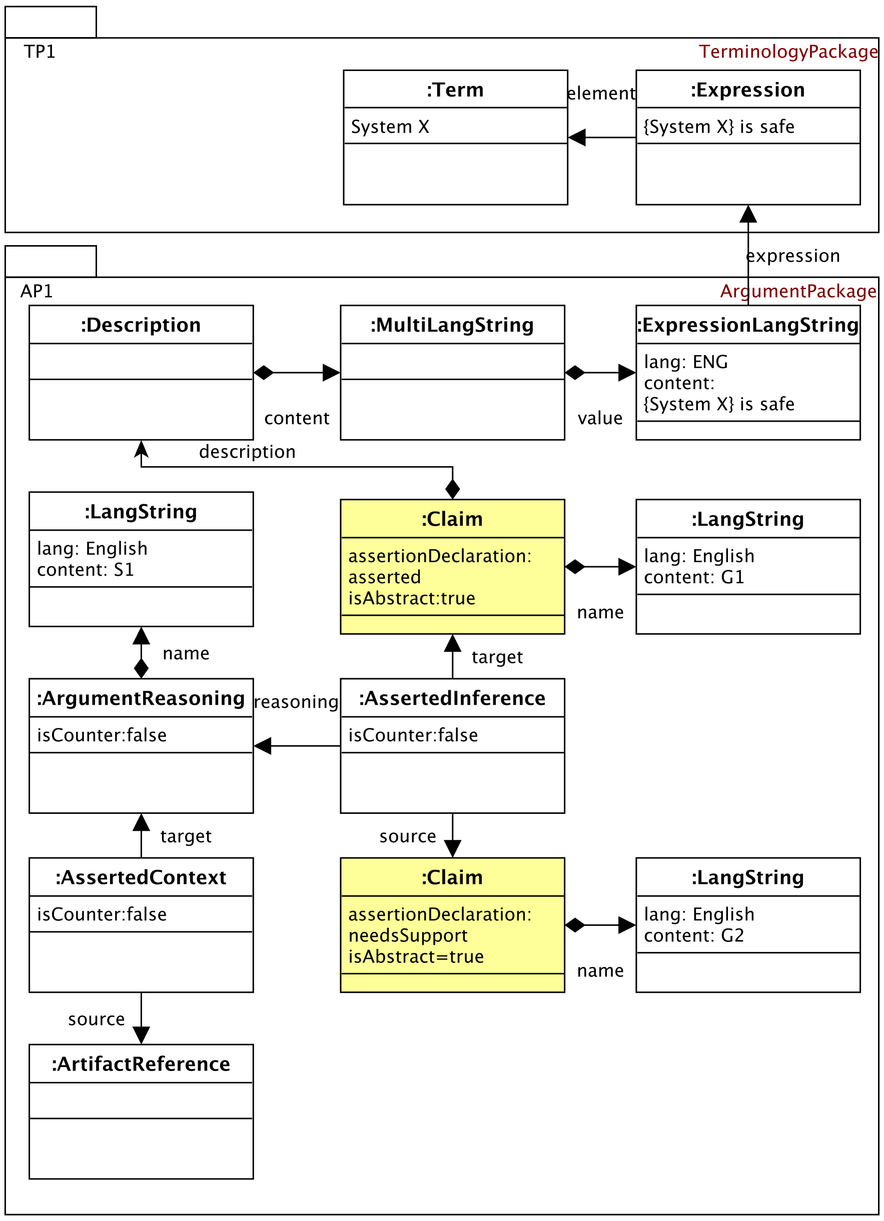


Fig. 32: An example SACM argumentation pattern.

The structure of the argumentation in Figure 32 follows that in Figure 6, where the *Claim* **G2** maps to *Goal* **G2**, the *ArgumentReasoning* **S1** maps to *Strategy* **S1**. Note that the *Claim*s in yellow colour are the templates, which have their *+isAbstract* feature set to true.

GSN pattern instantiation is often a manual procedure as safety case developers need to comprehend the GSN pattern and replace the *role*s in the pattern with actual system information. In [23], a model-based approach is proposed, which makes use of a *weaving model* to link the elements in the GSN with elements in system models. This is typically due to the fact that in GSN there are no means to specify instantiation rules for the GSN patterns. In SACM, as previously mentioned in Section 4.3, the element *ImplmentationConstraint* can be used to specify the instantiation rules for patterns. The users of SACM can associate *ImplementationConstraint* to any element and the use of language is not restricted. Evidently, pattern instantiation needs tool support and model management engine to execute the *ImplementationConstraint*. The language used in *ImplementationConstraint* are not limited to computer languages. *ImplmentationConstraint*s described in natural languages can also be used as instantiation rules, except that the instantiation procedure is limited to manual only.

When a SACM model instantiates a SACM pattern, it can relate to the SACM pattern via the *+abstractForm* feature. For example, if a *Claim* C1 created by instantiating the template **G1**, it can refer to **G1** via the *+abstractForm* for future reference.

5.4 Example: A case study on the European Train Control Systems (ETCS)

As previously discussed, in SACM, assurance cases can be integrated to form an overall assurance case. Integrating assurance cases is a typical task performed when system components (or sub-systems) are integrated to form an overall system. Sometimes, system components/sub-systems are developed independently, together with their assurance cases. Hence, in safety-related domains, the premise of system integration is integration of assurance cases of components/sub-systems, to argue the dependability of the to-be-integrated system.

To illustrate how assurance case integration is performed in SACM. We provide an example[10] taken from an engineering story in European Train Control System (ETCS) we encountered in the research carried out in the DEIS project [20]. In this example, we consider the scenario where the assurance cases of on-board and side-track components of ETCS are integrated to form an overall assurance case.

The example SACM model for the ETCS case study is shown in Figure 33. For simplicity, we only show the top level *Claim*s of the assurance cases. In Figure 33, there are three *AssuranceCasePackage*s. *On-Board ACP* (at the top

[10] We reduced the complexity of the model structure by incorporating the name and description of the elements directly in the elements themselves.

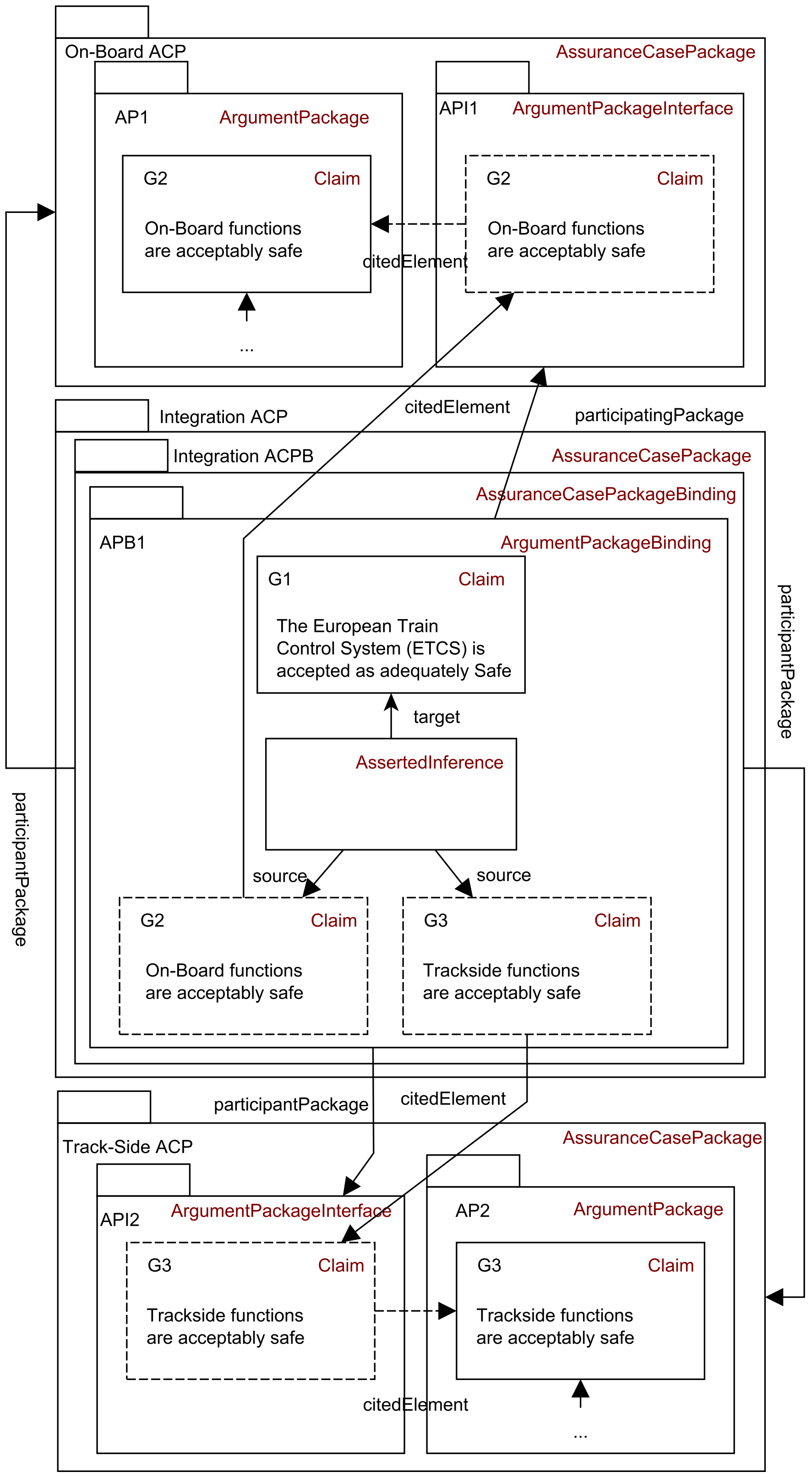


Fig. 33: A case study on the assurance case integration of the European Train Control Systems (ETCS).

of the figure) is the assurance case for the on-board component of ETCS, *Track-Side ACP* (at the bottom of the figure) is the assurance case for the side-track component of ETCS, and *Integration ACP* (in the middle of the figure) is the integration assurance cases which integrates the two component assurance cases together.

For *On-Board ACP*, *ArgumentPackage AP1* contains the argument regarding the safety of the on-board component. As discussed in Section 4.2, system engineers may wish only to disclose the top level *Claim* externally, hence *ArgumentPackageInterface AP1* is used which contains a citation of *G2* which will be referenced externally. Same principle is applied to *Track-Side ACP*, where the top level *Claim G3* is cited in the *ArgumentPackageInterface API2*. It is to be noted that both *On-Board ACP* and *Track-Side ACP* contains their *ArtifactPackage*s and *TerminologyPackage*s, which are not shown due to the complexity of the model structure.

To integrate *On-Board ACP* and *Track-Side ACP*, *AssuranceCasePackage* named *Integration ACP* is created. *Integration ACP* contains an *AssuranceCasePackageBinding* (*Integration ACPB*) specifically to bind *On-Board ACP* with *Track-Side ACP*. Within *Integration ACPB*, an *ArgumentPackageBinding* (*APB1*) is used to bind *API1* and *API2* via the *+participantPackage* feature. In *APB1*, top level *Claim G1* argues the safety of ETCS and two supporting *Claim*s *G2* and *G3* are in place. Note that *G2* and *G3* are citation *Claim*s which cites *G2* in *API1* (which in turn cites *G2* in *AP1*) and *G3* in *API2* (which in turn cites *G3* in *AP2*). It is also to be noted that within the *AssuranceCasePackageBinding* (Integration ACPB), there are also *ArtifactPackageBinding*s and *TerminologyPackageBinding*s which binds the artifacts and expressions used in the on board component and track side component.

The integration of assurance cases in SACM is achieved using various package bindings. It is also possible to include additional arguments in the binding *AssuranceCasePackage* when deemed necessary. Users of SACM may also argue the trustworthiness of the cited *Claim*s in other packages to ensure the confidence in citing argument elements.

6 Metamodel for existing notations and their interoperabilities to SACM

SACM is designed to support existing safety notations such as GSN and CAE. In previous sections, we briefly demonstrated the semantics of SACM elements by comparing them with GSN notations. In this section, we provide a GSN metamodel and a CAE metamodel that are compliant to SACM. We also discuss the interoperability from GSN and CAE to SACM.

6.1 The GSN Metamodel and the inteoperability from GSN to SACM

As previously discussed, SACM provides a richer set of features comparing to GSN, which includes the ability to standardise evidential and informational artifacts in the models, standardising controlled grammar and terminologies, as well as modular organisation and integration of artifacts and terminologies. In general, creating a metamodel for GSN is a simple task, for there are only several concepts that GSN captures. However, we think it is more ideal to create the GSN metamodel by extending SACM elements, so that not only the GSN metamodel can inherit features provided by SACM, but also making the interoperability from GSN to SACM easier.

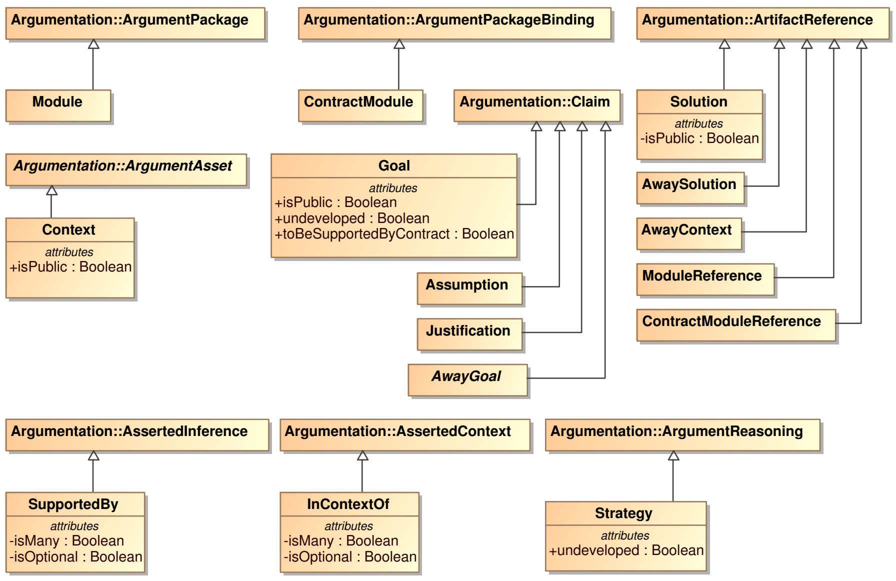


Fig. 34: SACM compliant GSN metamodel.

Our version of the GSN metamodel is shown in Figure 34. In GSN, argumentations are organised in *Module*s, which is made as a sub-type of *ArgumentPackage* in SACM; *ContractMoudle* are essentially contracts that binds *Module*s together, thus it is a sub-type of *ArgumentPackageBinding*.

Elements *Goal*, *Assumption*, *Justification* and *AwayGoal* are made sub-types of *Claim* in SACM. A *Goal* can be *uninstantiated*, which basically means it is abstract, this is captured by the *+isAbstract* feature in SACM's *SACMElement* class. A *Goal* can be *public*, which is deprecated in SACM, a *Goal* can also be *undeveloped* and *toBeSupportedByContract*, which are captured individually.

Elements *Solution*, *AwaySolution*, *AwayContext*, *ModuleReference* and *ContractModuleReference* are sub-types of *ArtifactReference* in SACM as they refer to artefacts that contain information they represent. *Context* is a slightly complicated concept, as it can either be a statement stating the context of a *Claim*, or it can refer to contextual information stored in an artefact. Thus, *Context* is made a sub-type of *ArgumentAsset*.

SupportedBy is made a sub-type of *AssertedInference* and *InContextOf* is made a sub-type of *AssertedContext*. *Strategy* is made a sub-type of *ArgumentReasoning* for it explains the intention of an *AssertedRelationship*.

The way that the GSN metamodel is created makes it inherently capable of modelling artefacts and terminologies due to the fact that the GSN metamodel also inherits the the *Base*, *AssuranceCase*, *Artifact* and *Terminology* packages. Our vision is that such GSN metamodel is able to create goal structures, and link evidential and contextual information from the goal structures to their supporting materials, modelling using the *Artifact* and *Terminology* packages.

```
let strategy = Strategy to be transformed;
let argumentReasoning = new ArgumentReasoning;
let incomingSupportedBy = the incoming SupportedBy to strategy;
let outgoingSupportedBy = all outgoing SupportedBys from strategy;
argumentReasoning.name = strategy.name.equivalent();
argumentReasoning.description = strategy.description.equivalent();
if strategy.uninstantiated == true then
    argumentReasoning.isAbstract = true;
end
let supportedByToSolution = all SupportedBys from outgoingSupportedBy
that connects to a Solution;
if supportedByToSolution is not empty then
    let assertedEvidence = new AssertedEvidence;
    assertedEvidence.target = incomingSupportedBy.source;
    for supportedBy in supportedByToSolution do
        assertedEvidence.source.add(supportedBy.target);
    end
end
let supportedByToGoal = all SupportedBys from outgoingSupportedBy
that connects to a Goal;
if supportedByToGoal is not empty then
    let assertedInference = new AssertedInference;
    assertedInference.target = assertedInference.source;
    for supportedBy in supportedByToGoal do
        assertedInference.source.add(supportedBy.target);
    end
end
```

Algorithm 1: Transforming Strategy into ArgumentReasoning

To enable interoperability from GSN to SACM, we provide a model-to-model transformation[11] defined using the Epsilon Transformation Language [24]. Most part of the transformation is straight forward - instances of the GSN elements are transformed into instances of the SACM elements that the GSN elements extend. There is one exception: the transformation from *Strategy* to *Argumen-*

[11] Available at: https://github.com/wrwei/MDERE/blob/master/technical

tReasoning. As discussed in Section 5.2, an *ArgumentReasoning* is associated to an *AssertedRelationship*, but in GSN, a *Strategy* acts as a node in a goal structure. This requires that analysis to be performed during the transformation, which is shown in Algorithm 1.

6.2 The CAE metamodel and the interoprability from CAE to SACM

Claims, Arguments and Evidence (CAE) [4] is another widely used notation to construct safety arguments. Concepts in CAE are similar to those in GSN. There has not been an official metamodel defined for CAE. Thus, we provide our own version of CAE that extends SACM, shown in Figure 35.

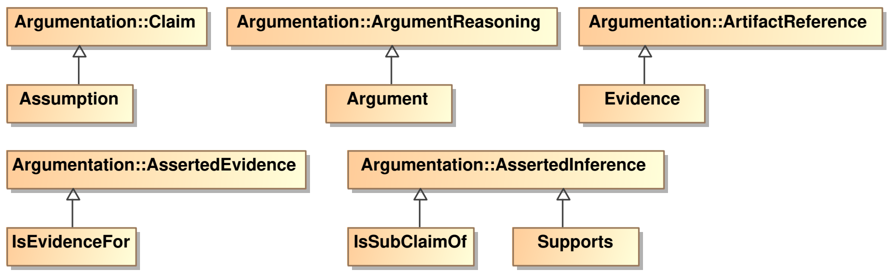

Argumentation::Claim
Argumentation::ArgumentReasoning
Argumentation::ArtifactReference
Assumption
Argument
Evidence
Argumentation::AssertedEvidence
Argumentation::AssertedInference
IsEvidenceFor
IsSubClaimOf
Supports

Fig. 35: SACM compliant CAE metamodel.

In CAE, there is a notion of *Claim*, thus it is not necessary to duplicate *Claim* in SACM. The *Argument* elements in CAE provides a description of the argument approach, which is functionally equivalent to *ArgumentReasoning*, thus it is made a sub-type of *ArgumentReasoning*. *Evidence* is made a sub-type of *ArtifactReference* because it is a reference to evidential materials. In CAE, there is a notion of *Assumption*, which is made a sub-type of *Claim*.

In CAE, there are three types of relationships, the *IsEvidenceFor* connects *Evidence* with *Claims*, thus is made a sub-type of *AssertedEvidence*; the *IsSubClaimOf* relationship connects sub-*Claims* to *Claim* and is made a sub-type of *AssertedInference*; the *Supports* relationship connects *Arguments* to *Claim* and is also made a sub-type of *AssertedInference*. Since there is no notion of modules in CAE, the argumentation is contained in *ArgumentPackages* inherited from SACM.

The transformation from CAE to SACM is similar to the transformation from GSN to SACM (which is also implemented in Epsilon Transformation Language), with the same issue that a model analysis is needed when transforming *Argument* to *ArgumentReasoning*. The detailed transformation is made publicly available[12].

[12] Available at: https://github.com/wrwei/MDERE/blob/master/technical

7 Tool Support and Future Work

7.1 Assurance Case Modelling Environment - ACME

With all its power in model-based system assurance, there is one drawback of SACM, which is the lack of concrete syntax, i.e. graphical representations of SACM. Without graphical representations, it is typically difficult for engineers to construct SACM. Hence, in order to exploit the benefits provided by SACM, we implement a tool (Assurance Case Modelling Environment, ACME[13]) based on the GSN metamodel we discussed in Section 6.

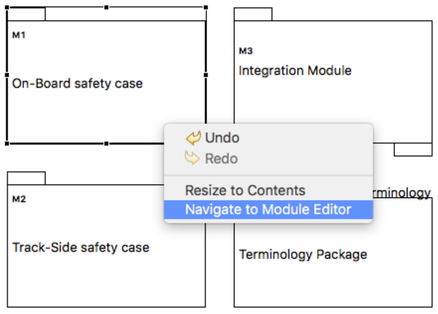


Fig. 36: The Assurance Case Package View of ACME.

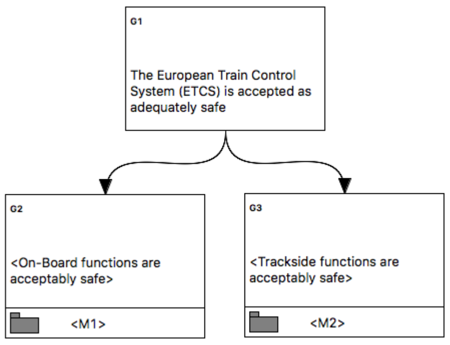

G1
The European Train Control System (ETCS) is accepted as adequately safe
G2
<On-Board functions are acceptably safe>
<M1>
G3
<Trackside functions are acceptably safe>
<M2>

Fig. 37: The Module View of ACME.

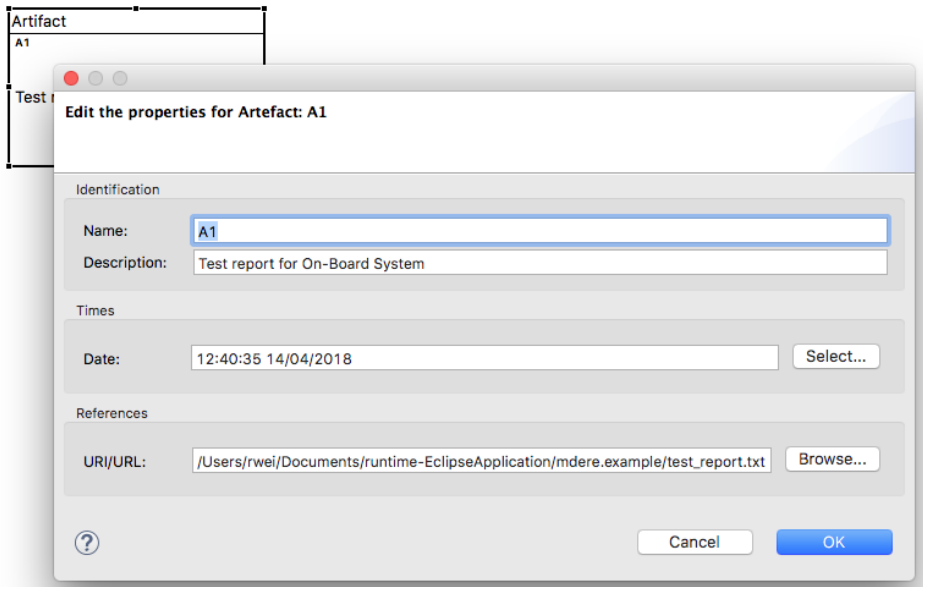
ACME is implemented using the Graphical Modelling Framework (GMF) [25], which supports the creation of editors based on metamodels defined using the Ecore metamodel provided by the Eclipse Modelling Framework (EMF) [26]. As previously discussed, the GSN metamodel inhertis all elements in the SACM metamodel, which means that the users of ACME are able to create GSN elements as well as elements inherited from SACM.

Figure 36 shows the Assurance Case Package View, within which the users are able to create Modules and Contract Modules of GSN, as well as elements defined in the *AssuranceCase* package of SACM. Figure 37 shows the Module view of ACME, where the users are able to create GSN elements.

The users are also able to create elements in the *Artifact* package, Figure 38 demonstrates how an *Artifact* can be created/edited. Creation tools are also provided for the elements in the *Terminology* package, where a *TerminologyPackage* is represented as a table in ACME, Figure 39 demonstrates how a *Terminology-Package* and its contents can be created/edited.

ACME eases the creation of assurance cases in the sense that the argumentation made in the assurance case can be modelled using GSN which system assurance engineers are more familiar with (in the sense that the models produced

[13] Available upon request.



Edit the properties for Artefact: A1

Identification

Name:	A1
Description:	Test report for On-Board System

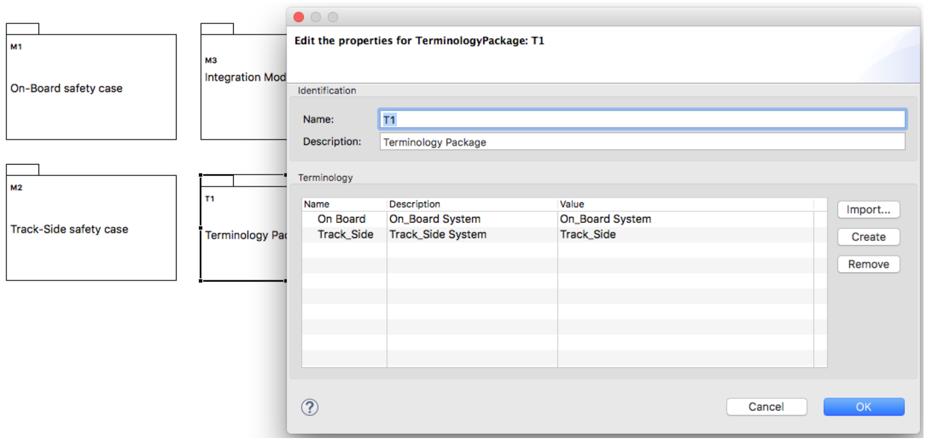
Times

Date:	12:40:35 14/04/2018

References

URI/URL:	/Users/rwei/Documents/runtime-EclipseApplication/mdere.example/test_report.txt

Fig. 38: Creation of Artifact.



Edit the properties for TerminologyPackage: T1

Identification

Name:	T1
Description:	Terminology Package

Terminology

Name	Description	Value
On Board	On_Board System	On_Board System
Track_Side	Track_Side System	Track_Side

Fig. 39: Editing a *TerminologyPackage*.

by ACME conform to the GSN metamodel described in Section 6). In addition, based on the transformation from GSN to SACM, ACME provides a conversion mechanism, where the created GSN model using ACME can be transformed into a SACM model, with the help of model-to-model transformation (implemented in Epsilon Transformation Language [24]), as shown in Figure 40.

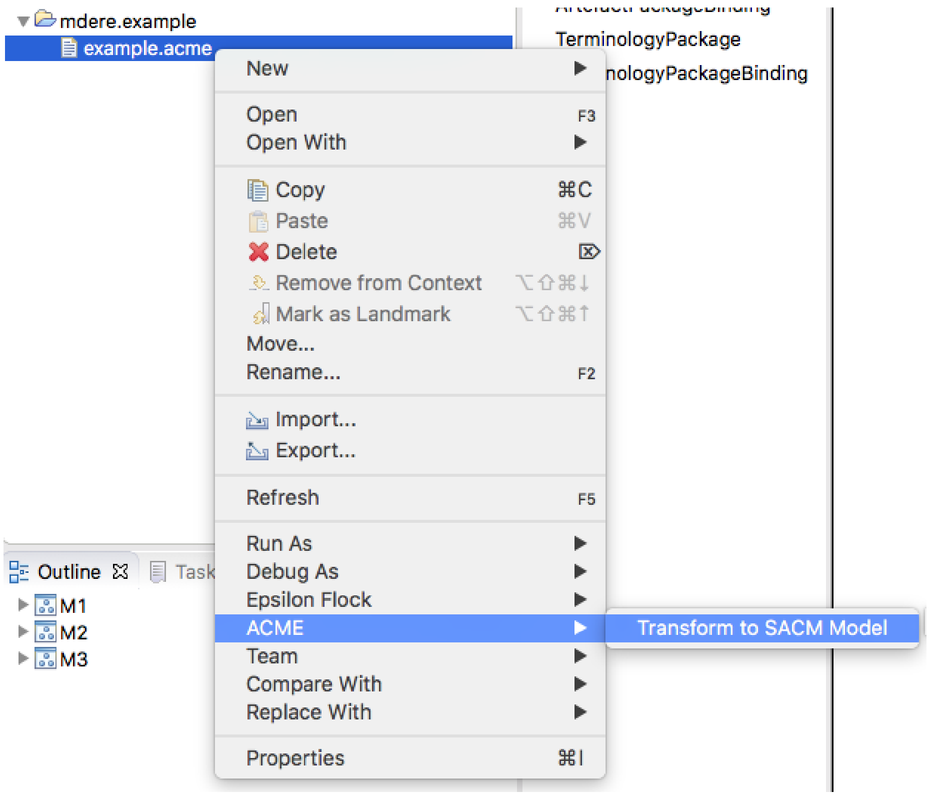


Fig. 40: M2M function provided by ACME to transform a GSN model to a SACM model.

ACME provides a transitional solution to model-based system assurance in the sense that it enables GSN users to fully exploit the potentials of SACM whilst being able to adopt the tool with their existing knowledge in using GSN.

7.2 Future Work

With regard to future work, our priority will focus on the support of importing legacy GSN/CAE models created by other drawing tools, typically those that can be consumed by computers but are not model-based. ACME integrated the

Epsilon platform[14] with it, which provides a set of extensible model management languages, as well as extensible support for models defined in various formats. Thus, as long as legacy GSN/CAE models can be consumed by computers, corresponding *driver*s can be created for the Epsilon platform to process such artefacts and transform them into GSN/SACM models via chaining transformations.

To the best of our knowledge, OMG is attempting to standardise the concrete syntax for SACM, we will provide support to the concrete syntax in ACME by creating Argumentation editors for SACM's *Argumentation* package. We will also investigate automated model validation to validate GSN/SACM models which has not been investigated before. We have not provided support for instantiating argumentation patterns, we will investigate the use of *ImplementationConstraint*s for pattern instantiation.

8 Conclusion

In this paper, we identified the importance of SACM for its role in model-based system assurance, SACM is more powerful than existing system assurance approaches (such as GSN and CAE) in the sense that it enables the users to store evidential, contextual information and controlled grammar within the model and reference them from the argumentation of the assurance case. We also provided a definitive exposition of SACM to explain its intended usage via examples. SACM has been sufficiently explained in this paper although extensive examples cannot be fully provided.

We also provided our version of GSN and CAE metamodels, which are compliant to SACM in the sense that users of these metamodels are able to utilise the facilities provided by SACM whilst still using GSN/CAE notations that they are familiar with. We also provide comprehensible model-to-model transformations from GSN/CAE to SACM to enable the interoperability from GSN/CAE to SACM.

We also briefly discussed the Assurance Case Modelling Environment (ACME) - a graphical modelling tool created based on our version of the GSN metamodel. ACME acts as a transitional solution from conventional GSN diagram creation to SACM model-based system assurance. ACME provides easy-to-use support for SACM facilities, as well as automated model-to-model transformation from GSN to SACM.

SACM provides a solid foundation for model-based system assurance, which provides means of system assurance for emerging technology domains such as Cyber-Physical Systems, autonomous systems, and Internet of Things, for its support for runtime system assurance, automated system assurance integration and automated reasoning for assurance argumentations.

Acknowledgements This work is supported by the European Union's Horizon 2020 research and innovation programme through the DEIS project (grant agreement No 732242).

[14] https://www.eclipse.org/epsilon/

References

1. Health Foundation. *Using safety cases in industry and healthcare.* December 2012.
2. Richard Hawkins, Ibrahim Habli, Tim Kelly, and John McDermid. Assurance cases and prescriptive software safety certification: A comparative study. *Safety science*, 59:55–71, 2013.
3. Tim Kelly and Rob Weaver. The goal structuring notation–a safety argument notation. In *Proceedings of the dependable systems and networks 2004 workshop on assurance cases*, page 6. Citeseer, 2004.
4. Claims, Arguments and Evidence (CAE). https://www.adelard.com/asce/choosing-asce/cae.html. Accessed: 06-04-2018.
5. Health and Safety Executive (HSE). *Safety Assessment Principles for Nuclear Facilities.* 2006.
6. UK Ministry of Defence (MoD). *Defence Standard 00-56 Issue 4: Safety Management Requirements for Defence Systems.* 2007.
7. Safety Regulation Group Civil Aviation Authority (CAA). *CAP 670 - Air Traffic Services Safety Requirements.* 2007.
8. Rail Safety and Standards Board. *Engineering Safety Management (The Yellow Book).* 2007.
9. Paul Chinneck, David Pumfrey, and Tim Kelly. Turning up the heat on safety case construction. In *Practical Elements of Safety*, pages 223–240. Springer, 2004.
10. Adelard Safety Case Editor (ASCE). https://www.adelard.co.uk. Accessed: 06-04-2018.
11. Integrated Safety Case Development Environment (ISCaDE). http://www.iscade.co.uk/. Accessed: 06-04-2018.
12. University of York Freeware Visio Add-on. http://www.cs.york.ac.uk/ tp-k/gsn/gsnaddoninstaller.zip. Accessed: 06-04-2018.
13. CertWare, NASA. https://nasa.github.io/CertWare/. Accessed: 06-04-2018.
14. Astah GSN Editor. http://astah.net/editions/gsn. Accessed: 06-04-2018.
15. Ari Jaaksi. Developing Mobile Browsers in a Product Line. *IEEE software*, 19(4):73–80, 2002.
16. Juha Kärnä, Juha-Pekka Tolvanen, and Steven Kelly. Evaluating the Use of Domain-Specific Modeling in Practice. In *Proceedings of the 9th OOPSLA workshop on Domain-Specific Modeling*, 2009.
17. Robin Bloomfield and Peter Bishop. Safety and assurance cases: Past, present and possible future–an adelard perspective. In *Making Systems Safer*, pages 51–67. Springer, 2010.
18. Samantha Lautieri, David Cooper, and David Jackson. Safsec: Commonalities between safety and security assurance. In *Constituents of Modern System-safety Thinking*, pages 65–75. Springer, 2005.
19. Structured Assurance Case Metamodel, Object Management Group. https://www.omg.org/spec/SACM/About-SACM/. Accessed: 06-04-2018.
20. Ran Wei, Tim P Kelly, Richard Hawkins, and Eric Armengaud. Deis: Dependability engineering innovation for cyber-physical systems. In *Federation of International Conferences on Software Technologies: Applications and Foundations*, pages 409–416. Springer, 2017.
21. Mario Trapp, Daniel Schneider, and Peter Liggesmeyer. A safety roadmap to cyber-physical systems. In *Perspectives on the future of software engineering*, pages 81–94. Springer, 2013.

22. Tim P Kelly and John A McDermid. Safety case construction and reuse using patterns. In *Safe Comp 97*, pages 55–69. Springer, 1997.
23. Richard Hawkins, Ibrahim Habli, Dimitris Kolovos, Richard Paige, and Tim Kelly. Weaving an assurance case from design: a model-based approach. In *High Assurance Systems Engineering (HASE), 2015 IEEE 16th International Symposium on*, pages 110–117. IEEE, 2015.
24. Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. The epsilon transformation language. In *International Conference on Theory and Practice of Model Transformations*, pages 46–60. Springer, 2008.
25. Graphical Modeling Framework. https://www.eclipse.org/gmf-tooling/. Accessed: 06-04-2018.
26. Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.