Assuring Dependable and Critical Systems: Implementing the Standards for Assurance Cases with ACedit

George Despotou, Aris Apostolakis, Dimitris Kolovos

Department of computer Science University of York, UK, george@cs.york.ac.uk
Software Engineer/Developer, aris.apostolakis@gmail.com
Department of computer Science University of York, UK, dkolovos@cs.york.ac.uk

Abstract. Assurance cases refer to the explicit documentation of claims about a system, and explanation of how these claims are convincingly supported by evidence generated during development and testing of that system. The purpose of an assurance case is to capture and communicate the rationale as to why one should have confidence in the operation of a system, and are chiefly used in domains where the operation of a system may have critical implications (e.g. safety or security). The Goal Structuring Notation (GSN) and the OMG Argumentation Metamodel (ARM) are two known approaches for constructing assurance cases. This paper introduces the basic concepts of assurance cases and presents a the Assurance Case Editor (ACedit)[1], a tool implementation of the two approaches based on eclipse. The tool supports the graphical construction of assurance cases in GSN and ARM and a number of model management techniques that apply to them.

Keywords: Assurance case, safety case, dependability case, security case, argumentation, GSN, ARM

1 INTRODUCTION

Assurance cases have evolved from the concept of safety cases, which exist to communicate an argument. They are used to demonstrate how someone can reasonably conclude that a system is acceptably safe from the evidence available [1]. Although in certain instances (e.g. the defence and the nuclear domain) a safety case is a legal or regulatory requirement, they have been recognized as a useful concept contributing to the quality and assurance of a system. This has resulted in the concept being introduced in various domains; examples include clinical safety cases for medical IT [2], security cased [3], reliability and maintainability cases [4], and dependability cases [5]. Although when generalizing the concept, definitions may bear differences in the

[1] The tool can be found at http://code.google.com/p/acedit/

various domains, all definitions converge. Over the last decades, systematic approaches to express assurance cases were introduced. The Goal Structuring Notation (GSN) and the Argumentation Metamodel (ARM) are two examples of such approaches. GSN is a graphical notation in which the elements of a case and their relationships can be expressed [1]. ARM, an OMG standard, has been specified in response to the need for safety and security cases, and its basic elements can be traced to GSN, having also been influenced by another safety case notation, the Claims-Argument-Evidence (CAE) notation [6].

2 THE ELEMENTS OF AN ASSURANCE CASE

An assurance case exists to communicate an argument, supported by evidence, about the assurance one can have on the operation of a system. A case consists of an argument and evidence supporting it. An argument is described as being "a connected series of statements or reasons intended to establish a position", and it consists of [6, 1]:

Claims: Claims represent statements about the system made at various abstraction levels. The highest level claim (e.g. system X is acceptably dependable) assumes the role of the overall position of the argument. Consequent claims establish (at different abstraction levels) specific positions about the system's operation (e.g. system X implements authentication, subsystem Y responds in timely manner), which contribute to establish the overall argument. Claims are true/false propositions capturing assertions that can be made about the system, In contrast to 'traditional' requirements, which make should/must demands. Furthermore they represent intent rather than a specification of the system.

Inferences: Claims are interconnected with each other, based on inference rules, forming a network. An inference is described "as the drawing of a conclusion from known or assumed facts or statements" [1]. In the case of argumentation, the known or assumed facts being the existing claims about the system, and the conclusions being the new claims about the system (or child claims). Evolution of an argument usually takes place until the claims can be directly supported by the available evidence.

Evidence: Evidence can be any relevant information from which a claim can be directly inferred. For example, statistical data can be evidence for a claim about a certain probability. Evidence can be diverse, including information such as statistical data, design reviews, testing data, expert opinion, and formal proofs. Insufficiency of evidence to support the developed argument structure reveals the need for activities that will produce additional evidence (e.g. additional system testing). Both argument and evidence are crucial elements of a safety case. "*An argument without evidence is unfounded whereas evidence without argument is unexplained*" [1]. A set of evidence does not constitute by itself a safety case. Argumentation can be used to capture any position about the system that the stakeholders choose to communicate, irrespective of the framework that was followed to establish this position. Establishing a (safety)

case can provide a number of advantages, regardless of it being imposed by standards [7].

Evolving in parallel to the system development, the argument claims are decomposed until they can be directly supported by evidence collected during the development and testing phases of the system. An assurance case conveys a degree confidence in the claims stated in it, depending on the quality of the argument and the strength of the evidence encompassed[2]. Creation of a case usually goes hand in hand with system development. A good case will provide the means to evaluate the fitness of the system to achieve the stated claims (e.g. that it is sufficiently secure), and the system enables the evaluation of the realism of the case. For example, a case making overly ambitious claims will not be easily supported by the evidence generated.

3 ASSURANCE CASE REPRESENTATION

A case is a conceptual entity initially conceived in the minds of the developers [8]. In order to represent the case, a means of capturing the underlying reasoning is required. GSN and ARM are two approaches used to represent the assurance case.

The Goal Structuring Notation (GSN) is a framework used to capture and graphically represent assurance arguments, and evolved over 15 years ago. Its pedigree can be predominantly traced in the safety domain, however it can represent any argument supported position. Lately it has been used in security and dependability case. GSN was mainly motivated by the inadequacy of plain text to clearly and traceably represent the inferences in an argument 1]. Throughout the years, GSN has been refined and has been optimised to provide sufficient concept richness to depict the concepts that constitute an assurance case.

GSN has been defined as part (a sub-package) of the Dependability Case Metamodel (DCM) [7], which has also been used by this tool. The latter defines a number of concepts needed to analyse the system and establish a case, whereas the former defines the concepts necessary to capture and depict the assurance argument. As of 2011, the GSN working group has published the GSN standard offering explanation of the concepts and guidance [9][3].

Two GSN extensions offer provisions for reusing arguments (the concepts of which are also implemented by ACedit):

- *Patterns*: Patterns provide the concepts to capture a generalised argument. Instantiating the argument for the system in question and in its context, will sys-

[2] There are a number of attributes that affect the quality of an argument and the strength with which the available evidence supports it, but they are not in the scope of this paper. The guidance in GSN standard and the GSN working group website (goalstructuringnotation.info) are good starting points to find more resources on what makes a good argument.

[3] The tool implemented is to the best of the authors' knowledge compatible with the draft version of the standard as of July 2011. The published version (November 2011) of the standard may contain very minor differences with the implemented metamodel.

tematically guide the developers through the considerations of the argument (i.e. the particular hazards for the system).

- *Modular GSN*: Modular extensions allow cohesive arguments to be captured as argument modules, which can be referenced from other argument modules. Contracts document the dependencies between the arguments. Modular arguments benefit the maintenance of the case as often a change can be contained within an argument module.

The Argumentation Metamodel (ARM) is an adopted standard by the OMG, and is the product of the OMG's system assurance task force (http://sysa.omg.org/) [10]. ARM has been influenced by GSN as well as another case representation approach (not implemented by (ACedit), the Claims Argument Evidence notation (CAE) [6]. The initial purpose of ARM was to represent the argument part of an assurance case, leaving capturing of evidence to the Structured Assurance Evidence Metamodel (SAEM); it was the initial intent that SAEM and ARM would work together to capture an assurance case. ARM and SAEM do have some overlap, with the former able to capture basic concepts regarding evidence. A complete separation would make ARM impractical, as it would not be able to represent evidence. More recent work by the system assurance task force has resulted in consolidating the two metamodels in one; the Structured Assurance Case Metamodel (SACM), which however is not yet an adopted standard.

4 TOOL ARCHITECTURE

The tool is implemented as a set of eclipse plug-ins. The tool implements a graphical user interface which consists of the GSN and ARM editors and a number of model management tools. ACedit consists of a number of layers. At the bottom lies the core framework layer of the tool. The tool can be thought of as a set of plug-ins on top of the functionality offered by the core eclipse framework. The second layer consists of the eclipse plug-ins that were used to implement the tool. ACedit uses the Eclipse Modelling Framework (EMF), the Graphical Modelling Framework (GMF), the Graphical Editing Framework (GEF), and Epsilon [11] The GMF and GEF plug-ins are utilized to implement the graphical user interface of the tool. EMF and Epsilon are responsible for the model development part. EMF was used for the specification of the metamodels and the generation the corresponding java code.

Epsilon was used to construct the plug-ins for model management tasks such as model validation and model transformation. Figure 1 shows a component diagram which describes the dependencies between the tool and the eclipse plug-ins. EuGENia [12] was used to automate the definition and generation of the graphical editor from the specification of the metamodel. The third layer consists of the tool plug-ins and metamodels. The metamodels (GSN and ARM) are fundamental parts of the tool, since the plug-ins on this layer are based on them. The plug-ins extend the eclipse framework and provide all the functionality for the tool. This layer includes the GSN plug-ins (responsible for the functionality of the GSN editor, which is based on the

GSN metamodel), the ARM plug-ins (responsible for the functionality of the ARM editors, which is based on ARM and the model management plug-ins which are responsible to provide functionality for model management tasks such as model-to-model transformation between the GSN and ARM metamodels.

Finally, the user interface layer, consists of the GSN editor, the ARM editor and the model management tools (GSN to ARM transformation, GSN validation and GSN wizards).

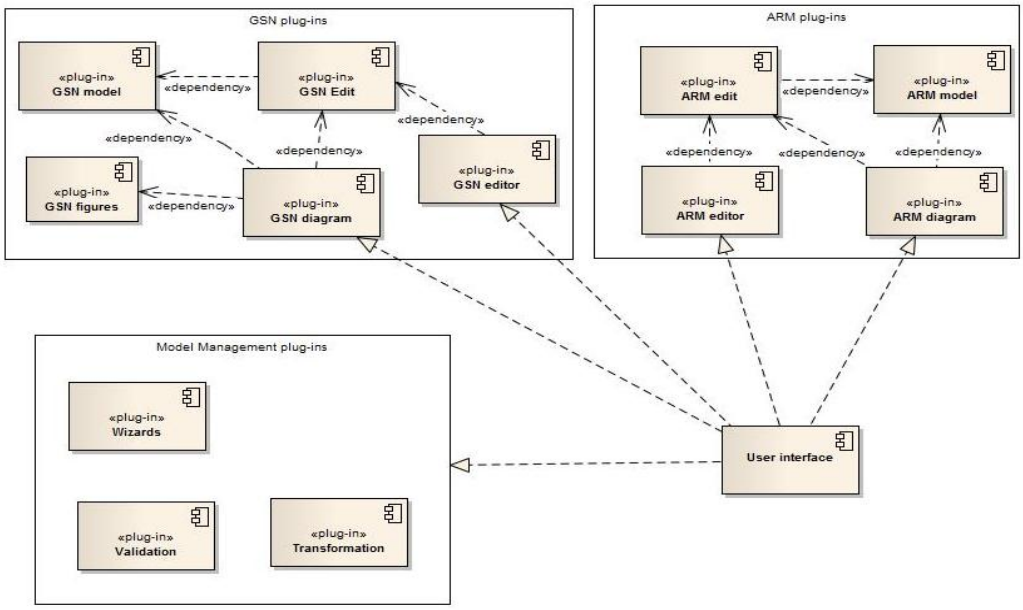

Figure 1 – Acedit Plug-ins (component diagram)

4.1 GSN and ARM Plug-ins

The GSN and ARM plug-ins are a set of plug-ins responsible for the visualization of the tool. The Edit, Editor, the model, and diagram plug-ins are generated using GMF and EuGENia tools. GSN has been implemented as a separated plug-in including the visualisation information of the generated GSN shapes.

The model plug-ins consist of the classes that implement the desired GSN and ARM metamodels. All the other plug-ins have dependencies on these. The edit plug-in consist of the classes enabling the access to the models. The editor plug-in is responsible for the representation of the model view. The diagram plug-ins consist of a set of packages and classes that process and combine the information of the model and edit plug-ins and implement the diagram code. Finally, the GSN plug-in consists of a set of Java classes which implement the rendering of the GSN elements. It should be noted, that the ARM standard does not include visualization specification. The shapes to visualize ARM were specified based on the similarity of concepts with

GSN. For example, an ARM claim was assigned the same shape (i.e. rectangle) with a GSN goal, as conceptually they are identical.

4.2 Model Management Plug-ins

The model management plug-ins implement the logic related with the model management tasks of the tool. Three different plug-ins have been implemented: the transformation plug-in, the validation plug-in, and the wizards plug-in. The plug-ins consist of two parts: the code that performs the actual model management tasks and the core part of the plug-in which integrates the plug-in with the tool at runtime.

The transformation plug-in is responsible for the conversion of GSN models to ARM models. It consists of a set of transformation rules[4] between the GSN metamodel and the ARM metamodel. Conversion from ARM to GSN models is possible, however this may need a number o assumptions to be made (or human input), as ARM generalizes on certain concepts of GSN. The validation plug-in consists of a set of constraints for the GSN metamodel[4]. The constraints are responsible to validate the user-created GSN model against the GSN standard.

The wizards' plug-in consists of a set of model refactoring functions for GSN diagrams. These support the corrections on how the GSN diagrams are displayed when changes are made to the model and when the user makes an action that is not compatible with the validation rules. For example,

4.3 GSN and ARM Editors

GSN and ARM editors are the result of the GSN and ARM plug-ins respectively. At this point the tool takes advantage of the GMF runtime. The tool inherits quality features from the GMF runtime (discussed in chapter 2) such as the properties views (which the user can modify the attributes of an element) and the animating zooming of a created diagram. These features apply to the diagram view of GSN and ARM.

The GSN and ARM editors consist of two views: the model view and the diagram view. In the diagram view, the user is able to create GSN and ARM diagrams. The diagrams are created according to the mapping between the metamodels elements and their graphical notation. In the model view, the user can create model instances of ARM and GSN metamodels respectively. The advantage of the model view is that elements of the metamodels that does have graphical representation can be created. The two views communicate by utilizing the Model View Controller (MVC) pattern. In the case of the GSN views, the MVC is implemented as follows:

- The model view, the model part of the MVC, is the GSN model instance which the user creates at runtime and derives from the GSN metamodel

[4] These can be found at the wiki pages of the tool (http://code.google.com/p/acedit/w/list)

- The diagram view, the view part, is the GSN diagram as presented by the editor's GUI. It can be represent two different views (module view and argument view) which are corresponding to the same model view.
- The controller is part of the GSN diagram plug-in code and refers to the EditPart package

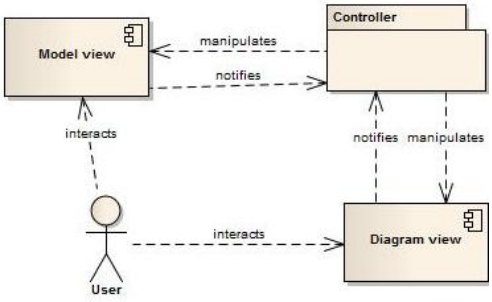

Figure 2 – ACedit packages

Figure 2 shows a component diagram with the MVC pattern as adapted in the project. The diagram shows how the user interacts with the diagram view component and makes a modification. Diagram view notifies the controller which manipulates appropriately the diagram and model view respectively. In the case of the ARM views the MVC pattern is utilized exactly the same like the GSN views.

5 MODEL MANAGEMENT FEATURES

Even simple system can include very big models of several thousand object. A large part of these objects may be possible to be automatically generated. Also it should be noted that use of the tool does not result in more objects and therefore in more effort. When assuring the operation of a system, engineers will need to either explicitly or implicitly reason about the properties of the system, design information (and evidence) collection tasks (e.g. testing), and document results. Structured argumentation and assurance cases provide the opportunity to engineers to explicitly document and trace reasoning and information that is generated (or considered) regardless. This is particularly useful for critical systems, for which implicit or undocumented reasoning or information may result in loss of traceability and ultimately in uncertainty about its operation. Nevertheless the potential large number of objects raises challenges that can benefit from model management features such as validation of the models, retrieval of objects as well as refactoring of the model.

From the early stages of the tool development it was obvious that model management techniques are valuable. Model management is used to implement the following functionality: transformations, validation, and widgets. In addition to the included rules, users can implement their own rules and share with them with the assurance

case community (the existing rules as well as links to the community's outlets can be found at the tool's wiki).

5.1 Transformations

One of the main objectives of the tool was to support transformation of GSN models to ARM models. This was achieved using the Epsilon Transformation Language (ETL) [13] to transform a created GSN model to a corresponding ARM model. The rules of the transformation have been specified based on the similarity of concepts between the two metamodels (and can be found at the tool's wiki page).

```
rule Goal2Claim
      transform s:GSN1!Goal
      to t : ARM!Claim{
      t.identifier:=s.identifier;
      t.description:=s.description;
      s.addIsTagged(t);

      if(s.toBeDeveloped==true){
            t.toBeSupported:=true;
      }
      else{
            t.toBeSupported:=false;
      }

      }
```

Figure 3 – Example ETL rule from GSN to ARM

Figure 3 presents an extract of the ETL transformation. This rule defines how a Goal element of the input source GSN model will create a claim element of the ARM model. The *rule Goal2Claim* denotes the name of the rule. The rule defines the source metamodel element (GSN Goal) and the target metamodel element (ARM Claim). The rest of the code it refers to the logic of the rule. For example, the identifier of the source model will be transformed to the identifier of the target model. The user can create a GSN diagram and then transform the model at runtime. A popup menu is displayed with the option GSN2ARM. When the user selects the transformation, a new ARM diagram and model file will be created in the same folder.

5.2 Model validation

Model validation is used to validate a created GSN diagram against the specification in the standard. The rules are defined using the Epsilon Validation Language (EVL) [14].

The constraints in the tool were used to capture additional constraints in a GSN model, which the standard itself does not include. It should be noted that constraints (what can and cannot be done in a model) complement the constraints caused by the structure (classes and associations) of the metamodel. A number of EVL constraints

could have been avoided by changing the structure of the metamodel. However, this would reduce the flexibility of the metamodel to evolve based on feedback from users in other than the traditional domains. This is natural as all languages evolve based on feedback from users and application in various domains. Changes to the structure of the metamodel may have greater impact to any tool. During specification of both the ARM and the GSN metamodels, a degree of flexibility was intentionally added which was then constraint using EVL rules, to reflect the prevailing consensus in practice.

Some of the EVL rules incorporated suggestions for specific fixes of validation errors. This improves the usability of the tool and the correctness of the generated models. For example, in the case of the away goal (when creating a modular GSN argument – please see the GSN standard for an explanation of the particular concepts), the user has to specify the *id* of the module that the away goal belongs to. Since module id's type is String, it is possible that a user may specify an id for a module that does not exist (Figure 4).

The keyword *context* specifies the element of the metamodel on which the contained constraints will be evaluated (*AwayGoal* in this case). The keyword constraint defines the name of the constraint and a body on which the invariants will be included. The keyword check defines the invariant to be evaluated by the constraint (in this case is the module *id* that the user gave exists). The message specifies the error message which is related with the failed invariant, followed by the fix which contains the title of the recommendation to fix the mistake and the do part, which provide the functionality of correcting the user's mistake. In the above case, the user is prompted to choose one of the available modules and the appropriate identifier is assigned to *AwayGoal*.

```
context AwayGoal{
      constraint correctModuleId{
            check : self.moduleIdExists(self.moduleIdentifier)
            message : 'Does not exist any module with identifier: ' +
self.moduleIdentifier + ' '
            fix{
                  title:'Change the name of the moduleIdentifier'
                  do{
                        var target := UserInput.choose('Select Module: ',
                        Module.allInstances());
                        if(target.isDefined){
                              self.moduleIdentifier:=target.identifier;
                        }
                  }
            }
      }

}
```

Figure 4 – Example of an EVL rule

The user can use the fix part of the constraint. User has to press right click on the error and to choose from the popup menu the option quick fix. A dialog appears in which the user can choose one of the available fixes for the validation error. When the

user selects the recommended fix and press the finish button another dialog appears that prompts the user to choose one of the available modules.

5.3 Model Wizards

This section presents the model wizards tool. This tool takes advantage of the built-in GMF mechanism of GMF and the Epsilon Wizards Language (EWL) [15] in order to offer in-place model transformation functionality for GSN diagrams through the wizards mechanism of GMF.

```
wizard visibility {

      guard : self.isKindOf(Goal)or self.isKindOf(Solution) or
      self.isKindOf(Context)

      title : 'change visibility from ' + self.visibility +' to ' +
      self.returnOppositeVisibility()

      do {
            if(self.visibility==ElementVisibility#Public){
                  self.visibility=ElementVisibility#Private;
            }
            else if (self.visibility==ElementVisibility#Private){
                  self.visibility=ElementVisibility#Public;
            }
      }
 }

operation ModelElement returnOppositeVisibility():ElementVisibility{
      if(self.visibility==ElementVisibility#Public){
            return ElementVisibility#Private;
      }
      else if (self.visibility==ElementVisibility#Private){
            return ElementVisibility#Public;

      }
```

Figure 5 – An example of a EWL wizard

Users can add new wizards to the GSN editor by only changing the EWL file. EWL is used to extend the wizards built in mechanism. The wizards implement in-place model refactoring in the GSN elements by changing their values. Figure 5 shows an extract of the EWL code. This wizard is responsible to change the visibility attribute of a GSN element. Visibility applies to Goal, Solution and Context element and this is indicated in the guard part of the wizard. The title part indicates the text will appear as the wizard title. In this example the text is dynamic and depends on the current status of the element. The operation *returnOppositeVisibility* helps to print the correct message. Finally, the do part of the wizard specifies the action of the wizard which in this case will change the visibility attribute of the selected element.

6 SUMMARY

An assurance case exists to communicate an argument, supported by evidence, about the assurance one can have on the operation of a system. In order to represent the case, a means of capturing the underlying reasoning is required. GSN and ARM

are two approaches used to represent the assurance case. ACedit implements and visualises an EMF version of the specification of the two standards as a GMF editor. ACedit offers model management infrastructure, and includes a number of rules often used by assurance case users including validation and transformation between the two standars.

REFERENCES

1. Kelly T., Arguing Safety, a Systematic Approach to Managing Safety Cases. PhD Thesis, Department of Computer Science, University of York, 1998.
2. Despotou G., Kelly T., White S., Ryan M., Introducing Safety Cases for Health IT, in proceedings of 4th International Workshop on Software Engineering in Health Care, 4-5 June 2012
3. Alexander R., Hawkins R., Kelly T., Security Assurance Cases: Motivation and the State of the Art, Report Number: CESG/TR/2011/1.
4. Def Stan 00-42: "Reliability and Maintainability (R&M) Assurance Guide": Part No: 3: R&M Case: Issue 3, UK Ministry of Defence.
5. Despotou G., Kolovos D., Paige R., Kelly T., Defining a framework for the development and management of dependability cases. In proceedings of the 26th International System Safety Conference, Vancouver, Canada
6. Bishop P., Bloomfield R., A Methodology for Safety Case Development, Proceedings of the Sixth Safety-critical Systems Symposium, February 1998
7. Despotou G., Kelly T., The Dependability Case as a means of Establishing System Assurance, poster presentation, 26th International System Safety Conference (ISSC), Vancouver, Canada. August 2008.
8. Habli I., Kelly T., Safety Case Depictions vs. Safety Cases - Would the Real Safety Case Please Stand Up? proceedings of the 2nd IET International Conference on System Safety
9. GSN Working Group, GSN Community Standard v.1. November 2011, http://www.goalstructuringnotation.info/archives/76
10. Object Management Group (OMG), Argumentation Metamodel (ARM) (FTF - Beta 1), http://www.omg.org/spec/ARM, August 2010
11. Kolovos D., An Extensible Platform for Specification of Integrated Languages for Model Management, PhD thesis, Department of Computer Science, University of York.
12. Eclipse foundation, EuGENia GMF Tutorial, http://www.eclipse.org/epsilon/doc/articles/eugenia-gmf-tutorial/
13. Kolovos D., Paige R., Polack F.,. The Epsilon Transformation Language, in Proc. 1st International Conference on Model Transformation, ICMT'08, Zurich, Switzerland, July 2008
14. Kolovos D., Paige R., Polack F., Detecting and Repairing Inconsistencies Accross Heterogeneous Models, in Proc. 1st IEEE International Conference on Software Testing, Verification and Validation, ICST 08, Lillehammer, Norway, April 2008
15. Kolovos D., Paige R., Polack F., Rose M. L., Update Transformations in the Small with the Epsilon Wizard Language, in Journal of Object Technology (JOT), Vol. 6, No. 9, Special Issue. TOOLS EUROPE 2007, October 2007