
Proceedings Paper:
Hawkins, Richard, Habli, Ibrahim, Kolovos, Dimitrios S. et al. (2015) Weaving an Assurance Case from Design: A Model-Based Approach. In: 16th IEEE International Symposium on High Assurance Systems Engineering, HASE 2015, Daytona Beach, FL, USA, January 8-10, 2015. IEEE, pp. 110-117.

https://doi.org/10.1109/HASE.2015.25

Weaving an Assurance Case from Design: A Model-Based Approach

Richard Hawkins, Ibrahim Habli, Dimitris Kolovos, Richard Paige, Tim Kelly
Department of Computer Science
The University of York
York, UK
{Richard.Hawkins, Ibrahim.Habli, Dimitris.Kolovos, Richard.Paige, Tim.Kelly}@york.ac.uk

***Abstract*— Assurance cases are used to demonstrate confidence in properties of interest for a system, e.g. for safety or security. A model-based assurance case seeks to bring the benefits of model-driven engineering, such as automation, transformation and validation, to what is currently a lengthy and informal process. In this paper we develop a model-based assurance approach, based on a weaving model, which allows integration between assurance case, design and process models and metamodels. In our approach, the assurance case itself is treated as a structured model, with the aim that all entities in the assurance case become linked explicitly to the models that represent them. We show how it is possible to exploit the weaving model for automated generation of assurance cases. Building upon these results, we discuss how a seamless model-driven approach to assurance cases can be achieved and examine the utility of increased formality and automation.**

Keywords-assurance cases, safety cases, arguments, model-driven engineering, weaving

I. INTRODUCTION

Systems used to perform critical functions require justification that they exhibit the necessary properties (such as for safety or security). Assurance cases provide an explicit means for justifying and assessing confidence in these critical properties. In certain industries, specifically in the safety-critical domain, e.g. defence, aviation, automotive, and nuclear, it is a regulatory requirement that a safety or an assurance case is developed and reviewed as part of the certification process [1]. An assurance case is defined as a "*reasoned and compelling argument, supported by a body of evidence, that a system, service or organisation will operate as intended for a defined application in a defined environment*" [2].

Assurance cases are typically represented textually, using natural language, or graphically, using structured notations such as the Goal Structuring Notation (GSN) [2] or Claims, Arguments and Evidence (CAE) [3]. In order to improve standardisation and interoperability, the Object Management Group (OMG) has recently specified and issued a Structured Assurance Case Metamodel (SACM) for the representation of assurance cases [4]. Both GSN and CAE conform to SACM.

However, the development and acceptance of assurance cases remain a significant challenge for engineers and assessors. Assurance cases are large and complex, with a great deal of explicit and implicit dependencies. In particular, the lack of integration with, and the limited traceability to, design artefacts can undermine confidence in the reasoning presented, and the evidence referenced, in the assurance case. In particular, the limited automated capability for both constructing and analysing the dependency of assurance cases on other engineering artefacts makes it difficult to navigate, check and maintain the assurance case arguments and evidence.

By exploiting metamodels and well-defined modelling languages, an assurance case representation can be treated as a model, potentially bringing the benefits of model-driven engineering, such as automation, transformation and validation, to what has been a lengthy and informal process. It also has the potential to improve traceability between the assurance case and the design and analysis models and therefore support the *coevolution* of the system design and the assurance case. This in turn can help highlight, in a timely manner, weaknesses in the design, evidence and potentially the argument.

In this paper we develop a model-based assurance approach, based on a weaving model, which allows integration between assurance case, design and process models and metamodels. We show how it is possible to exploit the weaving model for automated generation and analysis of assurance cases. Building upon these results, we discuss how a seamless model-driven approach to assurance cases can be achieved and examine the utility of increased formality and automation.

Specifically, the contributions of this paper are as follows:

a) A model-based approach to managing assurance case models;
b) A weaving model that allows interoperation;
c) A GSN metamodel that is compliant with SACM;
d) Automatic instantiation of assurance argument patterns.

This paper is organised as follows. Section 2 describes our approach. In section 3, we use a case study to illustrate how our approach is implemented. Section 4 describes how our approach relates to other work in this area. Section 5 presents conclusions based upon our work to date and discusses future development of the approach.

II. MODEL-BASED SAEETY CASES

Our approach is illustrated in Fig. 1. In the following sections, we discuss in detail each element of the approach.

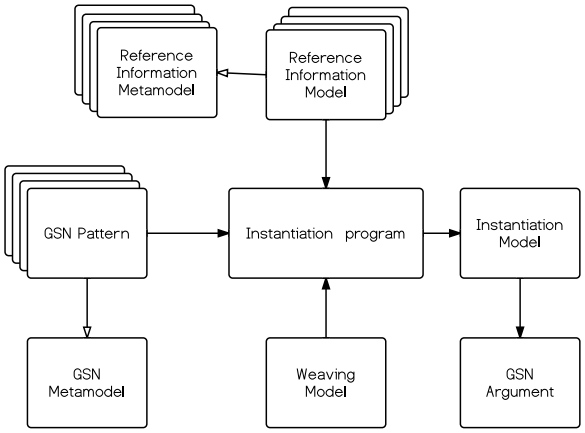

Figure 1. Overview of the Model-Based Assurance Case Approach

A. GSN Patterns

Patterns are widely used in software engineering as a way of abstracting the fundamental design strategies from the details of particular designs. The use of patterns as a way of documenting and reusing successful assurance argument structures was developed by Kelly [5]. Assurance argument patterns provide a way of capturing the required form of an assurance 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. Assurance argument patterns can be captured using GSN [2].

Assurance argument patterns essentially define requirements for the necessary information (e.g. to instantiate the assurance claims) and evidence (to support those claims). It is possible to manually obtain this information from design or analysis documentation, or directly from an engineer, and instantiate the argument patterns; this is current practice. Our approach instead uses the models themselves to automatically instantiate the patterns. Instantiation involves both instantiating '*roles*' in the argument patterns, and making instantiation choices.

Roles are instantiable entities within elements of the argument pattern. They represent an abstract entity that needs to be replaced with a concrete instance appropriate for the target system. For example in Fig. 2, the role within this assurance claim, represented in curled braces is '*Function*'. This entity must be replaced with the name of the relevant function of the system. In addition, argument patterns will often include multiplicity relations, where the number of required argument elements must also be determined (e.g. an entity created for each of the functions present in the system design).

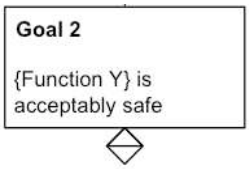

Figure 2. A GSN Argument Element Requiring Instantiation

Assurance argument patterns will also often represent choices for different argument approaches that may be adopted. At instantiation, the assurance claims most appropriate for the target system must be chosen from the options provided in the pattern.

Decisions on how to instantiate choices and optional elements within argument patterns are currently based on experience and judgement. Using a model-based approach allows consistent, reusable instantiation rules to be established, ensuring consistent instantiations. This will be particularly important where complicated relationships between multiple models are required.

Our approach considers the GSN argument patterns as models. As with all the models used, the argument pattern models must be consistent with an explicitly defined metamodel, and must be in a machine-readable format. We discuss the format of argument pattern models in Section 3. Below we address the issue of a metamodel for GSN.

B. GSN Metamodel

GSN argument patterns conform to the syntax and semantics defined by the GSN standard [2]. Currently the GSN standard does not define a GSN metamodel. However, as already discussed, an OMG standard metamodel for structured assurance cases (SACM) exists [4]. We have extended the SACM metamodel to include specific GSN entities as shown in Fig. 3.

For example, the notion of *Claim* in SACM is extended to cover *GSN_Goal*, *GSN_Assumption*, *GSN_Justificaton* and *GSN_ContextAsAssertion*. Further, the pattern notation of GSN is richer than that of the SACM core, so we have also introduced these additional pattern extensions into the metamodel in Fig. 3 (e.g. a new entity *GSN_Choice* and attributes *choice* and *multiplicity*). Our GSN metamodel, including a more detailed description, is to be included as a proposal in issue 2 of the GSN standard [2].

C. Reference Information Models

What we refer to as the reference information models are the set of models containing the information required for the instantiation of the assurance argument pattern. In theory, since the potential scope of assurance argument patterns is unbounded, so are the number and type of models that will be required. In practice however, the required information models will mainly be design, process and analysis models for the system. The particular models required can only be determined from the specific argument patterns themselves, with the information need being established from the roles of the instantiable elements.

The information models are expected to be diverse in nature, however our approach only requires that the models conform to a defined metamodel. Each model may have a different metamodel, and this must be explicitly defined such that it can be used in developing the weaving model (Section 2.D). Our approach places no restrictions on the tools and notations used to generate the models, which in turn maximises the information that becomes available for automatic pattern instantiation. For most argument patterns

multiple models are required to provide the instantiation information. Consider for example an argument pattern regarding the error behaviour of a component (Table 1). By picking out some of the key roles in the argument pattern we can see the different reference information models needed.

TABLE I. REFERENCE MODELS FOR DIFFERENT ROLES

Role	Reference Information Model
Component	AADL specification
Component error model	AADL error model specification
Error effect	FMEA analysis results
Stage	FMEA process model

There are a number of things to note here. Firstly, some of these models are of a type that is normally informally defined. For example the results of Failure Modes and Effects Analysis (FMEA) are normally captured as a simple table, and the FMEA analysis process will often be described using a textual description in a process document, or as a flow chart. Secondly, it is not necessary to have all the models relating to an argument pattern in machine-readable format in order to undertake automation of the instantiation. Those entities where the information model is available will be instantiated, while those entities where the model is not available will remain uninstantiated, requiring subsequent manual instantantiation. Thirdly, there is clearly an interrelationship between these four reference information models. For example, the failure modes and effects identified in the FMEA analysis results must relate to error types in the AADL error model. These inter-model relationships, although generally implicitly understood, are often not explicitly documented. As part of creating the weaving model (Section 2.D), our approach requires that these relationships be explicitly captured.

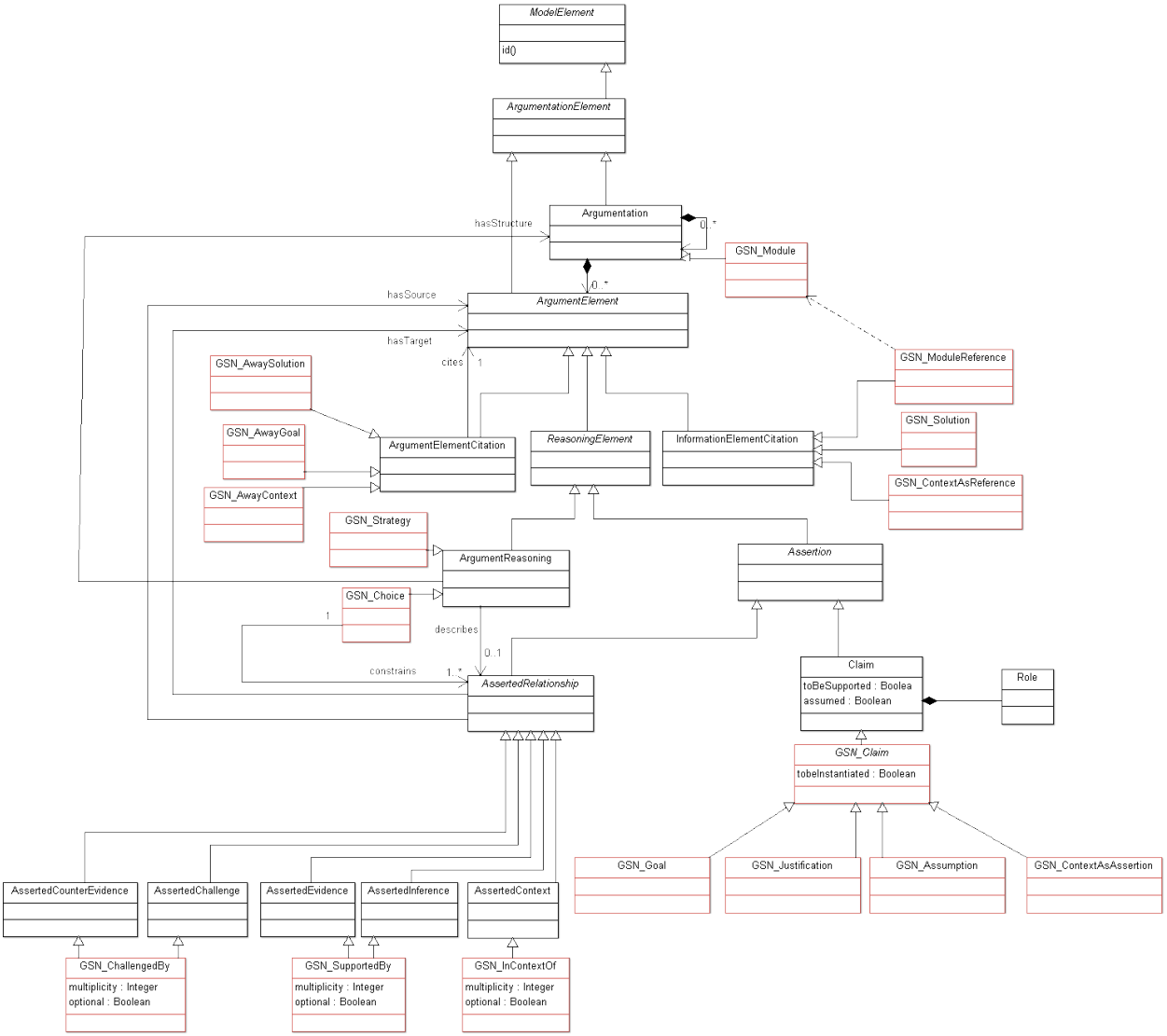

Figure 3. Proposed GSN Metamodel

D. Weaving Model

A weaving model is at the heart of our approach. It is the weaving model that links the reference information metamodels to the GSN argument patterns. Model weaving is an approach to model transformation defined by [8] as an operation "*whose primary objective is to handle fine-grained relationships between elements of distinct models, establishing links between them. These links are captured by a weaving model. It conforms to a metamodel that specifies the link semantics*". The weaving can be performed either manually, i.e. by linking the related elements by hand, or automatically, i.e. through a model transformation. An advantage of model weaving is that the mappings between the models themselves are also considered as models bringing expressiveness, flexibility and genericity [9].

In our approach a weaving model captures the dependencies between the roles in the GSN patterns and individual reference information metamodels and also between the multiple reference information metamodels. It is the dependency information captured in the weaving model that enables the argument instantiations to be automatically performed. We provide more details on the weaving model through an example in Section 3. It is always necessary to identify these dependencies when instantiating an assurance argument pattern.

Normally, however, such as when manually instantiating argument patterns, these dependencies are implicit or hardwired. Our utilisation of a weaving model makes this dependency information explicitly defined. The weaving model also enables the specification of mechanisms for capturing the more complex dependencies between models that are often required for an assurance argument (e.g. the dependencies between a fault model and the model of the analysis process used to generate it). There are existing approaches and tools, such as that described in [9] that support the creation of weaving models.

E. Model-Based Technology

Using a model-based approach allows us to take advantage of the extensive set of model management tools that are available. This brings the opportunity to harness the tools in order to quickly and easily add extra functionality and features. In particular we make use of the Epsilon family of languages and tools for model management [10]. Epsilon includes the following languages that are particularly useful with respect to assurance cases:

- **Epsilon Object Language (EOL)** is an imperative programming language for creating, querying and modifying models. We use EOL for the pattern instantiation program.
- **Epsilon Transformation Language (ETL)** is a hybrid, rule-based model-to-model transformation language. It can transform many input to many output models, and can query, navigate and modify both source and target models. ETL supports transforming diverse model inputs prior to use by the pattern instantiation program.
- **Epsilon Validation Language (EVL)** is a validation language that implements model constraints. EVL allows us to automatically check and enforce sets of constraints, both on the assurance arguments themselves, and also on the relationships between the argument and the reference information models.
- **Epsilon Generation Language (EGL)** is a template-based model-to-text language for generating code, documentation and other textual artefacts from models. We use EGL in order to generate the output of the instantiation program. EGL provides us with the flexibility to provide a number of options for the output format (e.g. GSN, text or tabular formats).

F. Instantiation

The instantiation program is an EOL program that runs on the Eclipse platform. It requires as input: GSN argument pattern models, reference information models and a weaving model. The instantiation program (Fig. 4):

1. identifies the elements requiring instantiation in the GSN argument pattern models;
2. determines which information from the reference information model is required to instantiate each GSN element by querying the weaving model;
3. obtains the required information from the relevant information models; and
4. outputs instantiation information.

For any information models that conform to a metamodel included in the weaving model, the instantiation program will identify the required information from those information models and perform the instantiation. To do this, the instantiation model uses the mappings in the weaving model in conjunction with the relationships between the entities in the metamodel, also defined in the weaving model.

```
for all nodes in GSN_Pattern_Models
  {if (tobeInstantiated=true)
    then instantiate(node)
  else
    add node to output GSNML file}

for all relationships in GSN_Pattern_Models
  {add relationship to output GSNML file}

instantiate(node)
  {if (relationship pointing to node is multiplicity)
    then makeMultiples(node)
  else
    makeInstance(node)} //for simplicity, this pseudo code does not consider choices

makeInstance(node)
  {for each node role
    {find target type of weaving model mapping for role
    for all elements of reference design model
      {if (element type=target type)
        extract information from reference design model element
        create node in output GSNML file using extracted information
        create a relationship pointing to created node in output GSNML file}}}

makeMultiples(node)
//number of instances required is determined from information from the
//reference design
  {identify the multiplicity role type for the relationship pointing to node
  find target type of weaving model mapping for multiplicity role type
  //for simplicity, this code does not describe how dependencies in the metamodel
  // are used to identify the relevant elements of the reference design model
  for all elements of reference design model
    {count if (element type=target type)}
  for 1 to count
    {instantiate(node)
    //the sub-structure below this must also be instantiated multiple times
    instantiate argument pattern model below node}}
```

Figure 4. Instantiation Pseudo Code

G. Instantiation Output

As a result of using the Eclipse framework, there are many different ways in which the instantiation program can output the instantiation information. It is important that the output is presented in a manner that is easily and clearly understood by a human. This is required for presentation as part of the assurance case, as well as for review purposes. It is also desirable for the output to be amenable to end-user editing to facilitate human instantiation of undeveloped parts of the assurance argument. The output argument instantiation model should therefore be independent of, but compatible with, existing GSN editor tools.

Our approach allows us to provide such an output in the form of a GSNML model file for the GSN instantiation model that is compliant with the GSN metamodel. This then allows representation and editing of the argument instantiation in manner familiar to the end-user.

III. Case Study

In this section, we use a security system to illustrate how our approach can be used to automatically generate an assurance argument directly from design models through the use of a weaving model.

A. System

The example we present is a software cryptographic controller system taken from [11]. The architecture for the system is shown in Fig. 5.

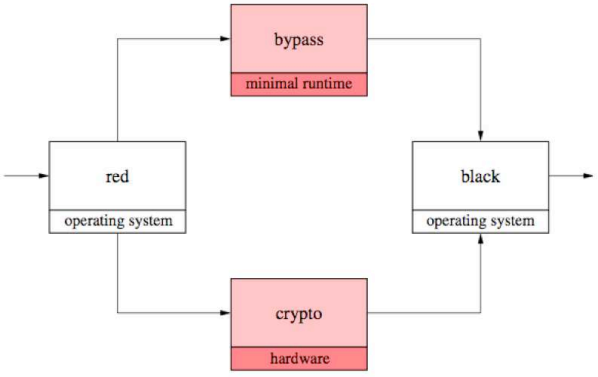

Figure 5. Software Architecture of Crypto Controller System

The system is a controller for end-to-end encryption. It takes inputs as clear text from the *red* network, encrypts the content of the message, and sends the encrypted message out on the *black* network. Inputs comprise both a header, which contains destination and other routing information, and the message content itself. Only the message content is encrypted since the *black* network has to read and process the headers so the message can be correctly routed to its destination.

For this cryptographic controller system, the overall security policy is that no unencrypted message content information shall be passed to the *black* network. The local policy for each of the components is that:

- crypto shall encrypt everything that leaves on its outgoing channel; and
- bypass shall ensure that only valid protocol headers are passed from red to black.

In this system the software in the *red* and *black* components can be completely untrusted (i.e. no assurance is required in these components in order to assure the overall security policy).

We wish to create an assurance case regarding the security of this system. The assurance argument pattern that can be used to structure the argument is shown in Fig. 7. This argument pattern is used to demonstrate the enforcement of the overall security policy through consideration of the enforcement of the locaNl security policy defined for each component in the system, as well as the composition of the components on the implementation platform (ensuring interference between components only occurs over defined paths). This pattern considers just the high-level structure of the argument. Further, more detailed patterns (not presented here) are available for creating arguments regarding the implementation of the components and their composition.

B. Implementation

We now provide a walk-through of applying our approach to instantiate the assurance argument pattern from Fig. 7 to create the high-level of the security argument for the cryptographic controller system.

Firstly we create a model of the assurance argument pattern that is conformant to the GSN metamodel presented in Fig. 3. The most straightforward approach is to use the Eclipse framework to generate a tree-based editor from the GSN metamodel and then use it to construct the models. The creation of GSN patterns is however a creative activity, often involving discussion and review with multiple stakeholders. In such situations, a graphical representation is normally desirable. To enable graphical model generation we developed from the GSN metamodel a graphical editor using GMF (Graphical Modelling Framework). Fig. 6 shows an extract from the resulting model in XML form (which we call GSNML files). This GSNML file was taken as input by the instantiation program.

```
<?xml version="1.0" encoding="UTF-8"?>
<gsnmetamodel:Case xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.
  <contains xsi:type="gsnmetamodel:GSN_Module" id="DMILS System Argument Pattern">
    <ArgumentElements xsi:type="gsnmetamodel:GSN_Goal" id="Goal: sysSecurity">
      <contents xsi:type="gsnmetamodel:Literal" literal="security policy is enforced"/>
      <contents xsi:type="gsnmetamodel:Role" role="DMILS System"/>
    </ArgumentElements>
    <ArgumentElements xsi:type="gsnmetamodel:GSN_ContextAsReference" id="Con: sysPolicy">
      <contents xsi:type="gsnmetamodel:Literal" literal="The system security policy is"/>
      <contents xsi:type="gsnmetamodel:Role" role="system security policy"/>
    </ArgumentElements>
    <ArgumentElements xsi:type="gsnmetamodel:GSN_InContextOf" hasSource="//@contains.0/@ArgumentE
    <ArgumentElements xsi:type="gsnmetamodel:GSN_ContextAsReference" id="Con: sysDescr">
      <contents xsi:type="gsnmetamodel:Role" role="DMILS system AADL model"/>
    </ArgumentElements>
    <ArgumentElements xsi:type="gsnmetamodel:GSN_InContextOf" hasSource="//@contains.0/@ArgumentE
    <ArgumentElements xsi:type="gsnmetamodel:GSN_Strategy" id="Strat: sysSecurity">
      <contents xsi:type="gsnmetamodel:Literal" literal="Argument over the individual software co
      <contents xsi:type="gsnmetamodel:Role" role="DMILS Systems"/>
    </ArgumentElements>
    <ArgumentElements xsi:type="gsnmetamodel:GSN_SupportedBy" hasSource="//@contains.0/@ArgumentE
    <ArgumentElements xsi:type="gsnmetamodel:GSN_Goal" id="Goal: components">
      <contents xsi:type="gsnmetamodel:Literal" literal="Trusted software components behave accor
    </ArgumentElements>
    <ArgumentElements xsi:type="gsnmetamodel:GSN_SupportedBy" hasSource="//@contains.0/@ArgumentE
```

Figure 6. Assurance Argument Pattern Model in GSNML (Partial)

For this example the required instantiation information can be obtained from the AADL [13] specification that has been created for the system, a small part is seen in Fig. 8. This is a straightforward exercise. Even, however, in cases where a

required information model is informally defined, such as the FMEA analysis process description mentioned earlier, it is often possible to generate simple XML representations sufficient to input to the instantiation program. As an example we have generated a process model from a flow-chart of the FMEA analysis process. The flow chart conforms to the generic process metamodel defined in [12].

We created a weaving model to capture the dependencies between the roles of the argument pattern model and the elements of the AADL meta-model (the AADL meta-model is defined in [13]). Fig. 9 shows a graphical representation of this weaving model. The left hand side shows the roles from the argument pattern, the right hand side represents the AADL meta-model, the horizontal arrows represent the mappings in the weaving model between roles in the argument pattern and elements of the AADL meta-model.

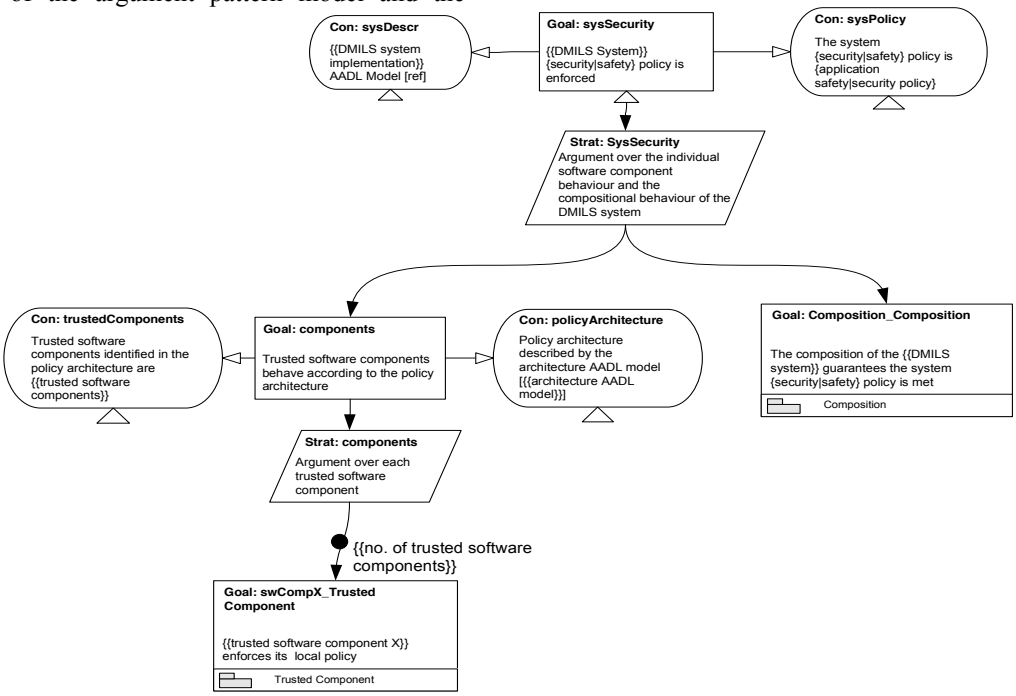

Figure 7. Assurance Argument Pattern

```
system CryptoController
  features
    inframe: in data port Frame;
    outframe: out data port Frame;
end CryptoController;
system implementation CryptoController.Imp
  subcomponents
    red: node Splitter.Imp accesses channels;
    bypass: node Bypass.Imp accesses channels;
    crypto: node Crypto.Imp accesses channels;
    black: node Merger.Imp accesses channels;
    channels: network CryptoNet.Imp;
  flows
    port inframe -> red.frame;
    port red.header -> bypass.inheader;
    port red.payload -> crypto.inpayload;
    port bypass.outheader -> black.header;
    port crypto.outpayload -> black.payload;
    port black.frame -> outframe;
end CryptoController.Imp;

--
-- Splitter component for decomposing frames int
--
node Splitter
  features
    frame: in data port Frame;
```

Figure 8 Extract of the AADL specification for the crypto controller system

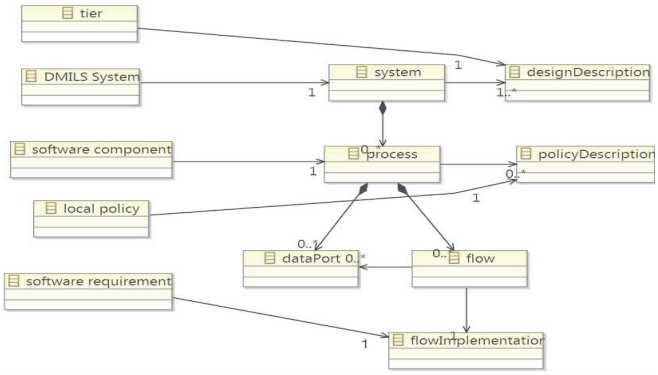

Figure 9. Representation of the Example Weaving Model

The models described above were used as input to our instantiation program. The instantiation program generated a complete model of the instantiated argument. This is provided as a GSNML file that includes the information for all the required argument elements and the relationships between the elements.

From this file the instantiated argument can be represented graphically using a graphical tool (e.g. our Eclipse based editor) or inputted to an existing GSN argument editor.

IV. Related Work

Our work builds on existing efforts for standardising the representation of assurance cases and providing automated means for analysis and reuse, particularly based on the OMG SACM standard [4] and the GSN standard [2]. Several tools now exist that comply with these standards (e.g. [3] [14] [15] [6]). There is also interest in extending the SACM standard in order to provide a modelling basis for machine-checkable assurance cases [16].

Khalil et al [17] present an argument pattern library that aims to define and automatically generate safety cases in a model-based development environment. However, they fall short of specifying the *means* by which a safety case model is integrated with design models and the *process* by which the instantiation of the safety argument patterns can be automated. The authors seem to be interested in applying their model-based approach to support modular certification [7], although without specifying the relationship between a modular architecture and its corresponding modular safety case.

Armengaud [18] discusses the automatic assembly of safety cases for automotive applications from information models generated from ISO 26262 work products (i.e. items of evidence such as design and test artefacts). However, the author does not address how these models inform, or relate to, the structure of the safety case argument.

At a more formal level, Rushby [19] promotes the use of formal verification techniques for the representation and analysis of assurance cases, with the aim of reaping the benefits of increased precision and "*pushbutton automation*" offered by recent advances in the development and use of formal techniques. Formalism here entails the definition of the safety argument using a mathematically based notation. This also involves defining a means for translating the structure and expression of safety cases into forms tractable for automated formal analysis.

Rushby makes a distinction between two aspects of assurance arguments [20]: (1) *logic doubt* that relates to the reasoning in the argument and (2) *epistemic doubt* that relates to our understanding of the system and its environment. Rushby's thesis is that it is possible to use formalism, supported by automation, to eliminate logic doubt, i.e. proving that an argument is deductively sound [21]. Expert judgement, however, is still necessary for dealing with epistemic doubts.

There has also been significant work on defining a formal basis for GSN arguments, patterns and modules [22] [23] [24] [25]. This issue is discussed in more detail in by Habli and Kelly in [26]. For example, Denney and Pai propose a formal basis for GSN arguments [22] and patterns [23] and offer automated means, implemented in the AdvoCATE tool [6], for the assembly of safety arguments and the instantiation of argument patterns. In both cases, i.e. assembly and instantiation, the automatic generation of the argument is based on a table that has the data entries needed for populating the argument.

Our approach complements the above but does not depend on having to predefine assembly and instantiation data in tables. Rather, this data is automatically extracted from the design and safety analysis models, based on a weaving model, and is then used to instantiate the argument pattern. This helps in assessing and ensuring traceability between the sources of information, e.g. in design, process and analysis models, and the assurance case. Automation in this way also has the potential to support the coevolution of system design and assurance cases.

V. Discussion and Conclusions

Automation and structured modelling provide two of the main pillars of our approach. Below we reflect on the utility of increased automation and formality for the development and assessment of assurance cases.

1. *Automatic instantiation of argument patterns:* argument patterns are intended to capture the essence of a potentially repeatable reasoning used in an assurance process. This reasoning requires extensive domain knowledge and experience (i.e. an in-depth understanding of what makes systems safe or unsafe in a particular domain or for a particular class of technology). Pattern construction is basically a creative engineering activity and as such will largely remain manual. However, once an argument pattern is defined, reviewed and accepted, instantiation will benefit from automated support. Machines, compared to humans, are more capable in terms of parsing large volumes of information, from diverse sources, and instantiating argument patterns based on extracting relevant information needed for the different argument elements (e.g. claims and assumptions). Our approach utilises automation for the argument pattern instantiation rather than for the pattern creation process. In particular, automation here, supported by increased precision enforced by the metamodels, improves repeatability and consistency in pattern instantiation.
2. *Mapping between argument pattern roles and external models*: many real assurance arguments are, unfortunately, often referred to as "*cartoon arguments*". They are based on common high-level argument patterns whose instantiation lacks sufficient details about the specifics of the system under consideration. That is, the traceability between the argument and evidence in the assurance case and the actual system design and analysis models is questionable. Our approach tries to bridge this gap by explicitly defining pattern roles that need to be traced to specific types of information in these models. This can help improve the validity of the argument against the available design and analysis models (a big challenge for safety cases as highlighted in the Nimrod Accident Review [27]). Further, these models themselves are interdependent. When this interdependency is explicitly defined, such as in our approach, it enables detailed analysis of the reasoning in the argument, e.g. reviewing the strength of a piece of design evidence (based on a design model) and auditing the process by which this evidence was generated (based on a process model that is linked to that design model). In our approach, complex model-to-model relationships are explicitly modelled and form part of the weaving model itself.
3. *Direct instantiation of the argument patterns from models:* unlike most existing means for automated

pattern instantiation (e.g. [23]), where the necessary information is extracted manually from the models, our approach directly links the argument patterns to the sources of information in these models and helps avoid potential humans errors (e.g. interpretation and translation errors). More importantly, the automated and direct instantiation from these models can highlight claims or evidence where the information in the models is incomplete. Our automated approach flags incompleteness in the form of *uninstantiated* or *undeveloped* elements that require a more detailed investigation by the analyst. Information can also be fed back directly to the models themselves, highlighting inconsistencies between aspects of the design and analysis.

4. *Advanced model management for assurance cases*: by being based on, and traced to, well-defined metamodels, an assurance case model can exploit the benefits of model-driven engineering such as model validation, merging, querying and transformation. For example, this includes argument validation against predefined internal constraints (e.g. coverage of a claim by the available evidence) and external constraints (e.g. faithfulness of referenced evidence against actual project data).
5. *Scalability and external validity*: our evaluation so far has focused on demonstrating the feasibility of exploiting a weaving model for automated generation of assurance cases. On-going research will evaluate the external validity and scalability of this approach based on applying it to systems and patterns defined by industry, e.g. the argument patterns currently being developed by the MISRA Working Group on Automotive Safety Cases and ISO-26262 [28].

In conclusion, this paper has shown how our approach can be used to generate an assurance argument for a system using information extracted *directly* from design, analysis and development models of that system. We have described how assurance cases are a synthesis of complex information from a variety of sources. Ensuring that the assurance case is consistently generated from a diverse and large set of source models is a challenging task. Our approach provides a model-based foundation for addressing this challenge.

ACKNOWLEDGEMENT

This work was part funded by the European Union FP7 D-MILS project (www.d-mils.org).

REFERENCES

[1] Health Foundation, “Using safety cases in industry and healthcare”, December 2012.

[2] GSN Community Standard Working Group, “GSN community standard,” 2011. Available at www.goalstructuringnotation.info/

[3] http://www.adelard.com/asce/choosing-asce/index.html

[4] Object Management Group (OMG), “Structured assurance case metamodel (SACM),” version 1.0, 2013. Available at www.omg.org/spec/SACM/

[5] T. Kelly and J. McDermid, “Safety case construction and reuse using patterns”, in proc. Safecomp 97, pp 55-69, Springer, 1997.

[6] E. Denney, G. Pai, and J. Pohl. "Advocate: An assurance case automation toolset". in proc. Workshop on Next Generation of System Assurance Approaches for Safety Critical Systems (SASSUR), pp 8-21, 2012.

[7] S. Voss, B. Schatz, M. Khalil, and C. Carlan, “Towards modular certification using integrated model-based safety cases,” in proc. VeriSure: Verification and Assurance Workshop, 2013.

[8] M. Didonet Del Fabro, J. Bézivin, F. Jouault, B. Erwan, and G. Gueltas, “AMW: A generic model weaver,” in proc. 1ères Journées sur l’Ingénierie Dirigée par les Modèles, 2005.

[9] M. Didonet et. al., “Applying generic model management to data mapping,” in proc. Bases de Données Avancées (BDA05), 2005.

[10] D. Kolovos, L. Rose, A. Garcia-Dominguez, and R. Paige, “The Epsilon book,” available at http://www.eclipse.org/epsilon/doc/book/, October 2013.

[11] J. Rushby, “Separation and integration in MILS (The MILS constitution),” Technical Report SRI-CSL-08-XX, SRI International, 2008.

[12] I. Habli T Kelly, “A model-driven approach to assuring process reliability”, 19th IEEE International Symposium on Software Reliability Engineering (ISSRE), Seattle, USA, November 2008.

[13] SAE, “Architecture analysis & design language (AADL), Annex C AADL Meta Model and Interchange Formats”, SAE International, 2006.

[14] Y. Matsuno, S. Yamamoto, "An implementation of GSN community standard," Assurance Cases for Software-Intensive Systems (ASSURE), 1st International Workshop, 2013.

[15] http://nasa.github.io/CertWare/

[16] OMG, “Machine-checkable assurance case language (MACL),” RFI, 2012. Available at http://www.omg.org/cgi-bin/doc?sysa/2012-9-4

[17] B. Schaetz, M. Khalil and S. Voss, “A pattern-based approach towards modular safety analysis and argumentation”, Embedded Real Time Software and Systems (ERTS2), February 2014.

[18] E. Armengaud, “Automated safety case compilation for product-based argumentation”, Embedded Real Tme Software and Systems (ERTS2), February 2014.

[19] J. Rushby, “Formalism in safety cases,” Making Systems Safer: Proceedings of the Eighteenth Safety-Critical Systems Symposium, Springer, 2010.

[20] J. Rushby, “Logic and epistemology in safety cases”, In Computer Safety, Reliability, and Security: Proceedings of SafeComp ‘13, Springer, 2013.

[21] J. Rushby, “Mechanized support for assurance case argumentation,” in proc. 1st International Workshop on Argument for Agreement and Assurance (AAA 2013), Springer LNCS, 2013.

[22] E. Denney and G. Pai, "A formal basis for safety case patterns", in proc. 32nd International Conference on Computer Safety, Reliability and Security (SafeComp '13), 2013.

[23] E. Denney and G. Pai, "A lightweight methodology for safety case assembly", in proc. 31st International Conference on Computer Safety, Reliability and Security (SafeComp '12), 2012.

[24] Y. Matsuno, “A design and implementation of an assurance case language,” in proc. IEEE/IFIP Dependable Systems and Netoworks (DSN), 2014.

[25] Y. Matsuno and K. Taguchi, "Parameterised argument structure for GSN patterns", in proc. International Conference on Quality Software (QSIC 2011) , pp96--101, IEEE, 2011.

[26] I. Habli and T. Kelly, “Balancing the formal and informal in safety case arguments,” VeriSure: Verification and Assurance Workshop, colocated with Computer-Aided Verification (CAV) 2014, Vienna, Austria, July 2014.

[27] C. H. Cave, “An independent review into the broader issues surrounding the loss of the RAF Nimrod MR2 Aircraft XV230 in Afghanistan in 2006,” The Stationary Office, Tech. Rep., 2006.

[28] I. Habli, et. al., “Safety Cases and Their Role in ISO 26262 Functional Safety Assessment”, 32nd International Conference on Computer Safety, Reliability, and Security, Toulouse, France, 2013.