Developing Assurance Cases for D-MILS Systems

Richard Hawkins, Tim Kelly, Ibrahim Habli
Department of Computer Science
The University of York
York, UK
{richard.hawkins, tim.kelly, ibrahim.habli}@york.ac.uk

ABSTRACT

When using a D-MILS approach for high-assurance systems it is often necessary to develop an assurance case, containing an argument supported by evidence, that demonstrates that the system has the required assurance properties (such as security or safety). In this paper, we describe our approach for developing a D-MILS assurance case, which is based upon a set of modular assurance case patterns that are automatically instantiated using a model-based instantiation process. We illustrate the application of our approach using a small cryptographic controller example and explain the benefits brought by our approach in support of D-MILS.

1. INTRODUCTION

The Distributed MILS (D-MILS) project [1] is extending the MILS approach to distributed computer systems operating across a network. When using a D-MILS approach for high-assurance systems a justification must be provided. This justification can be provided by an assurance case, containing an argument supported by evidence, that demonstrates that the system has the required assurance properties (such as security or safety).

Our approach to developing an assurance case for a D-MILS system seeks to minimise the cost and effort associated with assurance, whilst ensuring that when required, the highest levels of assurance can be demonstrated. In addition we also seek to support the objectives of a DMILS approach, such as compositionality of independently developed components.

Our approach is based around three fundamental elements: assurance argument patterns, modularity, automated instantiation directly from system models.

1.1 D-MILS Assurance Case Patterns

Although each D-MILS system that is created will be unique in terms of its requirements and functionality, there are many features that will be common to all D-MILS systems. In particular all D-MILS systems will share common architectural features, particularly at the level of the D-MILS platform. In addition, D-MILS systems will be assessed using a common verification framework. Because of the large number of shared features of D-MILS systems it is desirable that all D-MILS assurance cases should adopt a common approach to arguing assurance. Assurance case patterns are an established technique for the documentation and reuse of argument structures [2]. Assurance case patterns allow the general structure of the required argument and evidence to be abstracted from the specific details of any particular argument through the use of abstraction and choices. The patterns can then be instantiated for the target system by using relevant information. We have created patterns for D-MILS system assurance case arguments. These are discussed in more detail in Section 2. An important advantage of a pattern-based approach is that it ensures a consistent argument approach is adopted for all D-MILS systems.

1.2 Modular Assurance Case

The verification approach being adopted for D-MILS is a formal approach that uses compositional verification techniques to prove desired properties of the system are met through the integration of independently assured components. It is desirable that the assurance case aligns with this compositional approach as far as possible. Modular assurance cases where originally introduced in [3] as a way of breaking up large assurance cases into separate but interconnected modules of argument and evidence, with each assurance case module reasoning about one aspect of the overall case. As described in Section 2, the patterns we have developed for D-MILS systems are split into modules (concerning for example each software or platform component) with dependencies captured by inter-module references ("away goals") to claims in other modules. Adopting a modular approach also supports the MILS philosophy of fostering a marketplace for MILS components, developed by different organisations, since the assurance case modules can be developed independently by each organisation. To support this we deliberately avoid constraining the assurance methods (or standard) adopted by third-party providers. Our approach identifies specific assurance claims that must be supported in third-party assurance case modules; it is the responsibility of the supplier to provide sufficient assurance in the truth of these identified claims (see Section 2 for more details).

1.3 Automated Instantiation

Instantiating an assurance case pattern involves identifying the necessary information relating to the target system required to choose and instantiate the assurance claims and to provide the required evidence. In this sense the instantiable elements of the patterns define requirements for information. It would be possible to manually obtain this information and instantiate the argument patterns; this is current practice. A manual approach however is not considered to be an ideal solution for DMILS systems. Large parts of the instantiation of the argument are based upon information about the system design as captured in the design and analysis models. The instantiation of such aspects are repetitive and mechanistic in nature and prone to human error. With some less rigid aspects of the assurance argument, the process of argument instantiation itself can be beneficial for the human engineer (who must analyse and develop a solution for an assurance claim). With the mechanistic aspects of argument instantiation, most value of human involvement instead comes from the ability to review the resulting instantiated argument. Our approach retains human review as a key element. Automated instantiation of the assurance argument patterns brings the following benefits over a manual approach:

- The argument is generated directly from, and is therefore consistent with, the design and development models themselves.

- Snap-shot instantiations can be produced quickly and easily to reflect the current state of development.
- Consistent, reusable instantiation rules are established, ensuring consistent instantiations. This is particularly important where complicated relationships between multiple models are required.
- Human instantiation error is mitigated, it also becomes easier to check and verify the resulting output.
- Automated instantiation as a means for design assessment, i.e. highlighting claims and evidence needed for assurance but not available in the system information models.
- Automated support for change management.
- The generated argument is still in a format that is amenable to human review.

Based on the current models of the system, the areas of the argument requiring further development and support are highlighted. This allows the human effort to be focused on analysing and addressing those areas where most value is added.

Ultimately, the aim is to use automation where it is applicable to make it as easy and quick as possible for the system developer to create the assurance case, whilst ensuring that a rigorous and consistent approach is adopted, and that aspects requiring further human consideration and analysis are explicitly highlighted.

In Section 3 we describe in more detail the model-based approach we have developed for automatically instantiating large parts of the assurance argument patterns directly from system models.

2. D-MILS Assurance Case Patterns

The modular structure of the assurance case for D-MILS systems is shown in Figure 1.

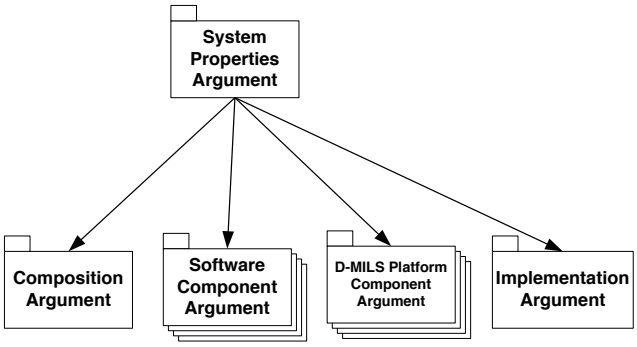


Figure 1. Modular structure of a D-MILS assurance case

Figure 1 shows the assurance case modules that contain the argument and evidence for various aspects of the D-MILS system assurance case. There is a system properties module that describes the top-level structure that supports the overall assurance claim for the D-MILS system; that the required safety and security properties are satisfied. This is supported by a number of other modules of argument. The composition module reasons (through formal analysis) that the system AADL model satisfies these properties. In making these guarantees, the compositional verification places assumptions on both the constituent software components, and also on the D-MILS platform components. These assumptions become local properties of those components, that must be assured. Separate assurance case modules are created for each component, where the argument and evidence relating to the satisfaction of these local properties is presented. The final module shown in Figure 1 is the implementation module. This must demonstrate that the AADL model (on which the required system properties have been proved) is correctly implemented. This involves consideration of the configuration of the platform and the network.

We have created assurance case patterns for the structure of the argument required in each of these modules. In the rest of this section we present and briefly discuss the basic structure of these patterns. The patterns are presented using the Goal Structuring Notation (GSN) [4], which is the most widely used graphical notation for representing assurance arguments. The main elements of GSN are shown in Figure 2. These symbols can be used to construct an argument by showing how claims (goals) are broken down into sub-claims, until eventually they can be supported by evidence (solutions). The strategies adopted, and the rationale (assumptions and justifications) can be captured, along with the context in which the goals are stated.

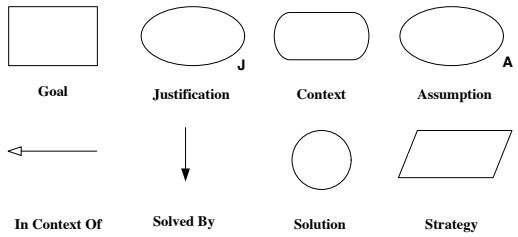


Figure 2. Main elements of GSN

In order to make reference to argument elements in other modules, it is necessary to use the concept of away references. Figure 3 shows an away goal reference from one module to a goal in another module. The use of away references allows inter-module dependencies to be documented.

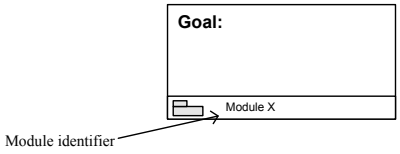


Figure 3. GSN away goal reference

In order to create patterns of argument, the basic GSN notation is extended to allow abstraction, multiplicity, choices and optionality. An argument is created for a system by instantiating the argument pattern using information specific to the target system. Abstract elements include explicitly defined "roles". 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 Figure 4, 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.

Figure 5 shows how (a) multiplicity, (b) choice and (c) optionality can be indicated on the relationships in an argument. For multiplicity relations, the number of required argument elements (n) must be determined, again using information from the target system (e.g. an entity created for each of the functions present in the system design). Argument elements may be denoted as optional in the pattern, or choices provided for different argument approaches that may be adopted (in Figure 5, one source node has

three possible alternative sink nodes). At instantiation, the assurance elements most appropriate for the target system must be chosen from the options provided in the pattern.

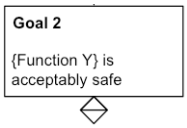
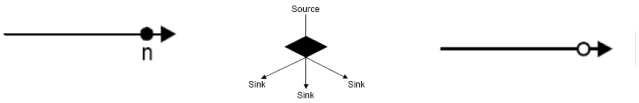


Figure 4. GSN away goal reference



(a) multiplicity (b) choice (c) optional

Figure 5. GSN away goal reference

2.1 System Properties Module

Figure 6 shows the assurance case pattern for this module. The top claim is that the defined assurance properties of the D-MILS system are satisfied. We consider functional, relal-time, security and safety properties, but any properties of interest could be considered. There can be seen to be an additional claim, "Goal: secPolicyComplete" that there is sufficient confidence that these defined properties are complete and correct with respect to the hazards and vulnerabilities of the D-MILS system. Although this claim is very important to the rigour of the overall case, it was outside of the scope of the D-MILS project, so we leave it here as an away reference that must be discharged elsewhere (as part of the system hazard and security analysis process).

The left hand side of the pattern claims that the properties are satisfied by the MILS-AADL model. A separate claim is made for each of the defined properties (the verification is conducted on a per-property basis) and these claims are discharged in the composition module. The right hand side of the pattern claims that the properties remain satisfied once the MILS-AADL model is implemented on the platform (the implementation is performed correctly). This claim is discharged in the implementation module.

In the centre of the pattern, claims are made that the required assertions are guaranteed by each of the trusted software components and by each component of the D-MILS platform. These assertions arise from the assumptions that are necessary for the compositional verification, and are defined in the composition module. The claims that the assertions are guaranteed are to be discharged by the relevant component module. As mentioned earlier, these components will often be provided by third-party providers, and as such it is desirable to assumed that any particular standard or approach has been adopted. We therefore leave the discharging of the required assertion to the third-party,

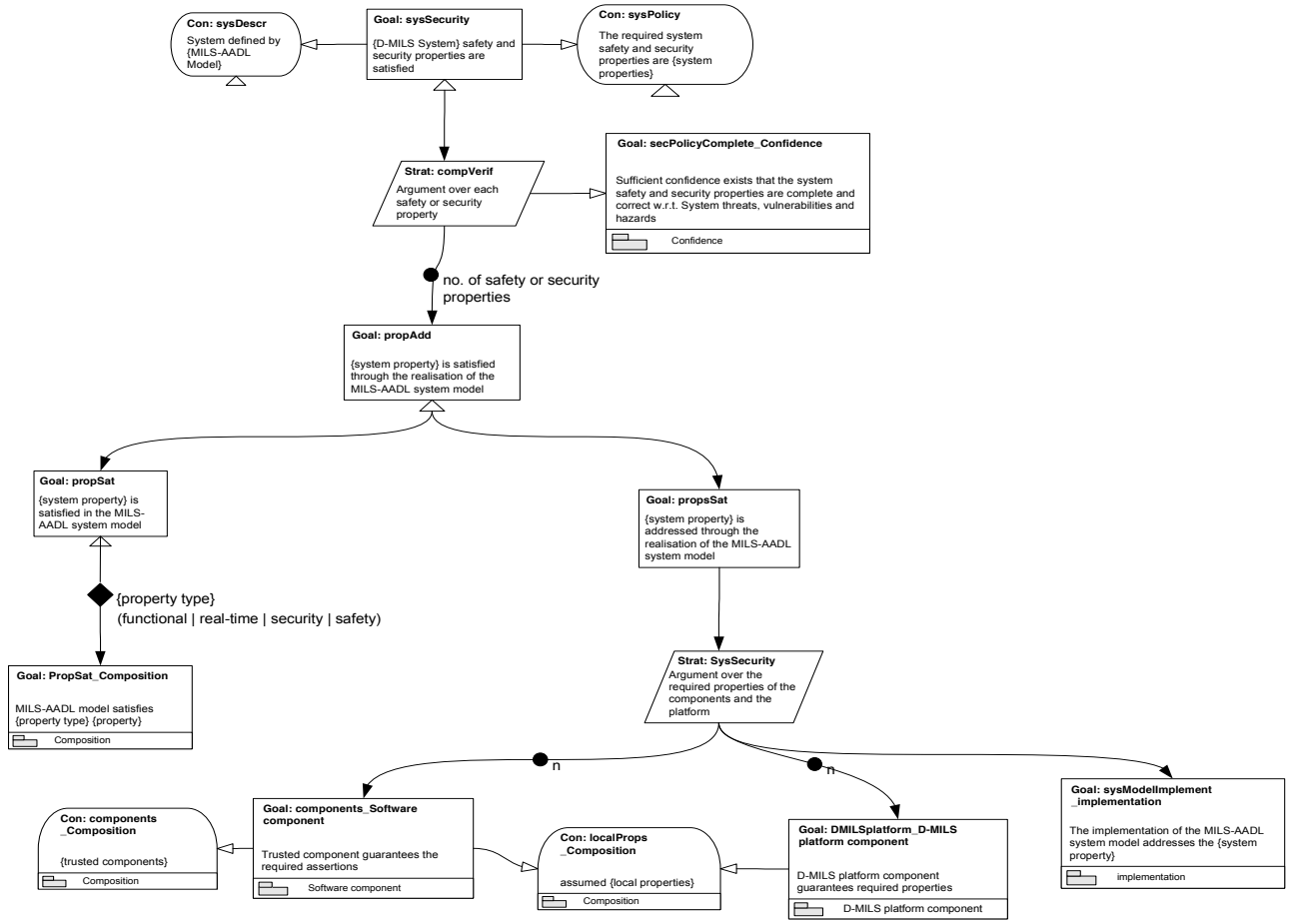


Figure 6. System properties assurance case argument pattern

and instead focus on ensuring the assertion is correctly and explicitly defined. This then in effect becomes an assurance requirement upon the provider of the component.

In this argument pattern we expect to automatically instantiate all of the elements except for the completeness and correctness of the defined properties (for reasons discussed earlier). This includes automatically defining the requirements for the third-party component providers.

2.2 Composition Module

Figure 7 shows the assurance case pattern for this module. The top claim of this module is instantiated for each property to be verified. If we focus firstly on the left hand side of the pattern, a claim is made that formal verification proves that the AADL model satisfies that property. It is then required to provide the results of the verification that proves this; the verification results themselves are used as evidence for this. In addition it must be shown that there is sufficient confidence in those formal verification results. As can be seen in Figure 7, this requires three claims to be supported: that the formal verification is undertaken correctly, that the assumed local properties of the components are computed correctly, and that the formal model used in the analysis is a correct representation of the MILS-AADL model. Only if these claims can be supported with sufficient assurance can the formal analysis results themselves be trusted. It should be noted that, since this aspect of the argument will generally be expected to be the same for all properties of the same type, since the approach used for verifying those properties is likely to be the same.

The right hand side of this pattern deals with the case where the property to be verified is a "safety" property. On D-MILS, the term safety property is used to consider properties required where failures occur in the system. In these cases the MILS-AADL error model must also be considered as part of the verification analysis, so the completeness and correctness of this model must also be demonstrated.

We also expect large parts of this pattern to be automatically instantiated. The elements where manual instantiation will be required are in the goals below "Goal: formalConf". Where reasoning is required about the rigour of the verification process itself, and also about the rigour of the verification tools applied, it is desirable that further reasoning based upon the knowledge of the engineers themselves is utilised.

2.3 Implementation Module

Figure 8 shows the assurance case pattern for this module. The properties are formally verified, but only on the MILS-AADL model. The purpose of this argument module is to demonstrate that that MILS-AADL model is correctly implemented. The implementation process for D-MILS involves using a configuration compiler to generate configuration files that implement the model on the D-MILS platform. This process

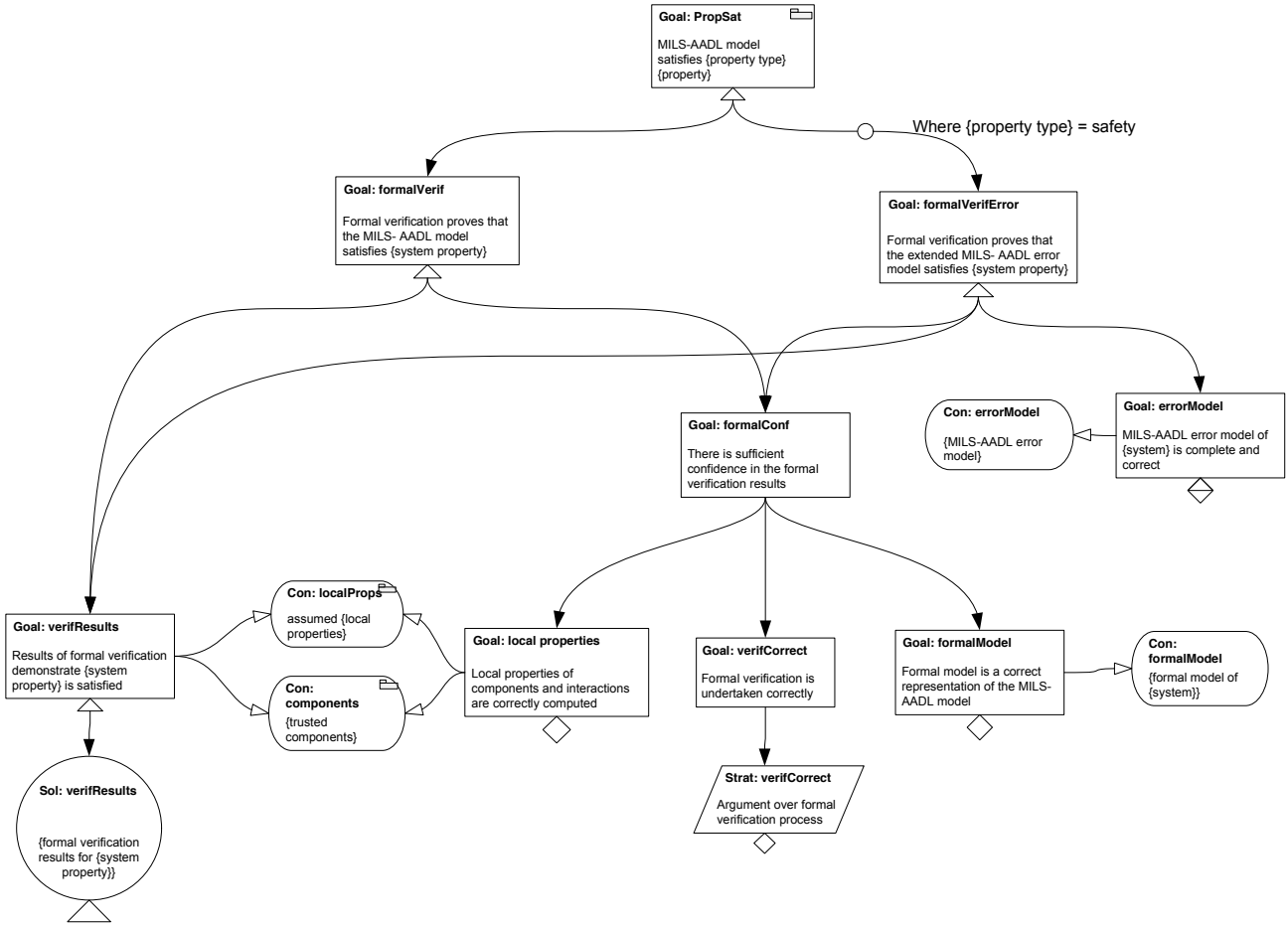


Figure 7. Composition assurance case argument pattern

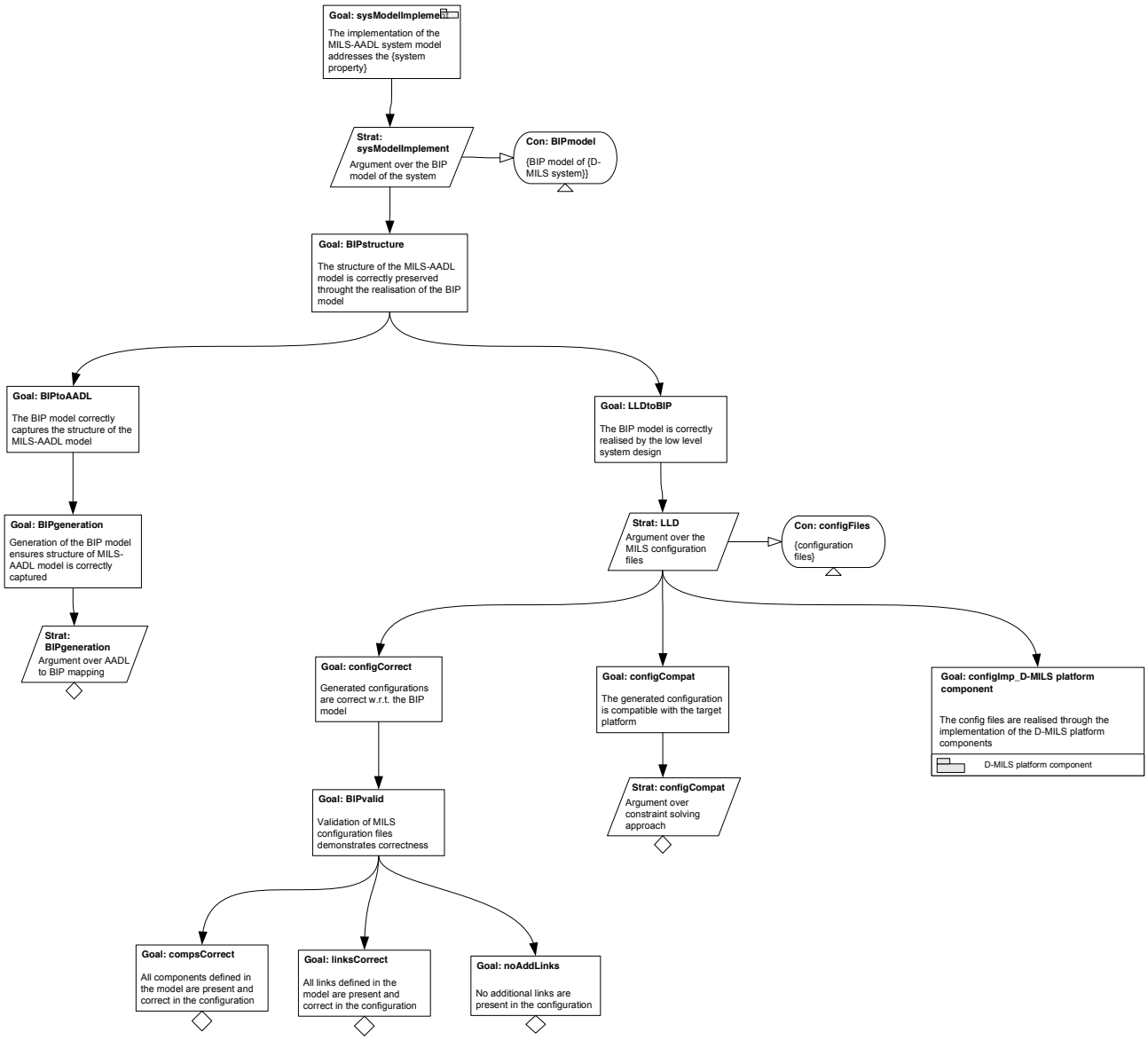


Figure 8. Implementation assurance case argument pattern

involves translating the MILS-AADL model into an intermediate representation, using the BIP (Behavior, Interaction, Priority) language. The BIP model can then be used to generate the configuration files through a process of constraint solving.

Essentially, this module presents an argument about model transformations, and seeks to demonstrate that these transformations are undertaken correctly. The argument firstly considers the BIP model. It must firstly be demonstrated that the structure of the MILS-AADL model is correctly captured. It must then be demonstrated that the BIP model is correctly realised by the generated configuration files. This requires assurance that the configurations are correct with respect to that BIP model, but also importantly that the configurations are compatible with the target platform. Finally, the configuration files must be implemented on the D-MILS platform. This claim is an away reference to the platform component modules.

The nature of these arguments, being largely about confidence in a translation process, are less amenable to automated instantiation than the other modules discussed. However some of the claims, particularly those relating to the validity of the configuration files, may be automatable using information extracted from the analysis.

3. Argument Instantiation Process

We have developed an approach that enables the instantiation of the D-MILS assurance case patterns directly from the relevant models of the system that are created (this will include design and analysis models). This model-based approach is illustrated in Figure 9, and described in detail in [5]. Here we illustrate how the approach has been implemented using a simple example D-MILS system. Firstly we briefly discuss some of the advantages of the approach we have adopted.

Firstly, the approach is tool and notation independent. So long as the tools used for the input models provide models that conform to their own defined metamodels they are compatible with this

approach. The notation in which the models are constructed is also unimportant so long as an XML representation of the model can be provided.

Secondly, a weaving model is at the heart of our approach. It is the weaving model that links the reference information metamodels to the patterns. The weaving model captures the dependencies between the role 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 metamodel that enables the argument instantiations to be performed. 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. Our utilisation of a weaving model makes this dependency information both explicit and precisely defined. The weaving model is also a useful mechanism for capturing the more complex dependencies between models that are often required for an assurance argument.

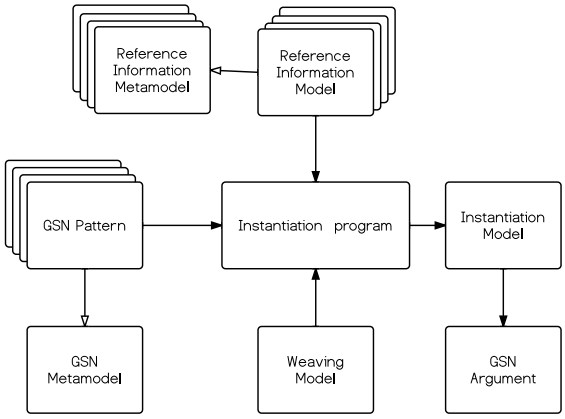


Figure 9. Overview of the Model-Based Assurance Case Approach

Thirdly, using a model-based approach allows us to take advantage of the existing extensive set of general model-based engineering 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 [6] family of languages and tools for model management.

Finally, as a result of the explicit weaving metamodel discussed above, it becomes possible to partially automate analysis and validation of the assurance case. Most automated analysis of assurance cases focusses on the verification of the argument, i.e. identifying logical doubts in the argument itself. Our approach helps with the assessment of epistemic doubts in the argument (i.e. with respect to the external information models for the system). It is of course possible to identify epistemic and validity challenges through a traditional approach, however this relies solely on the judgement and experience of the engineer to, for example, identify all relevant contextual links correctly. Our approach helps in providing a consistent and systematic identification of such issues.

3.1 Example Automated Instantiation

Here we illustrate our approach using the example of a simple D-MILS system. The example we present is a software cryptographic controller system taken from [7]. The architecture for the system is shown in Figure 10.

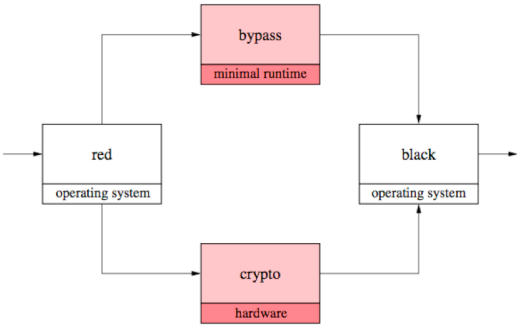


Figure 10. Software architecture of crypto controller system

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 property to be assured is that no unencrypted message content

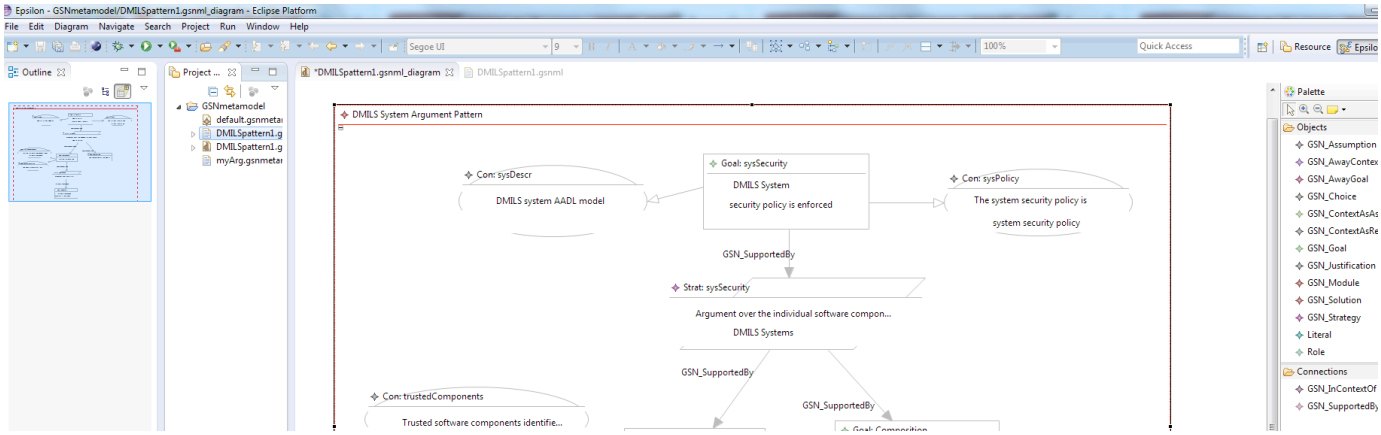


Figure 11. Graphical editor for creating GSN argument pattern models

information shall be passed to the black network. The local properties required of each of the components for compositional verification are:

- crypto shall encrypt everything that leaves on its outgoing channel
- 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 prove the overall security property). Using our approach, assurance case modules would therefore only be required for the crypto and bypass components.

```
<?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 12. Assurance Argument Pattern Model in GSNML (Partial)

We now briefly describe how our approach would be used to create an assurance case for this system. Firstly we must create models of the assurance argument patterns that are conformant to the GSN metamodel [5]. To enable graphical-based model generation we have developed from the GSN metamodel a graphical editor using GMF (Graphical Modelling Framework). Figure 11 shows a screen shot of our graphical editor, and Figure 12 shows an extract from one of the resulting models in XML form, (which we call GSNML files). This GSNML file is taken as input by the instantiation program.

If we consider just the system properties pattern presented in figure 6, the instantiation information for this can all be obtained from the AADL specification that has been created for the system, as seen in Figure 13. In the general case, multiple models will be required to instantiate a pattern. Each model is taken as input.

A weaving model is created 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 [8]). Figure 14 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. The model weaving can be performed either manually, i.e. by linking the related elements by hand as we do here, or automatically, i.e. through a model transformation. We aim to exploit automated model weaving approaches, such as those described in [9] to facilitate this step.

```
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 into header
--
node Splitter
  features
    frame: in data port Frame;
    header: out data port Header;
    payload: out data port Payload;
end Splitter;
node implementation Splitter.Imp
  flows
    port fst(frame) -> header;
    port snd(frame) -> payload;
end Splitter.Imp;
```

Figure 13. Extract of the AADL specification for the crypto controller system

The models described above (pattern models, AADL model and weaving model) were used as input to our instantiation program, which is an Epsilon Object Language (EOL) [6] program that runs on the Eclipse platform. 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.

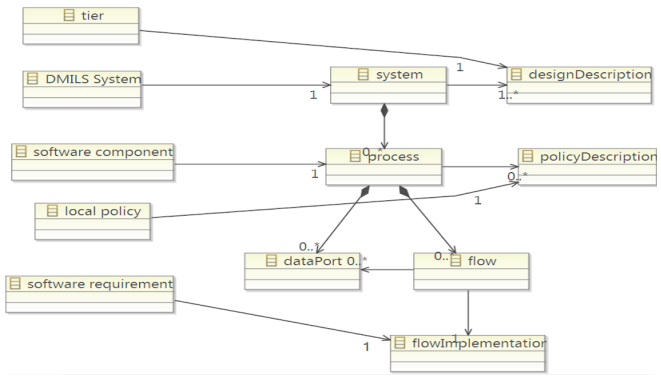


Figure 14. Representation of the example weaving model

4. Conclusions and Future Work

In this paper we have described our approach for generating assurance cases for D-MILS systems. The approach is based upon a set of modular assurance case patterns and a model-based instantiation process. We have described how our approach brings

particular benefits in support of D-MILS. This includes ensuring a consistent approach is adopted for all D-MILS systems through the use of patterns and automated instantiation. We also support compositionality and the use of independently developed components through the use of modularity.

The work described in this paper is still under development and will be developed further. In particular the approach will be applied to full-scale industrial case studies to evaluate the effectiveness and ease of application. The prototype tools that we have developed require further development and validation, particularly the tool support for creating the weaving models (there are existing tools in this area that we will look to harness). The D-MILS patterns themselves are also being developed further. We would also like to explore the opportunities our approach provides for automated verification and validation of the assurance case.

5. Acknowledgements

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

6. References

[1] http://www.d-mils.org/

[2] Hawkins, R., Clegg, K., Alexander, R., and Kelly, T., 2011. Using a Software Safety Argument Pattern Catalogue: Two Case Studies. In *Proceedings of the 30th International Conference on Computer Safety, Reliability and Security (SAFECOMP '11),* (Naples, Italy, 2011).

[3] Kelly, T., 2001. *Concepts and Principles of Compositional Safety Case Construction*. Technical Report COMSA/2001/1/1, The University of York.

[4] GSN Committee, 2011. *GSN Community Standard Version 1*. http://www.goalstructuringnotation.info/.

[5] Hawkins, R., Habli, I., Kolovos, D., Paige, R., Kelly, T. 2015. Weaving an Assurance Case from Design: A Model-Based Approach. To appear in *Proceedings of the16th IEEE International Symposium on High Assurance Systems Engineering (HASE)* (Daytona Beach, Florida, USA, January 2015).

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

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

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

[9] Didonet Del Fabro, M., Bézivin, J., Jouault, F., Erwan, B., and Gueltas, G., AMW: A generic model weaver. In *proc. 1ères Journées sur l'Ingénierie Dirigée par les Modèles*, 2005.