Support for Safety Case Generation via Model Transformation

Chung-Ling Lin, Wuwei Shen
Department of Computer Science
Western Michigan University
Kalamazoo, MI, USA
{chung-ling.lin, wuwei.shen}@wmich.edu

Richard Hawkins
Department of Computer Science
The University of York
York, UK
richard.hawkins@york.ac.uk

ABSTRACT

Assessing the safety of complex safety- or mission-critical systems under ever tightening time constraints with any degree of confidence is a growing challenge for industry and regulators alike. One method of helping to address this situation is through the use of assurance cases. Challenges abound here as well; too little or too much abstraction or poorly constructed arguments can affect confidence that a system will perform as intended. The automatic generation of a (safety) assurance case not only can expedite a development process but also leverage the ability to perform compliance checking. In this paper, we propose a novel framework which weaves a safety case pattern, guidance metamodel, and a development process metamodel together to generate a safety assurance case, which facilitates checking the conformance of the system to the guidance. As a case study, we use the GPCA infusion pump project as a subject to illustrate how this framework can aid in compliance checking using the infusion pump guidance published by FDA as a reference oracle.

Keywords

Compliance checking; model transformation; safety-critical systems; safety case.

1. INTRODUCTION

Assessing the safety of complex safety- and mission-critical systems, such as medical devices, under ever tightening time constraints with an acceptable level of confidence is a growing challenge for industry and regulators alike. One method of helping to address this is through the use of safety assurance cases (or safety case in short) [1]. For instance, the U.S. Food and Drug Administration (FDA) recently released an infusion pump a guidance document on the total product lifecycle for infusion pumps [2], which recommends infusion pump manufacturers to use safety assurance case ("safety case") as a structured means to organize and present to FDA the information supporting the safety claims of their infusion pump devices. In this paper, we take the infusion pump guidance as an example to discuss how to automatically construct a safety case in safety critical domains.

The construction and review of a safety case for an infusion pump system are a daunting task for various stakeholders such as manufacturers and FDA regulators due to the following reasons. Firstly, the guidance provides general requirements on what types of safety properties that a safety case should argue about and what kind of evidence it should collect from development artifacts. But, it leaves it up to device manufacturers to decide the ways of constructing a safety case in terms of using the collected evidence to support the specific safety claims articulated for their devices. This however creates a gap between the guidance's requirements and the device development process for the device that needs the manufacturers to properly bridge when constructing their safety cases. This gap also makes it challenging for regulators to review the safety cases, because they need to first understand how guidance requirements are mapped to the safety claims in the safety cases and then evaluate the trustworthiness and qualification of the collected evidence in supporting these claims. Exacerbating the problem is the poor quality of evidence and arguments assembled in the safety cases: many safety cases suffer from the structural problems, such as too little or too much abstraction and poorly constructed arguments.

Secondly, like many other guidance documents or standards across the safety critical industries, the guidance intends to be generic to ensure its applicability to as many infusion pump devices as possible. Consequently, it creates a space for different stakeholders, such as suppliers, clients, and certifiers, to come up with different understanding/interpretation of the guidance's requirements. For example, the guidance recommends manufacturers to conduct hazard analysis to identify the risks associated with their devices and use the results to define the safety claims to be included in the safety cases. However, it leaves it up to manufacturers to decide the specific hazard analysis techniques to use and the process of using such techniques. The difference among stakeholders in interpreting the guidance creates a communication gap between them. Safety cases need to be constructed properly to help to remediate the difference, rather than making it worse.

To address the above challenges, we propose a novel model-based framework, called SPIRIT, that applies the notions of safety case patterns and model weaving to support the mechanical generation and validation of safety cases. Central to SPIRIT is to utilize safety case patterns [3] to enable the mechanized and consistent generation of safety cases for the same type of systems. In this way, the cost of constructing safety cases can be reduced and the confidence of such safety cases can be improved, by reusing the safety case patterns that have been proven as successful in past practices to promote the communication among stakeholders. Beside the safety case pattern, SPIRIT requires two additional inputs: a guidance metamodel, in the format of a UML class diagram, to denote the guidance and remediate the stakeholders' difference in interpreting the guidance; and a development process metamodel that defines how a manufacturer designs their infusion

pump. Thus, SPIRIT can weave a safety case pattern, the guidance metamodel, and a development process metamodel together to generate a safety case that enhance the argument of an infusion pump's conformance to the guidance.

In [4], the authors described a model-based approach for developing safety cases based upon the concept of model weaving. In general, a safety case pattern allows some variables, called roles, to be defined in its nodes. These roles can be mapped to different classes or concepts depending on the application, a process known as instantiation. In this paper, we extend the above idea via a two-step mapping given in a weaving model to support compliance checking. Firstly, recall the guidance recommends what kind of information can be used to construct a safety case. To this end, a weaving model first maps roles in a safety case pattern to concepts of the guidance defined in the guidance metamodel. However, to comply with the guidance, a manufacturer should demonstrate how the safety case is linked to the various artifacts produced by their development process. Secondly, the weaving model gives the mapping relations between concepts of the guidance and classes of development process metamodel, illustrating how the development process complies with the guidance.

Once a safety case pattern, the guidance metamodel, and a development process metamodel are fed to SPIRIT, it generates an Atlas Transformation Language (ATL) program [5], which can be executed to generate a safety case for any infusion pump device developed according to the development process metamodel. In this paper, we use an example infusion pump system, the Generic Patient-Control Analgesia (GPCA) infusion pump, to complement the entire application process of SPIRIT. The results demonstrate that SPIRIT can support both the validation of a safety case pattern (against the input infusion pump development process), and the generation of safety cases for infusion pump systems.

In summary, we make the following contributions in the paper.

1) The automatic support of safety case pattern validation and safety case generation, to increase the reusability of successful safety case patterns and to improve the capability of building and reviewing safety cases;

2) A weaving model mechanism that connects various elements from a safety case pattern, the infusion pump guidance, and a development process for an infusion pump together;

3) Integration of safety case generation into a development process.

2. PROBLEM STATEMENTS AND SPIRIT'S SOLUTIONS

Although the guidance documents released by FDA are not legally binding, the guidance contains agency's recommendation of good practices to assure and improve some aspects of safety and effectiveness of medical devices. A question thus facing the medical device industry and FDA regulators is that, *given a manufacturer's own practice, how can they find a solution to develop and certify a medical device which follows the related FDA guidance document(s) in an effective and efficient fashion?*

As our first step of answering the above question, we present a model-based framework, SPIRIT, to mechanize the generation of safety cases that argue safety-critical systems satisfy the safety requirements posed by the guidance document(s). In its current

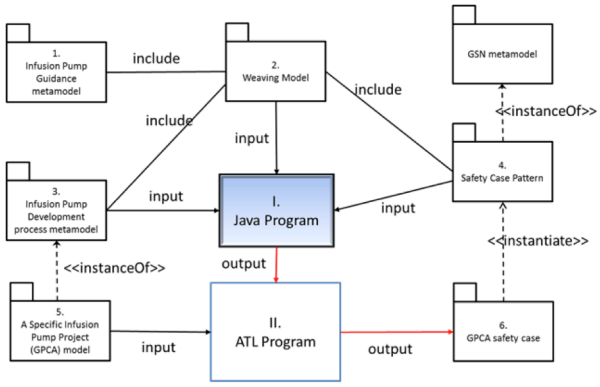

Figure 1 The overall structure of the SPIRIT framework

implementation as illustrated in Figure 1, SPIRIT considers the infusion pump guidance issued by FDA and helps to develop and validate a safety case that argues the software in an infusion pump developed under a manufacturer's own practice satisfies the infusion pump guidance. It should be noted that, although currently focusing on the infusion pump area, the idea underlying SPIRIT can be customized and applied to other safety-critical industries where the problem of certifying a software system against regulatory/legal/standards requirements is of importance.

Underlying SPIRIT is a model-driven approach where a number of metamodels are employed. Firstly, it takes as input a conceptual metamodel that intends to capture the infusion pump guidance. Next, it prompts infusion pump developers to provide a metamodel that defines their software development processes, which should specify the main artifacts produced as a sequence of development activities throughout the process. Lastly, it accepts as input a safety case pattern (Figure 1(4)) and uses it as the basis to produce a safety case for any infusion pump (software) project that is developed following the development process metamodel. Apparently, to ensure the quality and confidence of the final safety case, the input safety case pattern should have proven its accuracy and effectiveness in capturing the requirements of certifying a system against the guidance document.

Figure 2 shows an example of a (partial) safety case pattern in the Goal Structuring Notation (GSN) [6] that can be fed to SPIRIT. The pattern consists of three main types of nodes: claims, also known as goals, which assert certain properties of a system and are denoted as rectangles; strategies (denoted as parallelograms) that are the reasoning steps to argue why a goal is supported by sub-goals or evidence; and evidence (denoted as circle) that are artifacts to support the truth of a goal. In Figure 2, the hierarchy of goals in a safety argument is represented as a tree structure, the root of which is an overall safety claim for the system. The pattern can contain additional context, assumption and justification nodes to help to establish sound arguments.

Notably, the safety case pattern in Figure 2 includes a set of expressions written in the Object Constraint Language (OCL) [7]. Each of such expressions contains variables, known as roles, that are enclosed within { and }. For example, node *G1* contains role *system*, which represents the target infusion pump device. All roles involved in a safety case pattern must be instantiated before a safety case can be produced for the target infusion pump system.

The core of SPIRIT is a weaving model (Figure 1(2)), following the notion proposed in [8], which instructs the instantiation of roles in the safety case pattern to the corresponding elements in

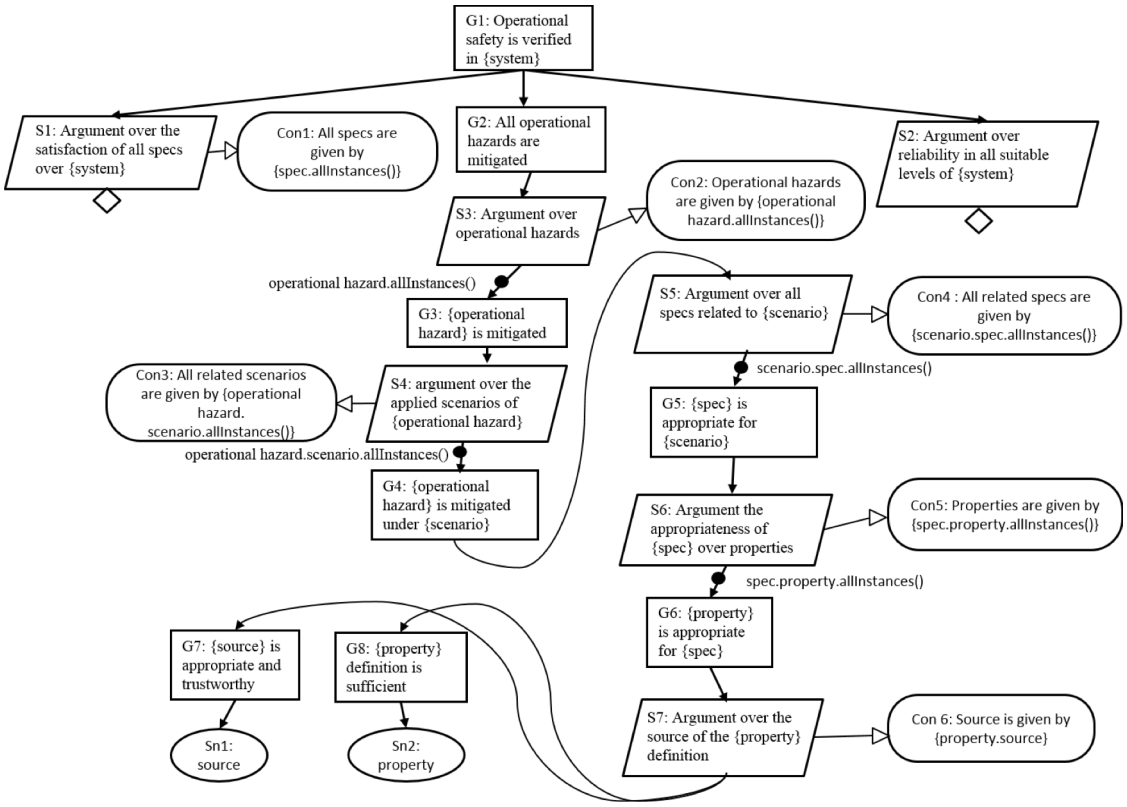

Figure 2 An example safety case pattern

other input metamodels. If the safety case pattern can be successfully instantiated (Figure 1(I)), it indicates that the development process conforms to the guidance. In this case, SPIRIT outputs an ATL program which transforms a specific project developed based on the development process metamodel (Figure 1(5)) to a final safety case. In contrast, if the safety case pattern cannot be instantiated, it suggests that the development process for the infusion pump project is not conformant with the guidance[1], since the safety case pattern reflects the certification requirements for infusion pumps.

More specifically, a weaving model acceptable to SPIRIT should define three types of mapping relationship: from a safety case pattern to the guidance metamodel, from the guidance metamodel to the development process metamodel, and lastly from roles in a safety case pattern to elements in the development process metamodel. Note that the last type of mapping relationship is for situations in which roles (e.g. *source* in Figure 2) cannot be mapped to any element in the guidance metamodel. In this case, the weaving model should appropriately map these roles to elements in the development process metamodel; otherwise, the safety case pattern cannot be instantiated. Finally, all roles are mapped to elements in the development process metamodel; and SPIRIT is thus able to further check whether the safety case pattern can be appropriately instantiated via the

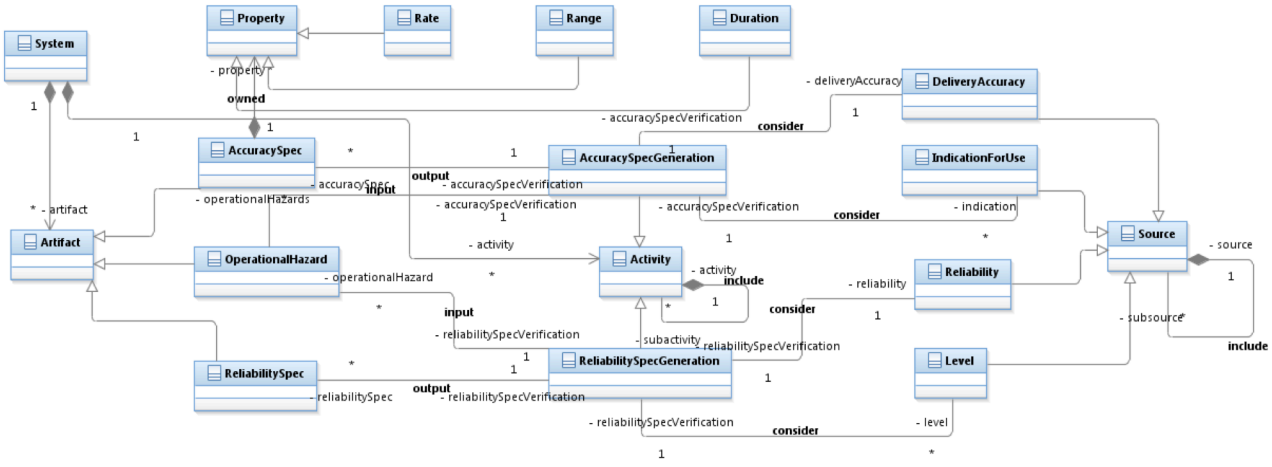

Figure 3 A partial conceptual model of the infusion pump guidance

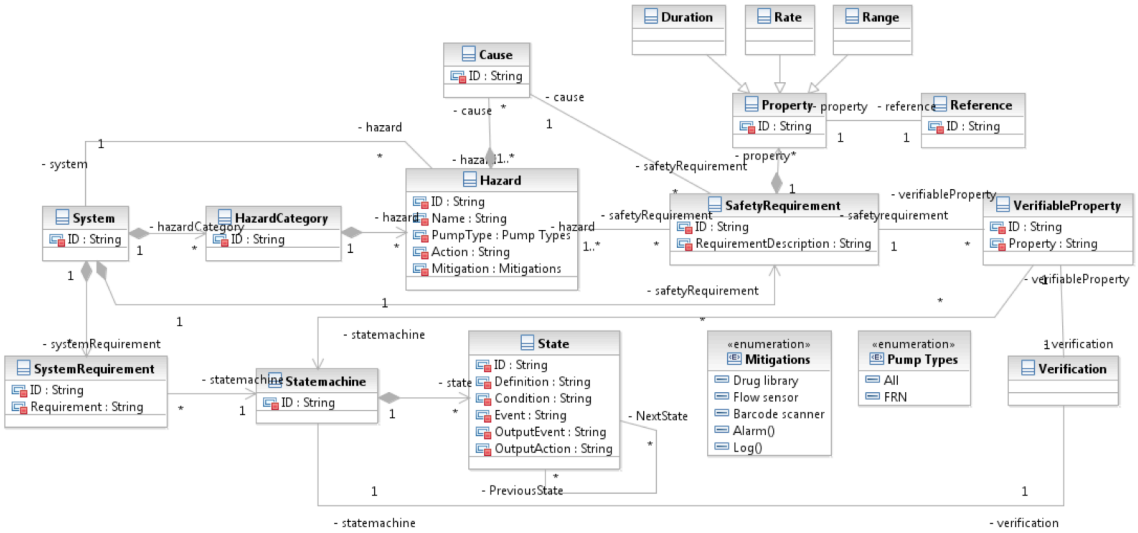

Figure 4 A development process metamodel

elements in the development process metamodel.

Assume that the safety case pattern in Figure 2 is applied to an infusion pump project developed based on a development process metamodel given in Figure 4, and the input guidance metamodel is defined in Figure 3. Table 1 presents a possible weaving model that correlates elements in these metamodels together to instantiate the safety case pattern. For instance, according to Table 1, role *spec* in the safety case pattern is mapped to metaclass *AccuracySpec* in the guidance metamodel and further mapped to metaclass *SafetyRequirement* in the development process metamodel.

3. MODELS IN SPIRIT

Models play an important role in SPIRIT. Central to SPIRIT is a conceptual metamodel which captures the main concepts and their relations specified in the infusion pump guidance. Having an explicit guidance metamodel helps to remediate the different interpretations that different stokeholds hold for the guidance, and hence improve the communication among them. In SPIRIT, we use a UML class diagram to denote the guidance metamodel, since UML has become the de-facto standard for building object-oriented software in the industry. Likewise, we employ a UML class diagram to represent a development process metamodel for an infusion pump project.

To establish a metamodel for the guidance, we adopted the approach presented in [9] that involved carefully going through the guidance and extracting the nouns, some of which are finally allocated to a concept/class name. Synonyms in the guidance were combined by a common concept/class name after consultation with domain experts. We also identified the relationships among various concepts, and formalized these relationships as associations connecting these concepts. Figure 3 shows the conceptual metamodel constructed from the guidance.

A safety case pattern, as seen earlier in Figure 2, provides the necessary structure of a safety argument without the details specific to the target system. Roles in the pattern are then instantiated to the concrete details of the target systems. We adopt GSN as the notation language for the safety case pattern, while a metamodel defining the GSN notation can be found in [8]. The GSN notation has shortcomings to denote safety case patterns. For example, it does not have syntax support to enforce the relationship that may exist between two different roles in the safety case pattern, such as the mitigation relation between roles *safety requirements* and *causes* (to hazards) in Figure 2.

Table 1 A weaving model

Role	Infusion pump guidance conceptual model elements	GPCA development process metamodel elements
{system}	System	System
{spec}	AccuracySpec	SafetyRequirement
{scenario}	IndicationForUse	Cause
{operational hazard}	OperationalHazard	Hazard
{property}	Property	Property
{source}		Reference

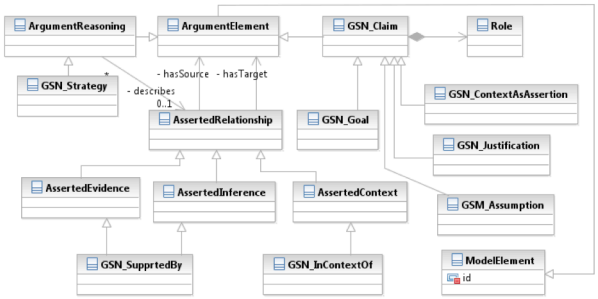

Figure 5 GSN metamodel used by the Java transformation

```
createMatchedRule(node){
1  if( isInstantiable(node)  == true) { // check if all roles are instantiable in this node
2       output the beginning of the ATL module using GSN as the OUT model and the input
            development process model as the IN model;
3       output the beginning of a matched rule;
4       output the ATL from section using the mapping relationship applied to node.getRole();
5       output the beginning of the ATL to section using the GSNmetamodel followed by
          Goal Class Name;
6       output the beginning of the ATL do section
7       output a binding within the ATL do section where the feature name is given by ID
          and the expression is given by node.getID();
8       output a binding within the ATL do section where the feature name is given by Content and
          expression is given by node.getContent() and node.getRole();
9       for each assertedRelationship that has node as the source argument element{
10         assign assertedRelationship.hasTarget() to target;
11         if (target instanceOf GSN_Goal/GSN_Justification/GSN_Assumption/GSN_Strategy)
12              output a binding with the ATL do section where the feature name is given by
                  subgoal/justification/assumption/strategy  and the expression is given by calling
                  the called rule for target.
13      }
14      output the ending of the ATL do section and the matched rule
15      output the ending of the matched rule;
16 }
   else
17      output "node is not instantiable"
18 }
```

Figure 6 Pseudo code for ATL matched rule generation

To overcome these shortcomings, we extend the GSN syntax to express the associations between roles. For instance, to denote all *specs* related to a *scenario* in the safety case pattern in Figure 2, we allow the use of OCL expressions such as *scenario.spec* and operations such as *allInstances()*. In this way, the OCL expression *scenario.spec.allInstances()* denotes all instances of *spec* related to a *scenario*. Such extension to the GSN syntax makes it less challenging to instantiate a safety case pattern to a (specific) target system.

4. IMPLEMENTATION OF SPIRIT

The SPIRIT framework provides two important features, validation and generation/transformation. The validation feature checks whether or not the safety case pattern can be instantiated for the target system. In other words, it checks if all roles/role expressions in the pattern can be mapped to elements in the development process metamodel.

Given a set of variables each of which should be finally instantiated to some class in the development process metamodel, then a role in the safety case pattern can be formally defined as follows:

- Any variable can be a role *r*, called a variable role; and
- If *r* is a role and *x* is a variable, then *r.x* is a role, called a derived role.

A safety case pattern can include a set of role expressions, which are defined as follows:

- Any role *r* is a role expression; and
- If *r* is a role, then *r.allInstances()* is a role expression.

During the validation phase, SPIRIT uses the weaving model to ensure that each role and role expression in a safety case pattern is valid. A role is valid if the following constraints can be satisfied:

- C1: For a variable role *r*, it should be mapped to a class in the development process metamodel according to the weaving model.
- C2: For a derived role *r.x*, if *r* and *x* are mapped to classes *C_1* and *C_2*, in the development process metamodel, then *C_1* and *C_2* should have (at least) one association between them.

Table 2 Binding statements for different types of node contents

Role/Literal	Mapped Class	ATL Statement
a	*a* → A	*var.Content* ← *var.Content* + *A.ID*
a.allInstances()	*a* → A	for (*e* in *Source!A.allInstances())* *IDs* ← *IDs* + *e.ID* + ' '; *var.Content* ← *var.Content* + *IDs;*
a.b	*a* → A *b* → B	*var.Content* ← *var.Content* + *A.b.ID*
a.b.allInstances()	*a* → A *b* → B	for (*e* in *A.b)* *IDs* ← *IDs* + *e.ID* + ' '; *var.Content* ← *var.Content* + *IDs;*
Literal *L*		*var.Content* ← *var.Content* + *L;*

A role expression is valid if the following constraints are satisfied:

- C3: If a role expression has the form *e.allInstances()* where *e* is a derived role *r.x* and the association between the classes mapped from *r* and *x* is *a*, then the multiplicity at the end of the class mapped from *x* for the association *a* must be more than 1.
- C4: If a role expression is a derived role *r.x* that does not end with *allInstances()* and the association between the classes mapped from *r* and *x* is *a,* then the multiplicity at the end of the class mapped from *x* for the association *a* must be 1.

Intuitively speaking, constraint C1 requires that each variable role be mapped to a class in the development process metamodel. For a role expression, SPIRIT makes sure that all classes mapped from the role expression have: 1) the appropriate associations (constraint C2); and 2) the correct multiplicity value at the appropriate end of each association (constraints C3 and C4). For example, for role expression *spec.scenario*, Constraint C2 ensures that classes *SafetyRequirement* and *Cause*, mapped from *spec* and *scenario* according to Table 1, do have an association between

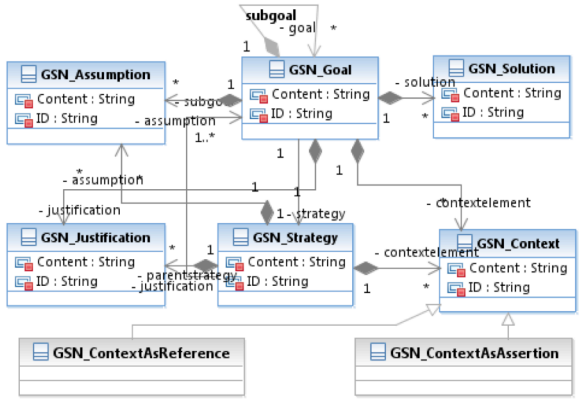

Figure 7 The output GSN metamodel

them in the development process metamodel. If there is more than one association, SPIRIT prompts the user to identify the association to be applied to the role expression. Once *spec.scenario* is decided to satisfy C2, SPIRIT checks C4 to see if the multiplicity at the end of class *Cause* for the association is one. If yes, *spec.scenario* can be instantiated. Similarly, for role expression *spec.scenario.allInstances()*, besides C2, SPIRIT also ensures the multiplicity at the end of class *Cause* for the association must be many.

Once a safety case pattern is confirmed to be instantiable, the transformation feature of SPIRIT takes over, which consists of two steps: 1) Java transformation, which utilizes a built-in Java program to produce an ATL program; and 2) ATL transformation, which executes the ATL program generated in the last step to produce a safety case for the target infusion pump project.

The built-in Java program executes Java Transformation as follows: it first identifies all nodes in the safety case pattern; then for a root goal node in the safety case pattern, it creates a matched rule, and for every other node in the safety case pattern it creates a called rule; it then replaces each role with project-specific elements based on the weaving model; and lastly it builds the structure of safety case via binding statements within ATL rules. In order to allow the built-in Java program to handle the input safety case pattern automatically, the input pattern is edited by the graphical GSN editor tool and formalized based on the GSN metamodel proposed in [8]. Figure 5 illustrates part of the GSN metamodel.

More specifically, a matched rule explicitly specifies, in its *from* section, what type of elements in the source metamodel to be matched. Thus, executing this rule generates the corresponding elements in the target metamodel, if a matched element is found in the source metamodel. A called rule, on the other hand, is essentially a sequence of imperative statements to explicitly generate the elements in the target metamodel.

Figure 6 lists the (pseudo) code to create a matched rule[2]. To explain how it works, we execute it based on the safety case pattern in Figure 2, the guidance metamodel in Figure 3, the development process metamodel in Figure 4, and the weaving model in Table 1. This execution results in the output ATL program shown in Figure 8.

Line 2 in Figure 6 first creates an ATL module, where the GSN metamodel in Figure 7 is declared as the output metamodel to define the structure of the final safety case; and the development process metamodel in Figure 4 is declared as the input metamodel. The reason of using the output GSN metamodel in Figure 7 is that it simplifies the generated ATL program. For example, two GSN goals are simply connected by an association according to Figure 7, as compared to Figure 5 depending on metaclass *AssertedRelationship* to establish the connection between two GSN goals.

Line 3 in Figure 6 then outputs a matched rule (statement 4 in Figure 8) for the root goal node *G1* in Figure 2; Line 4 then outputs the *from* section of the matched rule (statements 5-6 in Figure 8), where it uses the weaving model in Table 1 to get class *System* in the development process metamodel; Line 5 similarly outputs the *to* section of the rule to the output ATL program, i.e., statements 7-8 in Figure 8 which defines an instance *g1* of class *GSN_Goal* in Figure 7.

The next step is to generate the *do* section of the matched rule, which consists of a sequence of binding statements to assign all attributes of *g1* with appropriate expressions. In particular, Line 7 in Figure 6 assigns the *ID* attribute of *g1* as *G1* (statement 11 in Figure 8), where the string *G1* comes from applying *node.getID()* to node *G1* of the safety case pattern; Line 8 produces the binding statements 12-14 in Figure 8 to assign the *Content* attribute of *g1*. Note that the content of a goal node in the final safety case can include a literal string and a role expression. To handle this situation, we follow Table 2 to assign the content of goal nodes.

To decide what sub-goal or strategy nodes should be generated to support a root goal node, Line 9 in Figure 6 checks every relationship that has the root goal node as the source node, and Line 10 finds the target node of such a relationship. In order to do so, Lines 9 and 10 utilizes the properties *hasSource* and *hasTarget* in the metamodel in Figure 5 to find the source and target nodes of each relationship.

Once the target node is found, Lines 11-13 in Figure 6 check its

```
1  module testmodule;
2  create OUT : GSN from IN : GPCAModel;
3
4  rule R1 {
5  from
6    s : GPCAModel!System
7  to
8    g1 : GSN!GSN_Goal (
9   )
10 do {
11  g1.ID <- 'G1'
12  g1.Content <- '';
13  g1.Content <- g1.Content + 'Operational
            safety is verified in  ';
14  g1.Content <- g1.Content + s.ID;
15  g1.subgoal <- thisModule.called1();
16  g1.strategy <- thisModule.called39(s);
17  g1.strategy <- thisModule.called41(s);
18  }
19 }
20 rule called1() {…}
21 rule called39(p0 : GPCAModel!System) {…}
22 rule called41(p0 : GPCAModel!System) {…}
```

Figure 8 The output ATL program

[2] We skip the pseudo code for generating called rules, as it is similar to that for matched rules

type, and create a binding statement to link the source and target nodes. For example, if the type of the target node is *GSN_Goal*, then these two nodes are connected, since the output GSN metamodel in Figure 7 indicates that two *GSN_Goal* classes are connected by the self-association and one of them must has its role name as subgoal. A binding statement is also generated for the target node to assign the invocation of the called rule produced for the target node to the *subgoal* feature of the root *GSN_Goal* node. Situations where the type of the target node is *GSN_Justification*, *GSN_Assumption*, or *GSN_Strategy* are handled similarly.

How to invoke a called rule depends on the format of role expressions. For example, there are three relationships in Figure 2, connecting the root goal node G1 to two strategy nodes S1 and S2 and one goal node G2, respectively. Moreover, Lines 9-12 in Figure 6 output three binding statements to invoke the called rules produced for nodes G2, S1, and S2 (see statements 15-17 in Figure 8).

The called rules generated for each of the rest nodes in the safety case pattern may or may not have parameters. Java Transformation generated a parameter for a called rule only if the corresponding node in the safety case pattern has a role expression. For instance, the called rule *called1()* in Figure 8 is generated for node *G2* in Figure 2. Since *G2* does not have any role expression, no parameter is assigned to rule *called1()*. In contrast, the called rule *called39* generated for node *S1* in Figure 2 has a parameter, since *S1* has a role expression *system*.

Once produced, the ATL program can be executed to generate a safety case for the target project. Executing the ATL program starts with executing the matched rule(s) to produce a root node in the output safety case; if any called rules are invoked during executing the matched rules and the subsequent execution, these called rules are executed to produce the rest nodes in the safety case.

Table 3 GPCA artifacts

GPCA development process metamodel elements	GPCA objects
System	GPCA system
SafetyRequirement	SR1.1, SR1.2, SR1.4, SR1.5, SR1.10, SR3.4.6, SR6.1.3, SR6.1.4
Cause	Flow rate does not match programmed rate Programmed rate too low Dose limit exceed due to too many bolus requests Bolus volume/concentration too high
Hazard	Underinfusion, Overinfusion
Property	Flow rate sensor is equipped Period is 15 minutes Flow rate is less than 90% of the programmed rate
Reference	FDA standard Expertise opinion Previous knowledge

5. CASE STUDY ON THE GPCA PROJECT

The generic patient-controlled analgesia infusion pump project (GPCA) [10] was an open-source project intending to demonstrate the applicability of model-driven development techniques to medical device (software) systems. Its development process is consisted of two steps. The first step includes the activities related

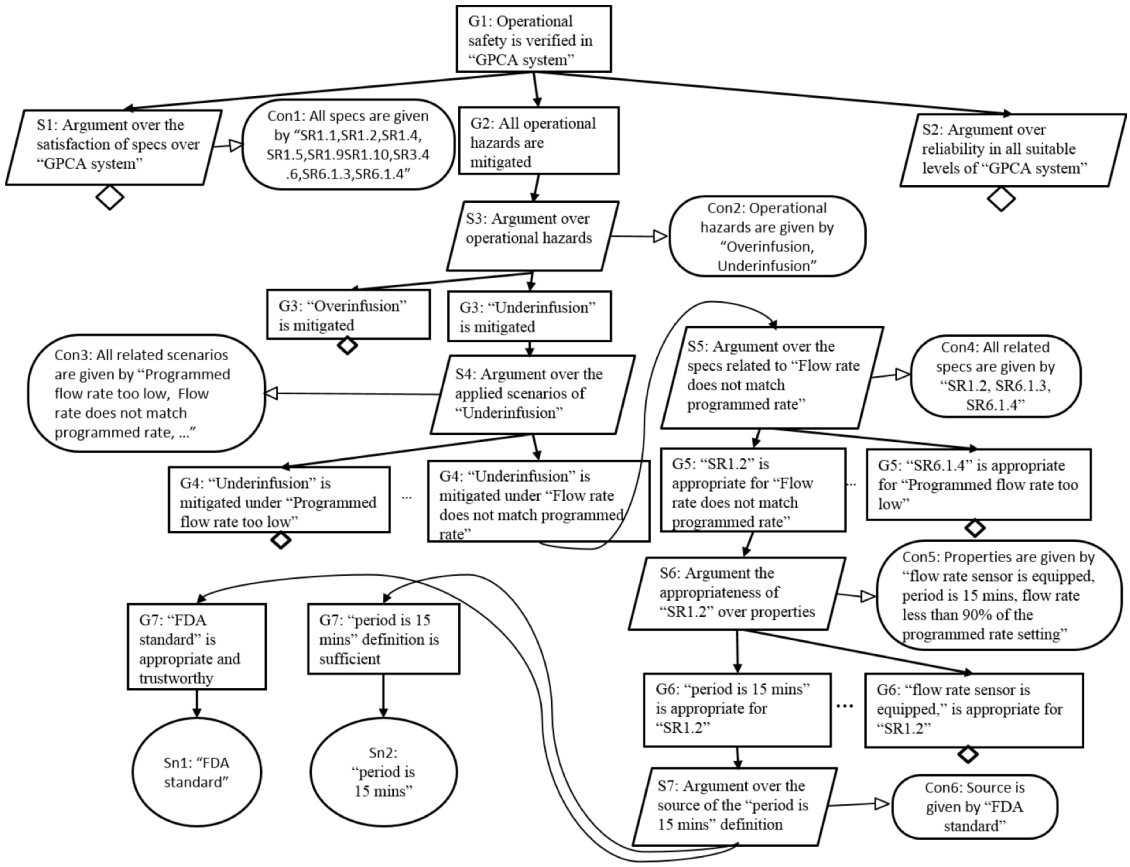

Figure 9 Safety case model of GPCA system

to safety requirements elicitation. Initially, a hazard analysis report was produced to enumerate typical hazards and their causes associated with this type of devices. Next, a set of generic safety requirements were developed to establish the desired safety properties for mitigating the enumerated hazards.

In parallel, a GPCA design model was developed using the Simulink/Stateflow tool [11] to capture the typical use scenarios and operation of patient-controlled analgesia pumps. The safety of the GPCA design model was formally verified against the generic safety requirements, and then the safety-assured design model was translated into the final implementation executable on the target hardware platform. Test cases derived from the safety-assured GPCA design model were applied to the final implementation to ensure that it did not deviate from the design model.

In our case study, we applied SPIRIT to generate a safety case for the GPCA project (partial artifacts of which are shown in Table 3, based on the safety case pattern shown in Figure 2. The GPCA project contains one system type object with the ID of "GPCA system". The two GPCA operational hazards considered in the case study are "Overinfusion" and "Underinfusion". Table 3 includes these two hazards, four potential causes to these hazards considered, as well as 8 safety requirements for mitigating these hazards. Safety requirement properties considered in this example include "Flow rate sensor is equipped", "Period (to trigger the under infusion alarm) is 15mins", and "Flow rate is less than 90% of the programmed rate setting". The references of the safety requirement properties in this example are "FDA standard", "Expertise opinion", and "Previous knowledge".

The complete safe case that SPIRIT generated for the GPCA project can be found in [12], while part of it is shown in Figure 9. In the generated safety case, all role expressions in the nodes in Figure 2 were instantiated by the elements in the GPCA project. For instance, the root goal node G1, which claims "the Operational safety is verified in {system}", was instantiated by "Operational safety is verified in GPCA system". Similarly, node S1 was instantiated by "Argument over the satisfaction of specs over GPCA system" and node S2 was instantiated by "Argument over reliability in all suitable levels of GPCA system". Since node G2 has no role expression, its content was copied to the corresponding node in Figure 9. Due to space limit, we skip the explanation of generating the rest nodes in the safety case.

A benefit of generating a safety case based on the input safety case pattern is that SPIRIT is able to detect structural errors in the generated safety case. This highlights the potential of SPIRIT in facilitating the certification of an infusion pump project against the guidance. In the case study, SPIRIT detected 39 errors in the generated safety case. These errors were mostly missing links from *GSN_Strategy* nodes to *GSN_Goal* nodes when a cause (to certain hazard) is not related to any safety requirement. For example, the *GSN_Strategy* node "S5 Argument over all specs related to Pump programmed but 'start' not pressed" is not linked to any instance of class *GSN_Goal* because the cause, "Pump programmed but 'start' not pressed", is not related to any generic GPCA safety requirement [13]. SPIRIT was able to detect such errors because the GSN metamodel in Figure 7 specifies that a *GSN_Strategy* node must have at least one *GSN_Goal* type property.

6. RELATED WORK

Panesar-Walawege et al. proposed the application of a UML profile to model a standard/guidance [9]. This UML profile is then used as a template to guide how a safety critical system can be developed and reviewed under the standard/guidance. However, the UML profile enforces only one way to model the standard/guidance based on the pre-defined stereotypes. Most systems do not use these pre-defined stereotypes and cannot therefore take advantage of the approach to facilitate the checking of conformance to the standard/guidance. Our work builds upon the notion of safety case patterns to support the development of safety cases for infusion pumps that follow the same development process.

Ayoub et al. [14] proposed a safety case pattern for a system developed using formal methods. The pattern considers the consistency between a design model and its implementation. Our safety case generation approach can complement their pattern as these two approaches target at different phases of a software development process.

Schaetz et al [15] proposed a pattern library that, similar to SPIRIT, facilitates the definition and generation of a safety case for a specific project. But, unlike SPIRIT, they failed to integrate the generation of a safety case into a design model.

Hauge et al. [16] proposed a pattern-based method to facilitate software design for safety critical systems. Under the pattern-based method is a language that offers six different basic patterns as well as operators for composing these patterns. One of the important ramifications of this method is the generation of a safety case, which is connected to the artifacts produced during a development process.

Denney et al. [17] proposed a lightweight approach to automatically create a safety case from a given set of artifacts. It however can only handle development artifacts created in tabular format, as compared to SPIRIT supporting artifacts in any kind of format (as long as these artifacts are defined in the development process metamodel).

7. CONCLUSION

We have presented a framework, SPIRIT, for automatically constructing safety cases for infusion pump systems, with a focus on arguing their conformance to the infusion pump guidance. SPIRIT utilizes a weaving model to allow infusion pump manufacturers to reuse successful safety case patterns in constructing convincing safety cases for their own devices in an efficient manner. This can not only reduce the cost of safety case constructions, but also improve the communication between various stakeholders.

8. ACKNOWLEDGMENTS

The authors would like to thank our FDA colleagues Mr. Paul Jones and Dr. Yi Zhang for many constructive discussions regarding the infusion pump guidance and the challenges facing FDA regulators. These discussions have motivated our research approach and the tool we present in this paper.

9. REFERENCES

[1] U. M. o. Defense, *Defense Standard 00-56 Issue 4 (Part 1): Safety Management Requirements for Defense Systems,* 2007.

[2] FDA, *Infusion Pumps Total Product Life Cycle-Guidance for Industry and FDA Staff,* 2014.

[3] T. Kelly and J. McDermid, "Safety Case Construction and Reuse Using Patterns," in *Safe Comp 97*, Springer, 1997, pp. 55-69.

[4] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *Future of Software Engineering*, 2007.

[5] Eclipse's ATL, Jan 2015. [Online]. Available: http://eclipse.org/atl/.

[6] G. Committee, "GSN Community Standard Version 1," 2011.

[7] Object Management Group (OMG), 2012. [Online]. Available: http://www.omg.org/spec/OCL/2.3.1/PDF/.

[8] R. Hawkins, I. Habli, D. Kolovos, R. Paige and T. Kelly, "Weaving an Assurance Case from Design: A Model-Based Approach," in *IEEE 16th International Symposium on High Assurance Systems Engineering (HASE)* , 2015.

[9] R. K. Panesar-Walawege, M. Sabetzadeh and L. Briand, "Supporting the verification of compliance to safety standards via model-driven engineering: Approach, tool-support and empirical validation," *Information and Software Technology,* vol. 55, no. 5, pp. 836-864, 2013.

[10] B. Kim, A. Ayoub, O. Sokolsky, I. Lee, P. Jones, Z. Yi, R. Jetley and B. Kim, "Safety-Assured Development of the GPCA Infusion Pump Software," in *Proceedings of the ninth ACM international conference on Embedded software*, 2011.

[11] Mathworks, "Simulink Modeling and Simulation Toolsuite," [Online]. Available: http://www.mathworks.com/products/simulink/.

[12] C.-L. Lin, March 2016. [Online]. Available: http://homepages.wmich.edu/~ckt7398/.

[13] D. . E. Arney, R. Jetley, P. Jones, I. Lee, A. Ray, O. Sokolsky and Y. Zhang, "Generic Infusion Pump Hazard Analysis and Safety Requirements Version 1.0," *Technical Reports CIS-893,* 2009.

[14] A. Ayoub, B. Kim, I. Lee and O. Sokolsky, "A safety case pattern for model-based development approach," in *NASA Formal Methods*, Springer, 2012, pp. 141-146.

[15] B. Schaetz, M. Khalil and S. Voss, "A Pattern-based Approach towards Modular Safety Analysis and Argumentation," in *Embedded Real-Time and Software Systems*, 2014.

[16] A. A. Hauge and K. Stølen, "A pattern-based method for safe control systems exemplified within nuclear power production," in *Computer Safety, Reliability, and Security, LNCS Volume 7612*, 2012.

[17] E. Denney and G. Pai, "A lightweight methodology for safety case assembly," in *Computer Safety, Reliability, and Security*, Springer, 2012, pp. 1-12.