4-2018

Support Assurance-Based Software Development for Mission Critical Domains Using the Model Driven Architecture

Chung-Ling Lin
Western Michigan University, calot.lin@gmail.com

Support Assurance-Based Software Development for Mission Critical Domains Using the Model Driven Architecture

by

Chung-Ling Lin

A dissertation submitted to the Graduate College
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
Computer Science
Western Michigan University
April 2018

Doctoral Committee:

Wuwei Shen, Ph.D., Chair
Fahad Saeed, Ph.D.
Bernard Han, Ph.D.
Ralph Tanner, Ph.D.

Support Assurance-Based Software Development for Mission Critical Domains Using the Model Driven Architecture

Chung-Ling Lin, Ph.D.

Western Michigan University, 2018

In the past decades, software development for mission critical applications has drawn great attention not only in various mission critical communities but also software engineering communities. One of the important reasons is that the failure of these systems can lead to some serious consequences such as huge financial loss and even loss of life. Therefore, software certification has become an important activity for mission critical applications in that software assurance for such a system should be certified. With the increasing complexity of a software system in mission critical sectors, certifiers have found hard time to understand how a software system has been developed to ensure software assurance. Assurance cases have been increasingly considered by many emerging standards and government guidelines as an important argument structure for software certification. An assurance case represents an argumentation structure which lays down all arguments made behind each step or activity during a Software Development Life Cycle (SDLC) as well as the relevant artifacts as evidence. In this dissertation, we develop a framework, called SPIRIT, to aid the development and certification of mission critical applications for both system developer and certifier by means of the Model Driven Architecture (MDA). The SPIRIT framework is tripartite and consists of: i) a pattern-based assurance case generation via safety patterns to automatically support assurance cases, ii) maintenance of an assurance case, and iii) a confidence calculation that applies the Dempster-

Shafer theory as a mathematical model to further deduce confidence of an assurance case for the certification purpose. As the results, the SPIRIT framework leverages both developers' and certifiers' capability as a means to develop assurance-based software development for mission critical applications.

TABLE OF CONTENTS

Table of Contents - continued

LIST OF TABLES

LIST OF FIGURES

List of figures - continued

CHAPTER I
INTRODUCTION

In the past decades, software development for mission critical applications has drawn great attention not only in various mission critical communities but also software engineering communities. One of the important reasons is that the failure of these systems can lead to some serious consequences such as huge financial loss and even loss of life. For instance, the failure of Therac-25 radiation therapy machine gives patients overdoses of radiation during radiation therapy, and three injured patients died due to the overdose of radiation. Another example is that the Northeast blackout of 2003 affected over 55 million people for two days. The primary cause of the blackout was a programming error of the alarm system. The last example we mention here is, in 2004, over 400 aircrafts lost the communication to the Los Angeles Air Route Traffic Control Center due to the failure of the voice communication system. Almost 800 flights were disrupted because of the software failure of the software upgrade to a subsystem for the Voice Switching and Control System (VSCS).

As a result, some international standard documents or governmental documents have been released to ensure that a software system for a mission critical application should be well designed to achieve all the goals and not produce any serious consequence to its clients and the public. Most of these standard documents not only stipulate some specific development process to follow but require some strict testing and validation requirements on some system activities and artifacts as well. For instance, DO-178C, Software Considerations in Airborne Systems and Equipment Certification [1] specifies the requirements related to the software testing such as test case selection: *"Specific test cases should be developed to include normal range test cases and robustness, that is, abnormal range test cases"*. Also, DO-178C has requirements on testing methods: *"Requirements-Based Hardware/Software Integration Testing: This testing method should concentrate on error sources associated with the software operating within the target computer environment, and on the high-level functionality. The purpose of requirements-based hardware/software integration testing is to ensure that the software in the target computer will satisfy the high-level requirements"*. Last, DO-178C on Test Coverage Analysis has the following requirement: *"Requirements-Based Test Coverage Analysis. The purpose of this analysis is to determine how well the requirements-based testing verified the implementation of*

the software requirements. This analysis may reveal the need for additional requirements-based test cases.".

Software engineers have spent a great effort to follow the development process and maximally satisfy the requirements on the system artifacts. However, recent study showed that, while great effort has been made, current approaches based on testing and inspection are generally not enough to ensure the correct behavior of a system. Especially, "*Testing, for instance, cannot yield assurance for many kinds of failures related to security and non-deterministic*." According to [2], the correct behavior of a software system can be summarized as software assurance properties, or simply referred to as software assurance, which includes quality-related attributes such as reliability, security, robustness, and safety as well as functionality and performance. Software assurance costs account for 30 to 50 percent of total project costs for most software projects whose behavior still cannot be guaranteed.

Software certification has become an important activity for mission critical systems in that software assurance for such a system should be judged by a third party such as a governmental agency, like Federal Drug Administrator (FDA) in the medical industry, instead of a team of software engineers who have developed a software system. In doing so, some human factors can be maximally reduced and judgement on software assurance can be made based on unbiased way. However, this leads to another challenging issue facing both the software engineering community and the certification community. With the increasing complexity of a software system in mission critical sectors, certifiers have found hard time to understand how a software system has been developed to ensure software assurance due to the following reasons. First, the heterogeneity of artifacts produced during a software development lifecycle (SDLC) hampers the capability of certifiers to understand the rational of how and why these artifacts have been produced. Second, missing traceability information linking the artifacts together further hinders certifiers the understanding how these artifacts have been developed and how they are related to each other according to standard documents. Last, lack of some automatic mechanism impedes the efficiency of certifier to draw a sensible decision on whether a system satisfies software assurance or not. In all, traditional software development for mission critical application should be upgraded to satisfy increasingly strict requirements on software assurance with its certification.

To demonstrate whether a software system has been developed to satisfy software assurance, software engineers should provide an argumentation structure which lays down all arguments made behind each step or activity during an SDLC as well as the relevant artifacts as evidence. The argumentation structure is called an assurance case in mission critical sectors and an assurance case bears considerable resemblance to a legal case, providing a convincing and persuasive argumentation with a wealth of reliable and supporting evidence information. Currently, there are two notations to represent an assurance case. One is Claims, Arguments and Evidence (CAE) [3]. The CAE notation was created by Adelard for presenting a safety argument. The CAE notation consists of three types of elements. A "Claim" element represents a statement asserted in an argument that can be evaluated as true or false. An "Argument" element explains the argument approach to support the parent "Claim". An "Evidence" element reference to an evidence to support a "Claim" or an "Argument.". The other one is called Goal Structuring Notation (GSN) [4]. GSN is a graphical argumentation notation to document an assurance case graphically. GSN represents the individual elements of any safety argument and the relationships between these elements to visualize an argument structure, which is called as a goal structure, to support a claim is true. In GSN, a claim is represented by a "Goal" node. An argument is represented by a "Strategy" node, and an evidence is represented by a "Solution" node. For any goal structure, the purpose is to show how goals are broke-down to sub-goals until a claim can be directly supported by referencing evidence. By using the GSN, it is possible to make clear that how the argument strategies are adopted and the context that a goal is stated. In many industries, GSN is considered as a standard format to represent an assurance case graphically. In this dissertation, we mainly consider GSN in that it has been widely used in safety critical industry.

As mentioned before, traditional software development concentrates on the activities as well as the artifacts to be produced during an SDLC but lacks the rational argumentation behind these activities and their produced artifacts. While recent progress made in the traceability community to emphasize on the establishment the relationship among artifacts produced by different activities, we think it is still far from what an assurance case should provide to convince software assurance in a software system. Therefore, we think traditional software development for mission critical applications should be upgraded to emphasize on an assurance case as a backbone which connects each argumentation to the corresponding activities and their output artifacts during an SDLC. However, manual generation of an assurance case where connection of

each argumentation to the corresponding activities is provided is time consuming and error prone; and in real applications this is real challenging if not impossible. For example, the preliminary assurance case for co-operative airport surface surveillance operations [5] is about 200 pages and is expected to grow as the interim and operational safety cases are created. As a result, some automation in support of assurance cases as a backbone of software development for mission critical applications should be sought.

Model Driven Engineering (MDE) [6] gains the popularity due to the complexity of problem domains where software system should be developed. MDE aims to bridge the gap between the problem domain and its implementation solution and therefore leverages the capability of software engineers to develop a software system in an effective and efficient manner. Thanks to the latest development in MDE, it is possible for MDE to develop a domain specific language tailoring for special needs in a specific domain. In line with the spirit of development of a domain specific language, we apply MDE to support the automation in generation and maintenance of assurance cases as a backbone during an SDLC. The main idea is that MDE enables us to integrate the generation of assurance cases into an SDLC by means of some safety patterns. The application of design patterns is not new and has been employed in software design in the past two decades. Design patterns have been proved to be an effective solution to develop a software system. Likewise, some safety patterns provide an effective method to reuse some successful structures from existing cases which can be from the same sector or even a different sector. The application of design patterns increases the success rate on software assurance as well as shorten a development lifecycle.

Furthermore, MDE enhances the capability to deal with software maintenance. Software evolution becomes an important feature for modern software development. Thus, the cost to maintain a software system accounts for a large portion of the total software cost. In the past, great human effort has been made to make sure that a system can work appropriately in a new evolving environment but the result is still far from being impressive. In this case, some progress especially some automation in support of software maintenance should be made for mission critical applications. Due to the automation in generation of assurance cases, it is feasible to develop some automatic mechanism to support assurance maintenance. In all, we can track an assurance case to a set of system artifacts which are linked to the assurance case. Once some modification is made in a system artifact, we trigger some automatic mechanism to monitor the

validity of the affected node(s), which further evaluates other nodes in the assurance case. Once the validity of a node in an assurance case becomes obsolete, we should be able to flag the invalidity as an error.

An assurance cases provides software engineers with an important means to convince certifiers that a software system has been well designed and implemented to achieve some goals about software assurance. It is the assurance case that transfers the confidence about a software system from software engineers to finally certifiers. Thus, confidence calculation of an assurance case is rather crucial to both software engineers and certifiers to serve the goal of software certification. While many theories on uncertainty reasoning has been proposed in the last several decades, these theories unfortunately cannot be directly applied to the evaluation of an assurance case. One of the reasons is that a software development is more like human-centric human activities. These human-centric activities cannot be accurately modeled as those theories on uncertainty reasoning. Fortunately, artificial intelligence (AI) has made impressive progress in the past decade; and a software system called Alpha Go can defeat the best human player, bearing the latest success of AI. In this dissertation, we employ some machine learning techniques to aid the theories on uncertainty reasoning in the evolution of confidence for an assurance case. We employ various models from the uncertainty theories on the calculation of an assurance case to find when these models are more appropriate in some circumstances.

The main contributions of this dissertation are summarized as follows:

1) Use a UML class diagram to model a standard document and automatically generate a UML profile with OCL constraints from the standard model.
2) Check the conformance of a domain model against the standard.
3) Use a pattern-based approach to support the auto generation of an assurance case.
4) Maintenance and Evaluation of an assurance case.

The remainder of this dissertation is organized as follows. Chapter II discusses the existing approaches for assurance case generation and evaluation. Chapter III introduces the overview of our SPIRIT framework. Chapter IV presents our approach to support the evaluation of the satisfaction of standards of a domain model. Chapter V presents our pattern-based approach to support the generation of assurance cases. Chapter VI discuss the maintenance and evaluation of an assurance case. Chapter VII illustrates a case study example using the coupled

tanks control system to demonstrate the use of our framework and analyze the experimental results. Chapter VIII concludes.

CHAPTER II
RELATED WORKS

2.1 Assurance Case Generation

Construction of an assurance case has been a hot topic since the software-intensive safety-critical systems are becoming popular and increasing complex. How to build a compelling argument draws more attention in safety-critical industry. Many challenges have remained in the assurance case community [7]. Various techniques have been proposed to address the problems and difficulties in the construction of an assurance case. Some researchers have presented the application of pattern in building an assurance case.

Kelly first proposed the assurance pattern as a way of documenting and reusing successful structures [8], the proposed pattern elements and the notations are further included in GSN standard document [4]. Since then, various approaches and techniques have been proposed to leverage the application of safety case patterns in safety-critical domains.

For instance, Ayoub et al. [9] proposed a safety case pattern for a system developed using a formal method. The pattern considers the satisfaction between a design model in terms of a formal notation and its implementation model. In this paper, they proposed a safety pattern, called *from_to* pattern, to support the construction of an assurance case for the Patient Controlled Analgesic (PCA) infusion pump. Their pattern defines two types of variables, the {from} variable and the {to} variable. The {to} variable refers to implementation of a system, and the {from} variable refers to a model of this system. But they did not present how to systematically generate a specific assurance case based on their template.

Schaetz et al [10] proposed a pattern library facilitates the definition and generation of a safety case for a specific project via MDA. In their approach, they defined a set of pattern library elements to represent a pattern which encapsulates all information on an assurance case into a library element and discussed a mechanism for the pattern instantiation. But, they failed to integrate the generation of a safety case into a design model.

Hauge et al. [11] 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 kinds of basic patterns as well as operators for composition. One of the important ramifications of this

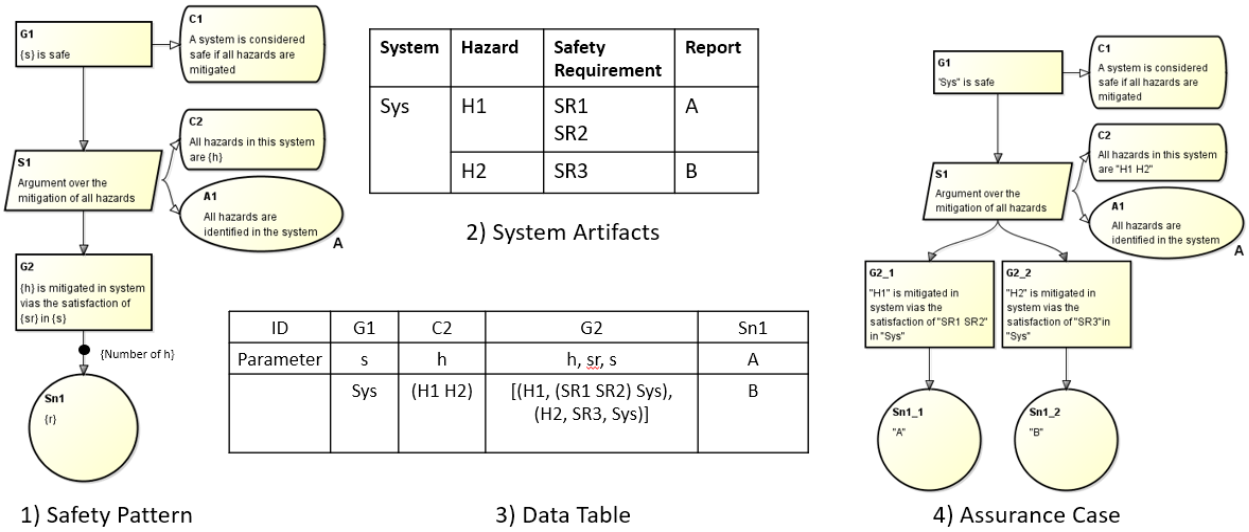
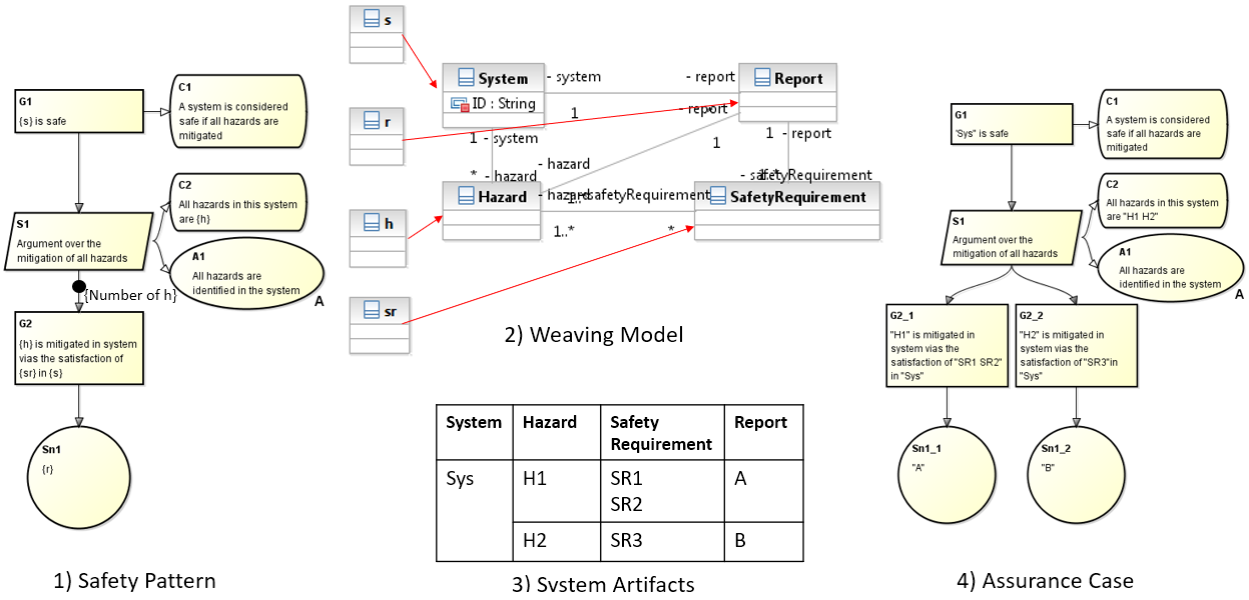


Figure 1 An example to illustrate Denney's assurance case generation approach

method is the generation of safety case, which is connected to the artifacts produced by the same method during a development process.

Denny et al. [12] proposed a lightweight methodology to automatically create the safety case from a given set of artifacts. Their approach was implemented by an algorithm which can create safety case fragments. Actually, their approach can be regarded as an instance of the application of SPIRIT since their algorithm was implemented based on the specific set of documents created in tabular format. They also proposed a new pattern language to generate an assurance case [13]. Their approach required a data table to connect system artifacts and variables in a safety pattern during its instantiation. Obviously, the mapping relationship between all system artifacts and variables would be manually provided. In that case, their approach does not support the evolution of the system. For example, if the relationships between system artifacts are changed, it requires a manual review of the data table and make multiple changes in the corresponding fields. An example to illustrate Denney's assurance case generation approach is shown in Figure 1. In this example, a safety pattern in GSN is defined as shown in Figure 1(1). In a safety pattern, the variables are defined in the pattern to represent abstract entities. For example, a variable {s} in defined in node G1. To generated an assurance case based on a safety pattern, the variables defined in a safety pattern are instantiated by specific system artifacts. In this example, the system artifacts in Figure 1(2) shows a specific System sys, two hazards, H1, H2, three safety requirements, SR1, SR2, and SR3 and two Report A and B are considered as the



System	Hazard	Safety Requirement	Report
Sys	H1	SR1 SR2	A
	H2	SR3	B

3) System Artifacts

Figure 2 An example to illustrate Hawkins's assurance case generation approach

artifacts of a system. In Denney's approach [13], a data table as shown in Figure 1(3) provides the instantiation information for each node in a safety pattern. For example, the second column in Figure 1(3) shows that the variable {s} is instantiated by system artifact "sys" in node G1. As the result, an assurance case is generated in Figure 1(4), and the content of node G1 is replaced by *"sys" is safe* in the assurance case. When an assurance case is generated for a system with a large amount of system artifacts based on a safety pattern, a large data table is required to specify the information for the variable instantiation. Without the integration of a design model, create and maintenance of the data table need to be done manually and become a time-consuming task.

Hawkins et al. proposed a Model-Based approach to weave an assurance case from design [14]. They use a reference model to model all artifacts and their relationship for a specific project and a GSN metamodel to denote an assurance case. When a pattern is instantiated, a weaving table is used to generate a specific assurance case. We use the same pattern with the same system artifacts as shown Figure 1 to illustrate Hawkins's [14] assurance case generation approach in Figure 2. Instead using a data table, Hawkins proposed a weaving model as shown Figure 2(2) to specify the relationships between the variables defined in a safety pattern and the system information metamodel. A weaving model shown in Figure 2(2) consists of an information metamodel shown in the right and the classes derived from each role in the pattern shown in the left. In this example, the information metamodel consists of four classes: *System*, *Hazard*, *Report*, and *SafetyRequirement*, and the classes *s*, *r*, *h*, and *sr* are derived from the

variables in the safety pattern in Figure 2(1). In a weaving model, the associations between the classes derived from variables and the classes in the information metamodel specifies the mapping relationships. In this example, variable {s} maps to class System, variable {r} maps to class Report, variable {h} maps to class Hazard, and variable {sr} maps to class *SafetyRequirement*. With the system artifacts as shown in Figure 2(3), an assurance case shown in Figure 2(4). is generated by instantiating the variables to specific artifacts via the weaving model. But the simple mapping relationship cannot generate a correct underlying reasoning chain in an assurance case because the relationships between variables in the pattern are not specified in their pattern. Therefore, different instantiations can be applied to the pattern based on the same set of system artifacts. In that case, an incorrect instantiation may be applied to instantiate a variable with inappropriate system artifacts when generating an assurance case. In this dissertation, we give a detail discussion of Hawkins's approach and our solution to resolve to problem in section 5.2.

Jee et al. [15] discussed the construction of an assurance case for the pace-maker software using a model-driven development approach, which is similar to ours. However, their approach emphasizes on the later stage of a software development such as a timed automata model as a design model and C code as implementation language. The approach considers the application of the results from the UPPAAL tool and measurement-based timing analysis as evidence.

Attwood et al. [16] proposed an approach to apply a linguistic model of understanding to identify mismatches and provide guidance on composition and integration when constructing an assurance case. Dominguez et al. [17] presented an experience in developing an assurance case for a rebreather system via the Goal Structuring Notation. Ray et al. [18] demonstrate an approach for safety assurance case argumentation based on the Generic Patient Analgesic Pump (GPCA).

2.2 Assurance Case Evaluation

The assurance case evaluation approaches can be categorized into two major categories, the Qualitative Approaches and the Quantitative Approaches. In Qualitative Approaches, Hawkins et al. [19] discussed a new structure, called assured safety argument, to verify the confidence of an assurance case via the confidence argument. An assurance safety argument consists of two types of arguments, the safety argument provides argument or evidence directly related to the system safety, and the confidence argument justifies the sufficiency of the evidence in the safety argument. Their approach identified the Assurance Claim Points (ACP) of a safety case and built the confidence arguments for each ACP to justify the safety case assertions. An example as shown in Figure 3 illustrates the ACPs identified from an assurance case. In this example, three ACPs, ACP1, ACP2, and ACP3 are identified and attached on the links of the assurance case, each of which corresponding to a different type of assertion. ACP1 corresponding to an asserted inference, ACP2 corresponding to an asserted context, and ACP3 corresponding to an asserted evidence. Different confidence arguments, which are represented as assurance cases, are generated for each type of ACP. For an ACP corresponding to an asserted inference as ACP1, the purpose of the confidence claim demonstrates why the satisfaction of sub-claims can support the root claim. For an ACP corresponding to asserted context as ACP2,

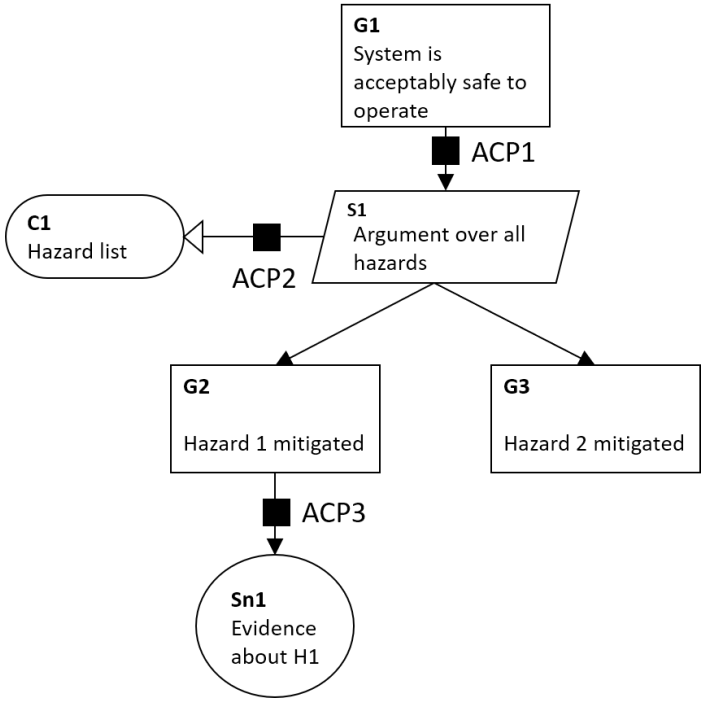


Figure 3 An example of ACP [19]

the purpose of a confidence claim demonstrates the context is appropriate to the assurance case elements where the context is applied. For an ACP corresponding to asserted evidence as ACP3, the purpose of a confidence claim demonstrated the evidence is appropriate to support the claim, and the trustworthiness of the evidence.

Ayoub et al. [20] proposed an approach to systematically identify the deficiency of an assurance case. Similar to Hawkins's approach [19], the components of an assurance case are separate into safety argument and confidence argument. They propose a *common characteristics map* to provide guidelines to support the systematically construction of a confidence argument. The deficits of an assurance case are identified during the construction of a positive confidence argument if any of the branches in a confidence argument is not supported by sufficient evidence.

The Quantitative Approaches apply mathematical models to calculate the confidence of an assurance case. Some authors directly use Bayesian Belief Networks (BBNs) to the safety case. Denney et al. [21] uses BBNs to build the confidence argument for the safety case and measure the confidence in the claims via the computation of the joint distribution based on the quantified sources presented in the safety argument. Zhao et al. [22] discussed how to convert an assurance case to a BBN model discussed the guidelines to measure the confidence for each claim in an assurance case based on confidential probability table (CPT) assigned to the non-leaf node of BBN.

Some assurance case confidence calculations are based on the Dempster–Shafer (D-S) theory of evidence developed by [23]. In the D-S theory based approaches, belief, disbelief, and uncertainty of each node in the confidence network are explicitly computed. Guiochet et al [24] proposed a confidence calculation model to model an assurance case in GSN and measure the confidence of the confidence calculation model based on the belief and uncertainty value of each node in the confidence network. Three different types of arguments, simple argument, alternative argument, and complementary argument are defined in their approach to propagate the confidence in a safety case. Wang [25] proposed a similar assurance case confidence measurement approach based on the D-S theory. According to [25], the confidence calculation of a claim depends on the trustworthiness of its sub-claim and the appropriateness of the sub-claims supporting the claim. The trustworthiness of a claim is given by a 3-tuple, denoting the confidence of the claim. The appropriateness of sub-claims to support their parent claim

considers how these sub-claims make contribution to support their parent claim. And Duan et al [26] proposed the confidence measurement approach based on the Baconian probability with the beta distribution to visualize the Baconian probability representing confidence.

The existing confidence calculation approaches for assurance cases requires a manual configuration of the values of some variables in a calculation model to calculate the confidence value of the root claim. For instance, Wang et al. [25] used the Dempster Shafer (D-S) theory as their main calculation model in order to deduce the confidence of an assurance case, where they manually set up the values for all variables in the calculation model. In that case, assurance case evaluation requires the certifiers' domain knowledge and experience, and the certification of a system has been conducted manually. As a result, the development process can be hindered, deployment of new systems can be delayed, and developers' creativity can be stifled.

CHAPTER III
APPROACH OVERVIEW

The V-model [27] is commonly used to represent the software development process in different domains. A V-model consists of three major phases: the verification phases, coding phase, and the validation phases. The verification phases consist of 1) Requirements analysis sub-phase collects the requirements from analyzing the needs from users. 2) The system design sub-phase generates software specifications to satisfy the requirements and design the complete system. 3) The Software Architecture sub-phase breaks down the design of system into modules, each of which takes up different functionalities of the system. 4) The module design sub-phase defines the actual logics of a module. The coding sub-phase implements the module design to a program. The validation consists of 1) Unit testing sub-phase executes the unit test plans at code level or unit level to verify each module can function correctly. 2) Integration testing sub-phase tests verifies the internal modules can coexist and communicate with each other. 3) System testing sub-phase checks the functionality and the communication of the entire system to verify all system requirements are met. 4) Acceptance testing sub-phase verifies the system meets user's requirements.

But the traditional software development process cannot support software for mission critical applications due to two major reasons. First, the system design of mission critical application must satisfy the standard documents. Second, an assurance case has been increasingly considered by many emerging standards or governmental guidelines as an important argument structure for certifier's certification of a mission critical application, which is called assurance-based software. To support the two aspects of the mission critical application development in the satisfaction of standard documents, generation, maintenance, and evaluation of an assurance case. We proposed a framework, called SPIRIT, illustrated in Figure 4.

The first goal of our proposed framework aids to check the conformity of a domain model against the requirements in a standard document. The main purpose of this goal is to ensure and assure the satisfaction of a standard document during an entire development lifecycle for a mission critical software system [28]. Because most of the standard documents in different

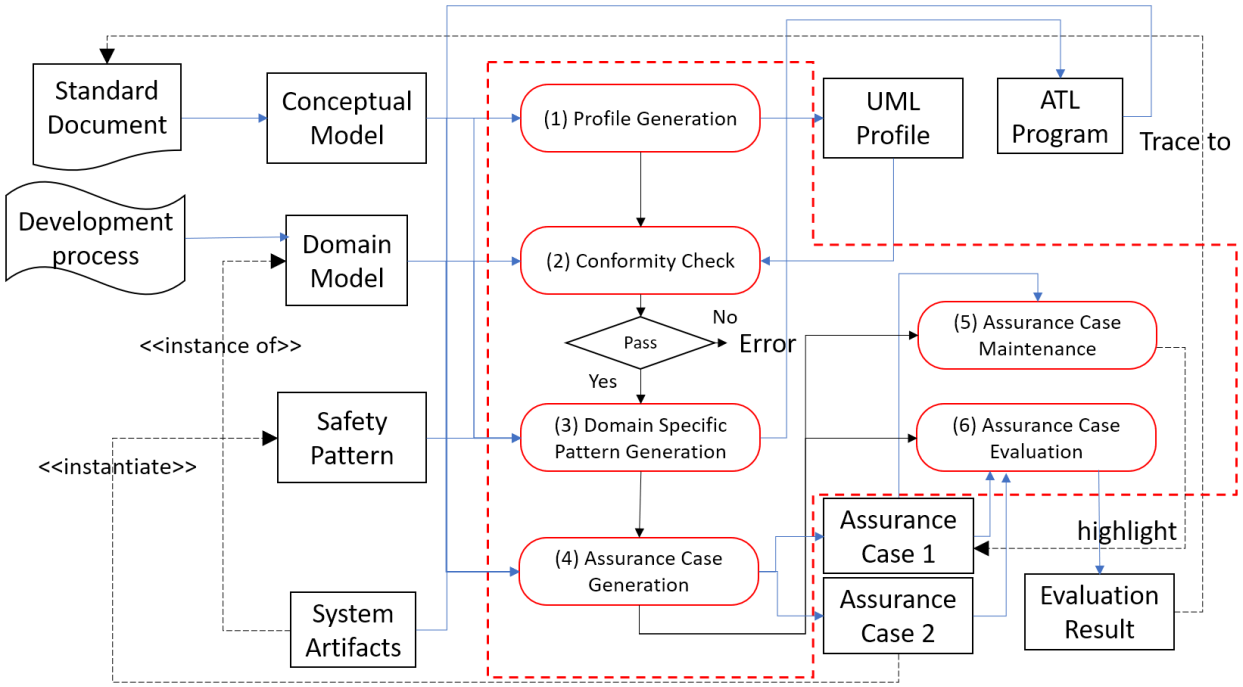


Figure 4 SPIRIT framework

domains are written in a natural language, the assessment of conformance is not straightforward because the description of the standard document is ambiguous [28].To achieve this goal, the SPIRIT framework automatically converts a standard conceptual model to a Unified Modeling Language (UML) [29] profile. Object Constraint Language (OCL) [30] constraints are generated and attached to the UML profile. When a system developer provides a domain model as input, SPIRIT can check the conformity of the domain model against the standard. That is to ensure that a domain model satisfies all standard requirements.

The second goal of SPIRIT is to aid both engineers and certifiers to develop and certify a mission critical system in an effective and efficient manner. We thus address the following problem statement: *Given a specific development process employed by engineers, while software assurance is finally made by a human judgement of fitness for use, how can we provide some maximal automation to generate an assurance case which not only lays out an argument structure with supporting evidence to support a claim about software assurance but also evolves during software maintenance*. To attack the challenging problem statement, SPIRIT provides the automation of the creation and maintenance of an assurance case in support of development and certification of software intensive systems. The Assurance Case Activities target on three major

aspects: 1) Automatically support of assurance case generation, 2) Maintenance of the generated assurance case, and 3) Evaluation of an assurance case.

The SPIRIT framework consists of six major activities, each of which targets on a specific task during a software development lifecycle for a mission-critical system: 1) Profile Generation activity takes a UML conceptual model as input and generates a UML profile as output. A UML class diagram models the concepts of the standard documents is thus called a conceptual model of a standard. The purpose of this activity is to formalize the standard documents written in natural language using a UML profile, so that the constraints defined in standard documents can be applied to a target domain model to verify if the design of a domain model satisfies the standard constraints. 2) The Standard Conformity Check activity takes a domain model developed based on the standard profile as input and generates an evaluation report as output. Domain model is a "*visual representation of conceptual classes or real-world objects in a domain of interest*" [31]. In SPIRIT, a domain model is represented as a UML class diagram to model the abstract components or artifacts in a specific domain. The purpose of this activity helps the user to verify the proposed domain model conforms to the standard document in the early phases of the development life cycle. Once a standard profile is generated by SPIRIT, a user can apply the generated profile to their proposed domain model. SPIRIT verifies the proposed domain model conforms to the standard document by checking if the proposed domain is valid instance of the standard profile or not. In any constraints defined in the standard profile is violated by the proposed domain model, then the proposed domain model is an invalid instance of the standard profile. In other words, some of the requirements defined in the standard documents are not satisfied. 3) The Domain Specific Pattern Generation activity takes the domain model, generic safety patterns, mapping tables, and configuration tables as input and generates a complete safety pattern for a specific domain represented as an Atlas Transformation Language (ATL) [32] program as output. This complete safety pattern is called a domain specific pattern. The purposes of this activity are: First, the framework generates a complete safety pattern via the combination of the generic safety patterns. Second, the framework ensures that a complete safety pattern can be appropriately instantiated based on the provided domain model. 4) The Assurance Case Generation activity takes the domain specific pattern, the domain model, and the system artifacts as input and generates an assurance case as output via the execution of the ATL model transformation. A domain model models the abstract concepts for different

systems within the same domain, the system artifacts is represented as a UML object diagram, which is an instance of the domain model. 5) The Maintenance activity monitors any modification in a domain model or any artifacts related to the project and carries out the impact calculation to figure out how the previous assurance case is affected by highlighting the affected nodes in the assurance case. 6) The goal of the Evaluation activity is to automatically deduce values for the assessment parameters such as the trustworthiness and appropriateness, i.e., disjoint contributing weights, and use the parameters to evaluate an unknown assurance case. The inputs of this activity are two assurance cases generated from the same safety pattern(s) for the same project in activity 4). The first assurance case is determined to be acceptable as the training data of the activity. Since the second assurance case is derived based on the safety pattern where the first assurance case is derived, the derived parameters are applied to the second assurance case to finally deduce its confidence.

CHAPTER IV
CONFORMITY CHECK

4.1 Standard Conceptual Model

To illustrate the standard document in a formal representation, we use a UML class diagram to model the concepts of the standard documents, and this class diagram is thus called a conceptual model of a standard. In a conceptual model, each concept is modeled as a UML class. And the relation between concepts are represented as associations between the corresponding classes. The method that how to elicit the concepts from a standard document is proposed by Panesar-Walawege et.al in [33].

5.1.2.h Each **system requirement** allocated to software should be *traceable to one or more* **high-level requirements**.

Figure 5 DO178C 5.1.2.h [1]

For example, section 5.1.2.h of the DO-178C standard [1] in Figure 5 shows that "Each system requirement allocated to software should be traceable to one or more high-level requirements." Terms "system requirement" and "high-level requirement", highlighted in bold, are elicited from the description of the standard to produce two classes *SystemRequirment* and *HighLevelRequirement* respectively. Likewise, an association called "*traceToHighLevelReq*" is produced based on the text highlighted in italic in Figure 5 to represents a "trace" relationship between system requirement" and high-level requirement. A conceptual model example derived from the text is shown in Figure 6. The multiplicity indicates that how many instances of "*HighLevelRequirement*" can be related to an instance of "*SystemRequirement*". The multiplicity value "1..*" defines the range composed by the lower bound and upper bound value. In this

Figure 6 A conceptual model example

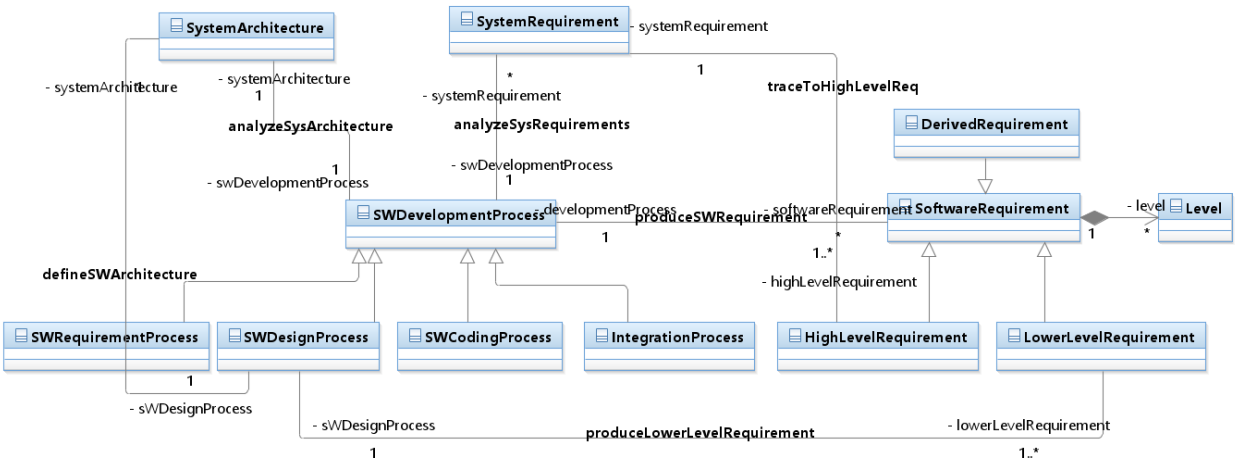


Figure 7 A partial conceptual model derived from DO178-C Chapter 5

example, the lower bound is 1, the upper bound * indicates an unbound upper bound value to represent the "one or more" relations described in the standard document. As the result, a partial conceptual model derived from DO-178C chapter 5 is shown in Figure 7.

The **software development processes** *are*

- **Software requirements process.**
- **Software design process.**
- **Software coding process.**
- **Integration process.**

Figure 8 DO178-C software development processes [1]

The DO178-C chapter 5 addresses the requirements of the software development process. Based on description of the standard document shown in Figure 8, the software development processes are Software requirements process, Software design process, Software coding process, and Integration process. Therefore, four concept classes, "*SWRequirementProcess*", "*SWDesignProcess*", "*SWCodingProcess*", and "*IntegrationProcess*", are generated in the conceptual model, which are corresponding to the four processes illustrated in the document shown in Figure 7. And these four classes are considered as sub-classes of a general concept class "*SWDevelopmentProcess*."

Similarly, a list of concepts and associations are extracted from the standard document shown in Figure 9. A concept "*SoftwareRequirement*" is extracted from the standard. "*SoftwareRequirement*" class has two sub-classes to address the "*HighLevelRequirement*" and "*LowerLeverRequirement*". An association between the "*DevelopmentProcess*" and

Software development processes *produce one or more* **level** of **software requirements. High-level requirements** are *produced directly through analysis* of **system requirements** and **system architecture.** Usually, these high-level requirements are further developed during the **software desigh process**, thus *producing one or more* successive, **lower levels of requirements.**

Figure 9 DO178-C software requirement [1]

"*SoftwareRequirement*" reflects the "produce" relation between the development process and software requirement. The next sentence describes that development process analyze system requirements and system architecture to produce the High-level requirements. Therefore, two concepts "*SystemRequirement*" and "*SystemArchitecture*" are generated and the associations between "*DevelopmentProcess*" and these two classes reflect the "analyze" relationship extracted from the description. The last sentence indicates that the lower levels requirements are produced during the software design process. A corresponding association between "*DesignProcess*" and "*LowerLevelRequirement*" is generated in the conceptual model. Similar to the previous example shown in Figure 6, the multiplicity "1..*" in this association shown that the "one or more" relations that the software design process produce one or more lower levels of requirements.

Another section in DO-178C chapter 5 shown in Figure 10 describes the activities involved in the software design process. Since the relation to address the low-level requirement are develop during the software design process is captured in our conceptual model from the previous section, we only add a new association between "*DesignProcess*" and "*SoftwareArchitecture*" to reflect the "define" relation between design process and software architecture. The last sentence describes another type of requirement. The derived requirement is produced during each software development process. Therefore, another concept, "*DerivedRequirement*", is created in the conceptual model, and "*DerivedRequirement*" is

The development of software architecture involves decision made about the structure of the software. During the **software design process**, the **software architecture** is *defined* and **low-level requirements** are *developed*. Low-level requirements are software requirements from which Source Code can be directly implemented without further information. Each **software development process** may *produce* **derived requirements.**

Figure 10 Software design process [1]

considered as a subclass of "*SoftwareRequirement*". Since the "produce" relationship is extracted from the standard description shown in Figure 9, and an association is created between "*SWDevelopmentProcess*" and "*SoftwareRequirement*". Therefore, there is no need to create additional association in the conceptual model.

4.2 UML Profile of a Standard Document

The previous section proposed an approach to formalize a standard document to a UML class diagram. To achieve the automatically conformity check of a domain model to a standard, an approach proposed in [33] illustrates a manual method to generate a UML profile from a standard conceptual model. In our approach, SPIRIT provides a feature to generate a profile from a conceptual model automatically. UML provides a generic extension mechanism which allows a user to define a customized model for a specific domain and ensure that the customized model is still a valid UML model. The primary extension construct in a UML profile is Stereotype. "*A stereotype defines how an existing metaclass may be extended, and enables the use of platform or domain specific terminology or notation in place of, or in addition to, the ones used for the extended metaclass.*" [29] The OCL constraints defined in a profile are evaluated when a profile is applied to a UML model. When the SPIRIT framework automatically builds a UML profile from a standard conceptual model, all classes, associations, and properties defined in the conceptual model are transferred to stereotypes in a profile as output. The OCL constraints are automatically generated from the multiplicity of association ends and attached to the corresponding stereotypes in the output profile. To achieve the UML profile generation feature, SPIRIT takes the standard conceptual model generated from the previous section as input represented as a UML file. SPIRIT reads the contents of the input conceptual model in UML file

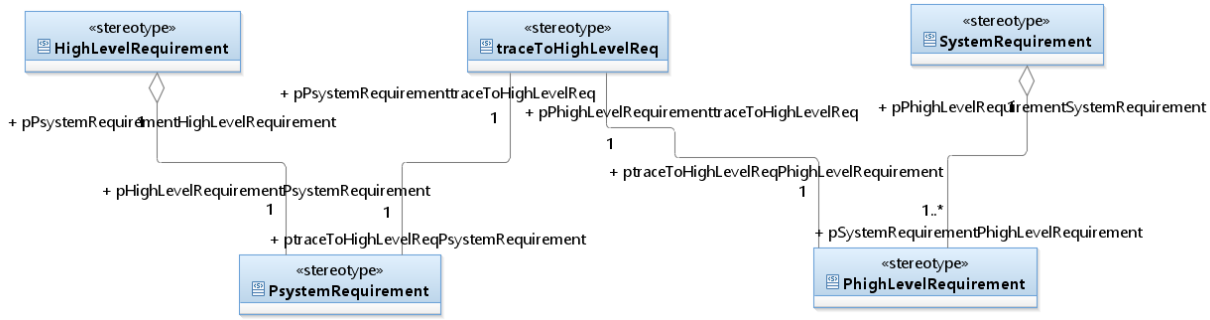


Figure 11 A UML profile example

via the eclipse UML2 API [34]. When a UML element is identified from the input model, SPIRIT reads the name and the type of this element and generates a stereotype which extends the corresponding UML metaclass. To generate the OCL constraints in the output profile, SPIRIT defines several OCL constraint templates to serve different types of standard requirements. When SPIRIT captures certain requirements from the standard conceptual model, the corresponding OCL constraint template is selected to generate an OCL constraint statements and attach to the corresponding stereotype. In the implementation of SPIRIT, we consider three different types of OCL constraints and three templates are used accordingly. The three types include 1) Multiplicity constraint, Type 2) Association constraint, and Type 3) Type constraint. The Multiplicity constraint enforces that an association ends multiplicity satisfies the constrained value defined via the lower bound and the upper bound attributes. The Association constraint enforce that a class has an association associated to a specific class. The Type constraint enforces that an association is set between two correct classes via checking the type of the two association ends of an association. A profile example shown in Figure 11 is based on the conceptual model shown in Figure 6. Four OCL constraints are generated in the profile:

The constraint 1 shown in Figure 12 is an example of our Type 1) constraint which enforces that a class applied stereotype "*SystemRequirement*" must have an association associated with at least one class applied stereotype "*HighLevelRequirement*". An OCL expression shown in Figure 12 specifies the stereotype where an OCL constraint is attached via a keyword "context". In constraint 1, the reserved word "context" denotes a context, i.e. *SystemRequirement"*, where the OCL constraint is applied. When SPIRIT generates an OCL constraint, the OCL constraint is automatically attached to a specific stereotype and only the constraint body is generated by SPIRIT. The reserved word "self" refer to the contextual instance [30], in this case, the instance of "*SystemRequirement*" stereotype. The OCL constraint 1 checks

```
context: SystemRequirement
self.base_Class.ownedAttribute-> select(p|p.association<>null)->collect(c:Property |
c.association)-> collect(a|a.memberEnd.class)->collect(c|c.getStereotypeApplications())->
select(s|s.oclIsKindOf(DO178CProfile::HighLevelRequirement))->size()>=1 or
self.base_Class.allParents().attribute->select(p|p.association<>null)->collect(c:Property |
c.association)-> collect(a|a.memberEnd.class)->collect(c|c.getStereotypeApplications())->
select(s|s.oclIsKindOf(DO178CProfile::HighLevelRequirement))->size()>=1
```

Figure 12 OCL constraint 1

```
context: traceToHighLevelReq
self.base_Association.memberEnd->collect(p:Property|p.class.getStereotypeApplications())-
>select(s|s.oclIsKindOf(DO178CProfile::SystemRequirement))->size()=1 and
self.base_Association.memberEnd->collect(p:Property|p.class.getStereotypeApplications())-
>select(s|s.oclIsKindOf(DO178CProfile::HighLevelRequirement))->size()=1
```

Figure 13 OCL constraint 2

whether the class, where the stereotype has been applied, has at least one property whose type is applied stereotype "*HighLevelRequirement*" to ensure that any instance of system requirement is related to at least one high-level requirement instance.

The OCL constraint 2 shown in Figure 13 is similar to constraint 1. In this case, the constraint checks that the class applied stereotype "*HighLevelRequirement*", has a property whose type is applied stereotype "*SystemRequirement*".

Constraint 3 shown in Figure 14 is an example of our Type 3) constraint. This constraint checks that an association applied stereotype "*traceToHighLevelReq*" is between two classes, one class has stereotype "*SystemRequirement*" applies to it. And the other class has stereotype "*HighLevelRequirement'* applies to it. Constraint 4 shown in Figure 15 is an example of our Type 2) constraint. In this example, the constraint checks that a class applied stereotype

```
context: HighLevelRequirement
self.base_Class.ownedAttribute->collect(c:Property | c.association)->select(a:Association |not
a.getAppliedStereotype('DO178CProfile::traceToHighLevelReq').oclIsUndefined())->size()
=1 or self.base_Class.allParents().attribute->collect(c:Property | c.association)-
>select(a:Association |not
a.getAppliedStereotype('DO178CProfile::traceToHighLevelReq').oclIsUndefined())->size()
=1
```

Figure 14 OCL constraint 3

```
context: HighLevelRequirement
self.base_Class.ownedAttribute-> select(p|p.association<>null)->collect(c:Property |
c.association)-> collect(a|a.memberEnd.class)->collect(c|c.getStereotypeApplications())->
select(s|s.oclIsKindOf(DO178CProfile::SystemRequirement))->size()=1 or
self.base_Class.allParents().attribute->select(p|p.association<>null)->collect(c:Property |
c.association)-> collect(a|a.memberEnd.class)->collect(c|c.getStereotypeApplications())->
select(s|s.oclIsKindOf(DO178CProfile::SystemRequirement))->size()=1
```

Figure 15 OCL constraint 4

"*HighLevelRequirement*" has an association which has stereotype "*traceToHighLevelReq*" applies to it.

4.3 Conformity Check

The definition of conformity is "conformity *is a noun and means* compliance with standards, rules, or laws" [35]. In our approach, we consider that a domain model conforms to a standard if the domain model is a valid instance of the standard. A UML model is considered as a valid instance of a UML profile if all constraints defined in the profile are satisfied by the model. When a UML profile is automatically generated from a standard document via SPIRIT, a user can apply the standard profile to their domain model, SPIRIT provides a feature to check the conformity of the domain model to the standard via the evaluation of the constraints. When any OCL constraints are not satisfied by a domain model, a constraint violation report is generated by SPIRIT, indicating the violated constraints and their related domain classes.

CHAPTER V
ASSURANCE CASE GENERATION

5.1 Introduction to Assurance Case

Assurance case is a structure to provide a convincing and persuasive argumentation with a wealth of reliable and supporting evidence information. The definition of an assurance case is: "*A reasoned and compelling argument, supported by a body of evidence, that a system, service or organization will operate as intended for a defined application in a defined environment.*" [4] An argument is defined as "*A connected series of claims intended to establish an overall claim.*" [4]. GSN defines six types of elements shown in Figure 16 (a) and two types of relationships shown in Figure 16 (b). A "goal" node represents a claim which contains a statement that can be assessment to be true of false. A "strategy" node describes the inference that how a goal node is supported by its supporting goal node(s). A "solution" node reference an evidence item. A "context" node presents a contextual artifact which describes the background information. An "assumption" node presents an unsubstantiated claim. A "justification" node presents a statement of rationale. The "*SupportedBy*" represents an inferential or evidential relationship. And the "*InContextOf*" represents a contextual relationship.

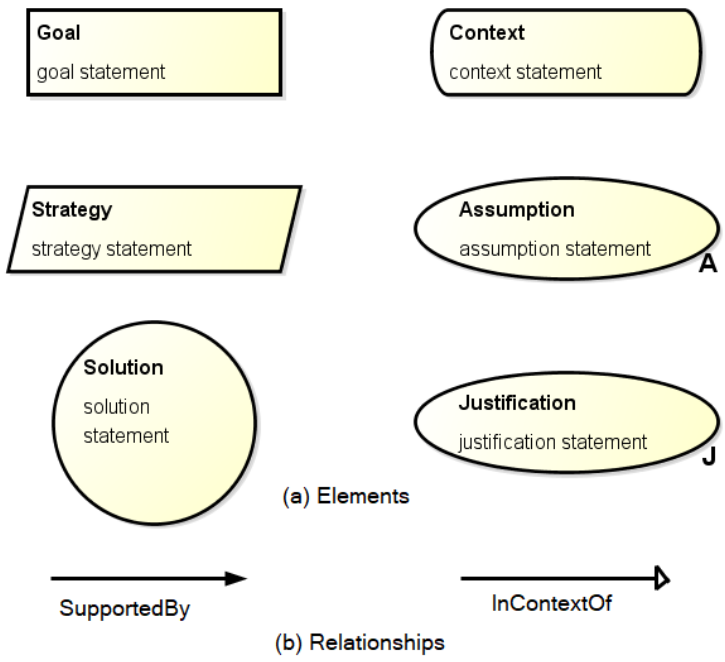


Figure 16 GSN notations

A GSN assurance case example illustrated in the GSN standard is shown in Figure 17. In this assurance case example, a goal node G1 contains a root claim of the argument structure which claims the control system is acceptably safe to operate. An explicit statement of the context is required for a claim in order to evaluate a statement is true or false [4], the context statements are provided in the context nodes in an assurance case. In this example, context nodes C1 and C2 are link to node G1 to explain the operating roles and the definition of the control system. While a claim cannot be directly evaluated to be true or false, a high-level goal can be decomposed or re-stated to a number of sub-goals. In this example, node G2 and G3 are sub-goals of G1, and G1 is the parent goal of G2 and G3. An assertion is made in an assurance case that when the sub-goals of a goal node are true, it is sufficient to establish that the claim in the parent goal is true [4]. In this assurance case example, the claim of G1 is re-stated to G2 and G3, which claims that all identified hazards are eliminated or sufficient mitigated, and the software in the control system are developed to safety integrity level (SIL) appropriate to the hazards involved. Context nodes C3, C4, and C5 are documented in the assurance case to interpret the

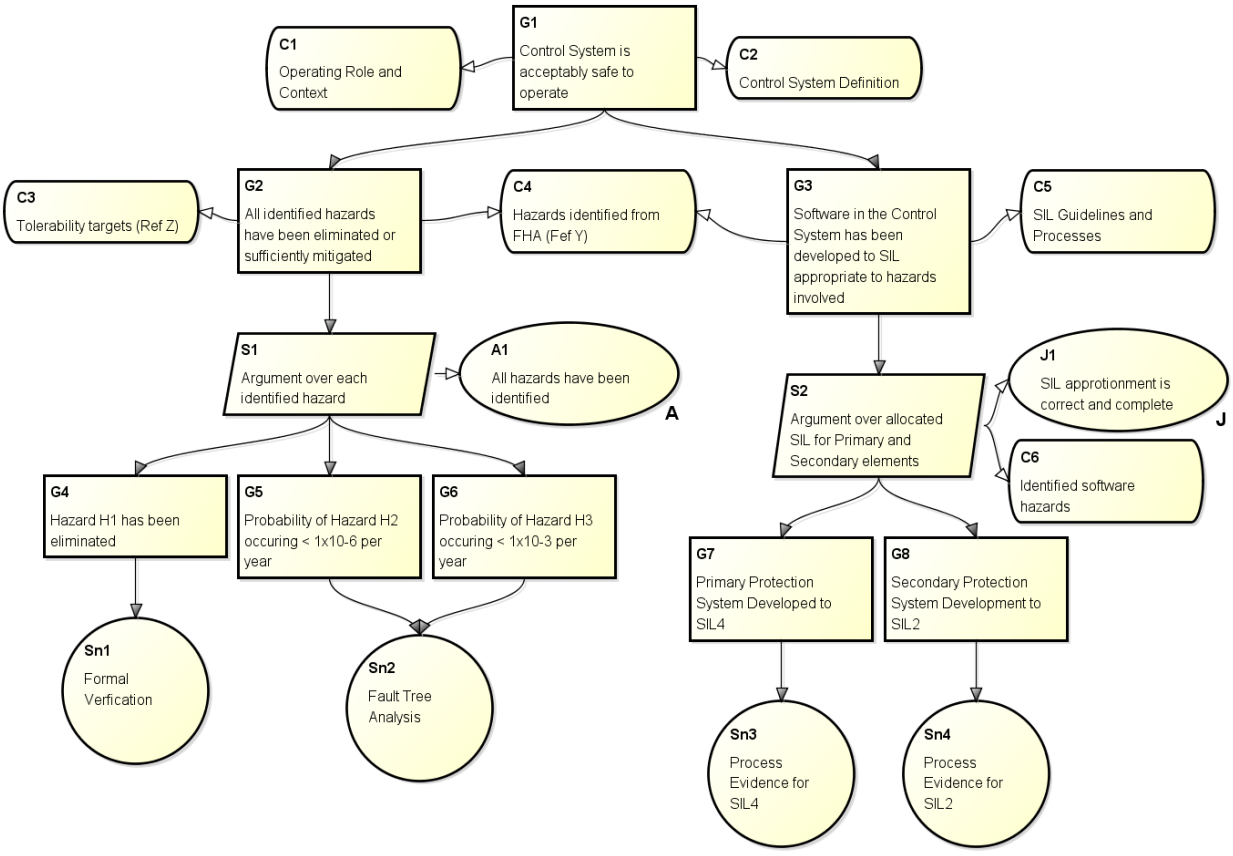


Figure 17 An assurance case example from GSN standard [4]

tolerability targets, method used to identify the hazards, and the guidelines and processes of safety integration level. When describing how a claim is supported by sub-claims, a strategy node can be used to document the reasoning step [4] and link the parent goal and its sub-goals. In this example, goal nodes G4, G5, and G6 are documented in the assurance case to support node G2, each of which claims a specific hazard is eliminated or mitigated. The strategy node S1 is documented in the assurance case to state the relationship between G2 and its sub-goals. Similar to G2, node G3 is supported by G7 and G8 via strategy S3, which claims the correspond SIL level of the primary protection system and secondary protection system respectively. Sometimes a claim or a strategy relies on an assumption to hold valid, an assumption is explicitly documented as an assumption node in GSN [4]. In this example, assumption node A1 is documented which assumes that all hazards in this system are identified. A justification provides the explanation that why a claim or strategy is considered acceptable [4]. In this example, justification node J1 justifies the use of SIL by asserting that SIL apportionment is correct and complete. When evidence can support the truth of a claim, the evidence is documented as a solution node in an assurance case. In this assurance case example, solution node S1 shows that the formal verification result support the truth of the elimination of hazard H1, the fault tree analysis result shown in solution node S2 support the truth of the mitigation of hazard H2 and H3, and process evidence for SIL 4 and SIL 2 are considered as evidences to support the truth SIL level of the primary and secondary system.

5.2 Pattern-Based Assurance Case Generation

Based on our experiment of the assurance case generation for different projects, we noticed that different assurance cases can share similar argument structures. Therefore, pattern-based assurance case generation approaches are applied in various environments. An assurance case pattern is called safety pattern. In the pattern-based assurance case generation approaches, safety pattern represents a pattern of generic argument that can be reused. The GSN extension to support argument patterns is defined in GSN standard [4]. In summary: 1) GSN pattern allows a user to define “roles” in a node statement to represent an abstract entity. These roles are instantiated by system artifact when an assurance case is generated for a specific system. 2) The Structural abstraction allows a user to define the “*Multiplicity*” and “*Optionality*” on GSN

relations. 3) The Modular extensions allow a user to represent the relationships between interrelated modules of argument.

Great efforts have been made to construct an assurance case to support software certification for safety critical systems. Denny et al. proposed a new methodology in the aviation domain to automatically assembly a lower level safety case based on the verification of the safety-related properties of the autopilot software in the Swift Unmanned Aircraft System (UAS) [36]. The methodology applied the annotation schema, which models the information such as the definition of a program variable during the verification of the safety-related properties, to automatically generate a safety case fragment. But, the annotation schema cannot be employed to model system artifacts and a high-level safety case thus is manually assembled.

Especially inspired by rapid and successful application of design patterns in software development, researchers have proposed safety patterns in construction of an assurance case. Hawkins [14] proposed a pattern-based approach to generated an assurance case based on mapping the roles defined in a safety pattern to the classes defined in the information metamodel via a weaving model generated based on the information metamodel. An example shown in Figure 22 illustrates how an assurance case is generated based on a safety pattern. In this example, four roles {s}, {h}, {sr}, and {r} are defined in the safety pattern shown in Figure 18(1). A weaving model shown in Figure 18(2) consists of an information metamodel shown in the right and the classes derived from each role in the pattern shown in the left. In this example,

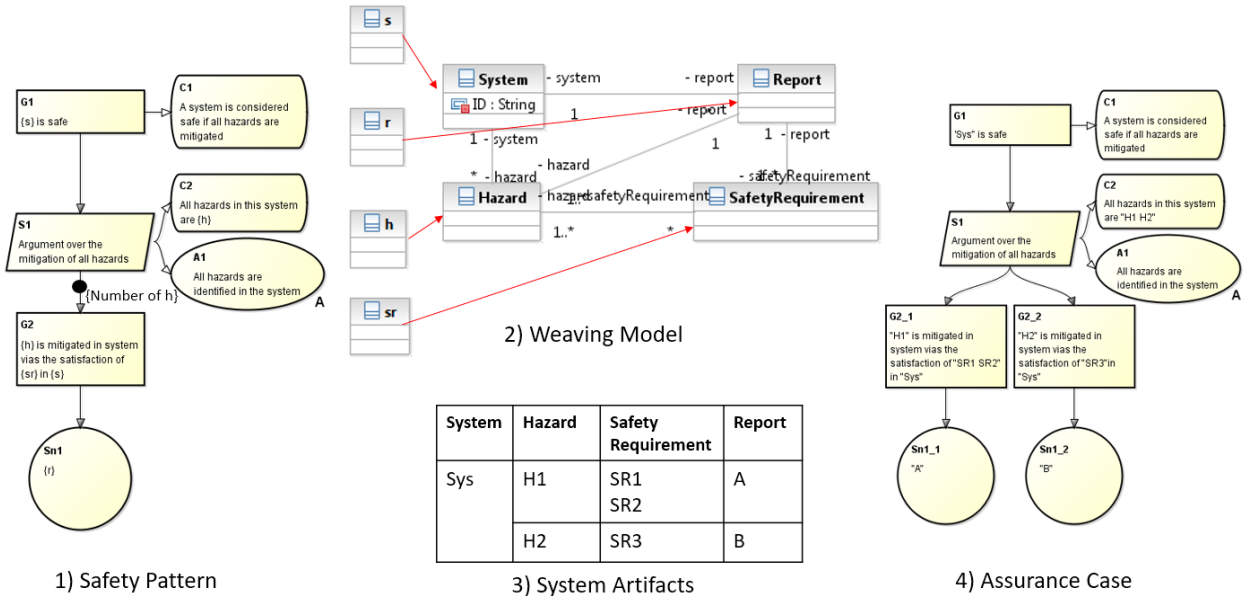


Figure 18 A weaving assurance case generation approach

the information metamodel consists of four classes: *System*, *Hazard*, *Report*, and *SafetyRequiremen*t, and the classes *s*, *r*, *h*, and *sr* are derived from the roles in the safety pattern in Figure 18(1). In a weaving model, the associations between the classes derived from roles and the classes in the information metamodel specifies the mapping relationships. In this example, role {s} maps to class *System*, role {r} maps to class *Report*, role {h} maps to class *Hazard*, and role {sr} maps to class *SafetyRequirement*. A table shown in Figure 18(3) illustrates a set of instances of the information metamodel represents the system artifacts of a specific system. In this example, *sys* is an instance of *System*, which contains two *Hazard* instances, *H1* and *H2*. Hazard *H1* links to two *SafetyRequirement*, *SR1* and *SR2*, and one *Report*, *A*. And Hazard *H2* links to one *SafetyRequirement*, *SR3*, and one *Report*, *B*. Based on the mapping relationship defined in the weaving model, when a set of system artifacts are given, a project specific safety case can be generated from the safety case pattern shown in Figure 18(4) by instantiating the roles to the system artifacts. In this example, the role {s} in node G1 is instantiated by *sys*, role 'h" is instantiated by *H1* and *H2*. The multiplicity, "number of h", attached on the link between S1 and G2 generates two goal nodes G2_1 and G2_2, and each node claims a specific hazard is mitigated. In the first goal node, G2_1, role {h} is instantiated by hazard *H1*, and role {sr} is instantiated by safety requirements *SR1*, *SR2*. In the second goal node, G2_2, role {h} is instantiated by hazard *H2* and role {sr} is instantiated by safety requirements *SR3*. G2_1 and G2_2 are supported by two different solution nodes Sn1_1 and Sn1_2. In Sn1_1, role {r} is instantiated by report *A*. In Sn1_2, role {r} is instantiated by report *B*. In this approach, the relationships between roles are not specified in the pattern. Therefore, more than one instantiations can be made to generate different assurance cases based on the same weaving model and system artifacts. An assurance case shown in Figure 19 is generated from the same safety pattern, weaving model, and system artifacts shown in Figure 18(3). In this assurance case, role {sr} in node G2_1 and G2_2 are instantiated by *SR1*, *SR2*, and *SR3*, which are the safety requirements related to the system Sys. But this instantiation of role {sr} is not appropriate since hazard *H1* addressed in G2_1 has no relation to safety requirement *SR3*. And hazard *H2* addressed in G2_2 has no relation to safety requirement *SR1* and *SR2*.

On the other hand, all current pattern-based approaches keep the argumentation structures of two safety patterns when combination. But, our experience in safety critical domains has shown that reusability of a safety pattern is increased when the argumentation structure can be

augmented when combing with a second safety pattern in another application scenario. To provide the flexibility of the use of safety patterns, SPIRIT supports two types of pattern combinations based on the type of node involved in the combination.

5.3 Extension of GSN Pattern

To address the problems in the existing approaches, we proposed our extension of the GSN pattern in the following aspects: 1) SPIRIT contains the definition of two types of roles, the expression of a role is combined with OCL expressions to illustrate explicit relations between roles and the multiplicity of relations. With the use of mapping tables, we provide an unambiguous mechanism to support the generation of an assurance case based on the instantiation of a safety pattern. 2) We propose the safety pattern catalog which provides not only a safety pattern in the extended GSN but also includes the description and applied metamodel to which the pattern can be applied. 3) We define a formal representation to illustrate a safety

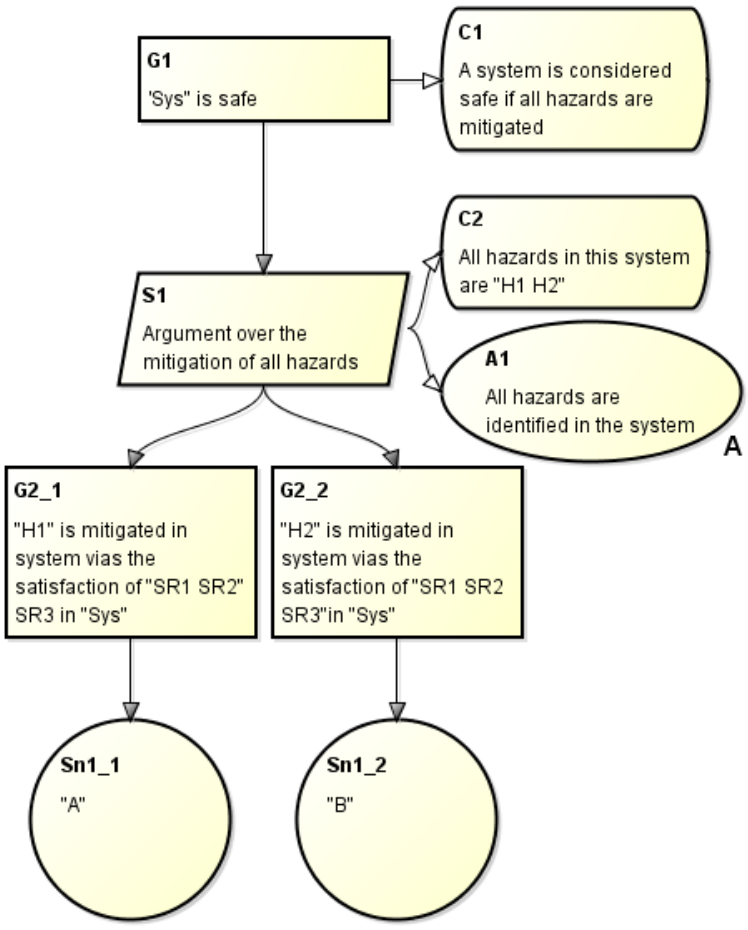


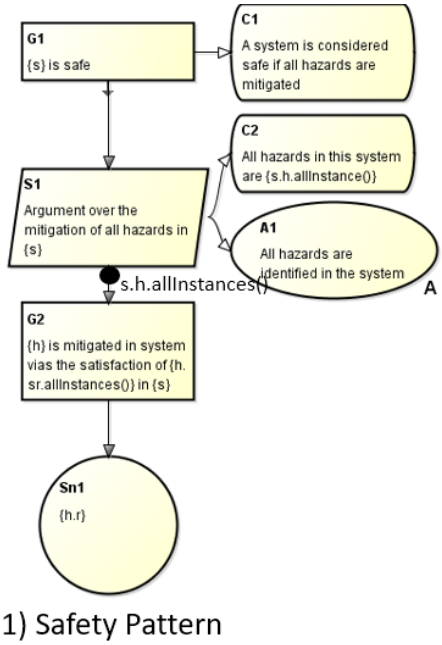
Figure 19 An assurance case example with inappropriate instantiations

pattern. 4) SPIRIT provides a pattern combination mechanism which allows a user to create a domain specific pattern based on the combination of multiple safety patterns.

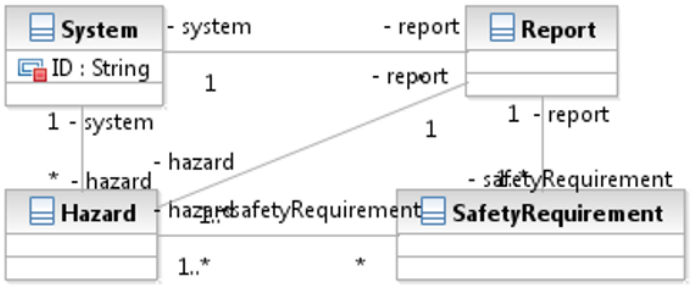
5.3.1 Extension of Pattern Syntax

We extend the syntax of GSN to accommodate the introduction of new variables as shown in Figure 20(1). Basically, there are two types of variables. One is given by the {s} variable where s is mapped to a class in an applied metamodel. We also extend the syntax of GSN via some expressions in the OCL. For instance, in Figure 20(1), expression "*s.h*" denotes the objects of class H via the role name h at the association end on class H which is connected to class C via an association. Furthermore, "*s.h.allInstance()*" denotes all objects of class H related to a given instance, given by c, of class C. In doing so, some variables are linked together and so are the corresponding instances required for an assurance case. The other type is given by {$a} where variable $a denotes a string during the instantiation of a pattern.

With the extension of the GSN pattern syntax, the safety pattern in Figure 18 (1) is represented in the new notation shown in Figure 20(1). In our approach, the mapping relation is provided via a configuration table as shown in Figure 20 (3). Therefore, role {h.sr.allInstances()} defined in node G2 denotes a set of safety requirements related to a specific hazard. With the same system artifacts as shown in Figure 18(3), only one valid instantiation of the pattern is made by SPIRIT to generate an assurance case as shown in Figure 18(4).



1) Safety Pattern



2) Domain Model

Role	Conceptual model	Domain model
{s}	System	System
{sr}	AccuracySpec	SafetyRequirement
{h}	OperationalHazard	Hazard
{r}		Report

3) Configuration Table

Figure 20 Safety pattern with syntax extension

The existing pattern-based approach [14] does not consider the standard documents in the generation of an assurance case. The weaving model illustrated in [14] only consider the relation between the pattern and the information metamodel for a specific project or domain. The relation between guidance and pattern and the relation between guidance and the project are not captured by the weaving model. In order to connect the safety case pattern, standard, and domain model, we proposed a two steps mapping approach to build the relations between Guidance, Pattern, and Project in SPIRIT. An example of the two steps mapping approach is shown in Figure 21. In this example, a node in the safety pattern contains a role {x}. The first step maps a role to a concept class defined in the conceptual model. In this example role {x} is mapped to class *HighLevelRequirement*. The second step maps a concept class to a domain class defined in the domain model. In this example, *HighLevelRequirement* is mapped to *SystemProperty*. Based on the two steps mapping, a derived mapping relationship is built which maps role {x} to domain class *SystemProperty*.

5.3.2 Safety Pattern Catalog

Unlike the other existing safety pattern-based approach, we propose a safety pattern catalog is necessary for the application of the safety pattern in that the catalog provides not only a safety pattern in the extended GSN but also includes the description and applied metamodel to which the pattern can be applied. The purpose of the safety pattern catalog representation is to provide the description of what a safety pattern does and thus the guidance about how to use a safety pattern. With the use of safety pattern catalog, a safety pattern can clearly demonstrate how it is applied in a concrete environment via an applied metamodel combined with the description. A pattern catalog consists of the following sections: 1) The graphical assurance pattern section, 2) The Description section contains the description of the overall purpose of a pattern and the description of each role defined in the safety pattern, and 3) The metamodel

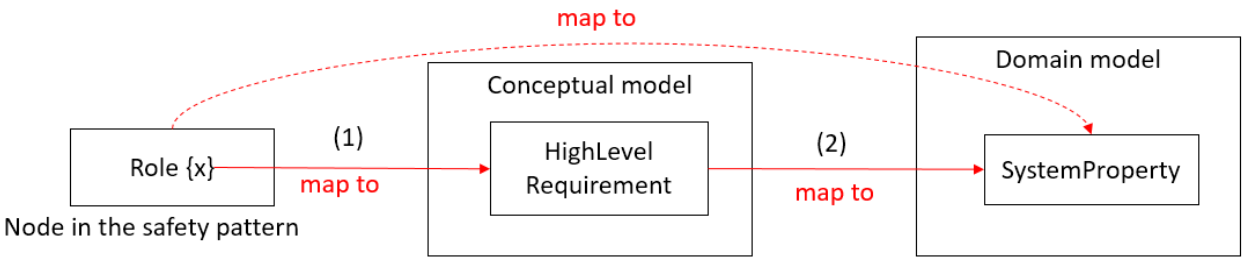


Figure 21 A two steps mapping example

section helping the explanation of what the safety pattern does given in the description section. Figure 22 shows a safety pattern catalog whose pattern in the extended GSN is supported by SPIRIT. In this example, the pattern catalog describes a safety pattern to re-state a root goal "{s} satisfies property {$a}" to sub-goal(s) "{h} satisfies property {$c}". To reason the restatement of the root goal to the sub-goal(s), a strategy node describes the relationship between the root goal and the sub-goal(s) is based on the relationship between {s} and {h}. The context node in this pattern displays an object of Class *s* and its related objects of Class *h* as described in the previous section. The metamodel section of this example illustrates the relationship between variable {s} and variable {h}, it provides the guidance to the user to use this pattern. In this example, if *s* is mapped to class *C1* and *h* is mapped to class *C2*. An association is required between *C1* and *C2*.

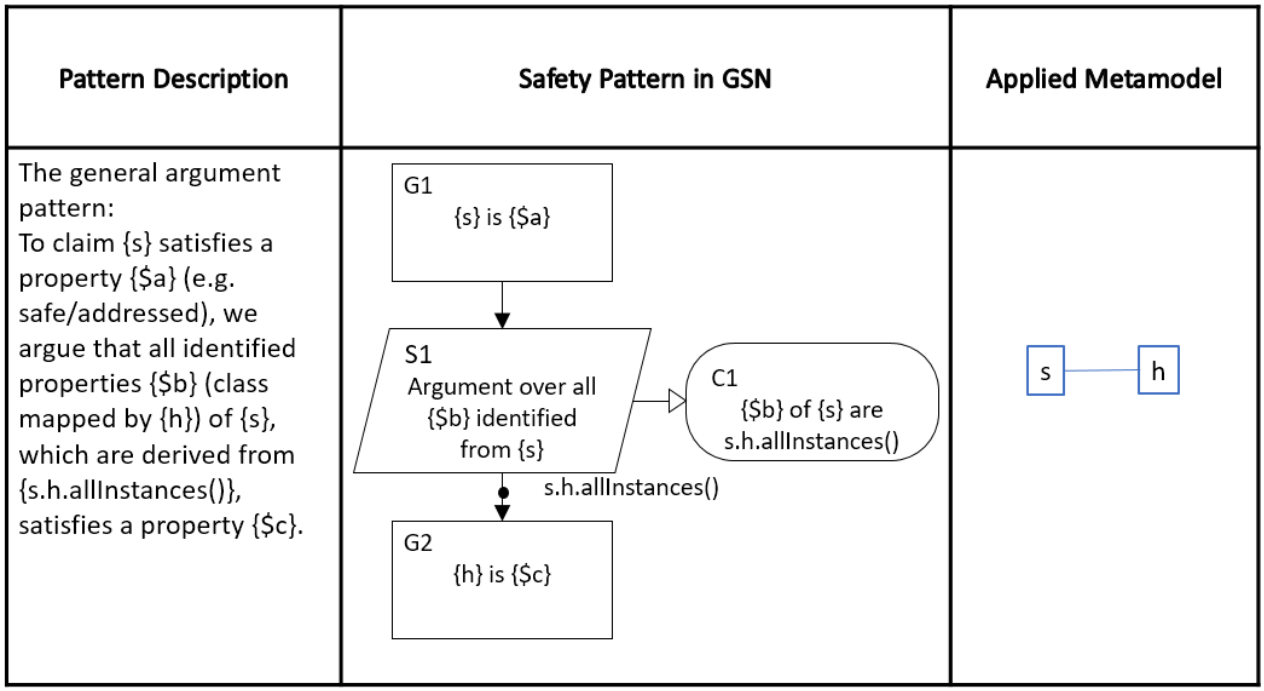


Figure 22 Safety pattern catalog example

5.3.3 Pattern Definition

To illustrate a safety pattern formally, we define some variables which are shown in Figure 23. First, all literals are denoted as the LI set. Generally speaking, a variable used in a safety pattern can be a simple literal string, denoting a class or type in a metamodel. Let literal

1. All Literals are denoted by set LI.
2. All variables are denoted by set V; A variable v is defined as follows:
   - A string s is a variable.
   - If v is a variable and s∈LI is a string, then v.s is a variable.
   - if v is a variable and op is an operation returning a collection in the Object Constraint Language, then v.op is a variable.
3. All Variable nodes are denoted by set VN; A variable node is denoted as n = <L, R> where R ⊆V, L ⊆ LI, and R ≠ Φ.
4. All normal nodes are denoted by set N ; A normal node n is a node where R = Φ.
5. All links is denoted by set LINK; A link: l= <sn, tn, m> where sn ∈ N ∪ VN, tn ∈ N ∪ VN and m is a variable v which ends with an OCL expression.
6. All patterns are denoted by set P; A pattern is denoted as p = <VN', LINK'> where VN' ⊆ N ∪ VN and LINK' ⊆ LINK.
7. All domain classes are denoted by set DC; A domain class is denoted as dc.
8. All mapping relationships are denoted by set MP; A mapping relationship is denoted as mp = <v,dc> where v ∈ V and dc ∈ DC. A mapping mp maps variable v to domain class dc.
9. All configuration tables are denoted by set C; A Configuration Table is denoted as c = <p, MP'> where p ∈ P and MP'⊆ MP. A configuration table contains a set of mapping relationships for pattern p.
10. All pattern connections are denoted by set PC; A pattern connection is denoted as pc = < sp,tp,n,m> where
    - sp ∈ C - A configuration table which includes a source pattern and a set of mapping relationship.
    - tp ∈ C - A configuration table which includes a target pattern and a set of mapping relationship.
    - n ∈ sp.p.VN' - a node n in the source pattern links to the root node of tp.p.
    - m - an OCL expression showing how a link between n and the root node of tp.p is set up.

Figure 23 Definition of variables

string *s* be such a variable. Then OCL expression *s.f* denotes another variable and so does *s.f.allInstances()* which denotes a set of instances of a Class associating with the class mapped by variable *s*. A formal definition of a variable is shown in Figure 23. All variables are denoted as the V set. A variable node n is denoted as a tuple <L, R> where L represents a subset of all Literals and R represents a subset of all Variables V. As a convention, for a general tuple t=<f1,f2>, we use t.f1 to denote the first element of the tuple while t.f2 the second element of the tuple. A variable node is used in a safety pattern. When R, in node n, is empty, i.e. Φ, the variable node n become a normal node so n is called a node. A link l is denoted as a tuple <sn, tn, m>, where sn ϵ N ∪ VN and tn ϵ N ∪ VN represent the source node and the target node of a link respectively, and m is an OCL expression showing how a link is set up. Next, a pattern p is denoted as a tuple <VN', L'>, where VN' represents a set of all variable nodes and normal nodes in p, and L' represents a set of all links in p. In order to support the instantiation of a safety pattern in a specific project via its domain model, we introduce the following two concepts. A mapping relationship mp is denoted as a tuple <v, dc> which maps variable v to the class dc in a domain model, and all mapping relationships are denoted as the MP set. A configuration table c is denoted as a tuple <p, MP'>, where p ϵ P is a safety pattern and MP' is a set of all mapping relationships in c. A configuration table plays a crucial role when a safety pattern is instantiated.

1. LI = {“is”, “Argument over all”, “identified from”, “of”, “are”, “is addressed”}
2. V = { s, $a, $b, h, s.h.allInstances(), p}
3. G1 = < {“is”}, {s, $a} >, S1 = <{“Argument over all”, “identified from”}, {$b, s}>, G2 = < {{h, $c})>, C1 = <{“of”, ”are”},{$b, s, s.h.allInstances()}>}
4. N = {}
5. l1 = <G1, S1, 1>, l2 = <S1, G2, s.h.allInstances()>, l3 = <S1, C1, 1>}
6. vn1 = {G1, S1, G2, C1}, p1 = <vn1, {l1, l2, l3}>

Figure 24 Formalized safety pattern 1

All configuration tables are denoted by the C set. In general, a configuration table c shows a safety pattern *c.p* as well as a set of mapping relationships for all variables in p. Next, to combine two safety patterns during the instantiation time, we introduce the concept of a pattern connection. A pattern connection pc is denoted as a tuple <sp, tp, n, m>, where sp ϵ C specifies a source configuration table, tp ϵ C is a target configuration table, n $\in$ sp.p.VN’ is the node of the pattern defined in source configuration table sp to be connected to the root node of the target pattern, and m is an OCL expression showing how two patterns are connected together. The formal representation of the safety pattern in Figure 22 is shown in Figure 24.

5.4 Assurance Case Generation

The assurance case generation flowchart is shown in Figure 25 which provides a detail view of the “Domain Specific Pattern Generation” and “Assurance Case Generation” activities illustrated in the SPIRIT framework shown in Figure 4. As mentioned in the previous section, our pattern-based assurance case generation approach is based on the instantiation of the variables defined in the safety pattern to concrete system artifacts. To ensure that a safety pattern can be correctly instantiated by the system artifacts, the sub-activity 1) shown in Figure 25 takes the Generic Assurance case patterns, Configuration tables, and a Domain model as inputs to validate if all of the roles defined in the assurance case patterns are appropriately mapped to domain classes via the mapping information defined in the configuration tables. If a pattern is instantiable, the sub-activity 2) aims to generate a domain specific pattern by replacing all variables in a safety pattern with domain classes via a configuration table. Sub-activity 3) employs pattern connection tables to generate a composite domain specific pattern via the combination of the generic safety patterns. As a result of activities (1), (2) and (3) a complete

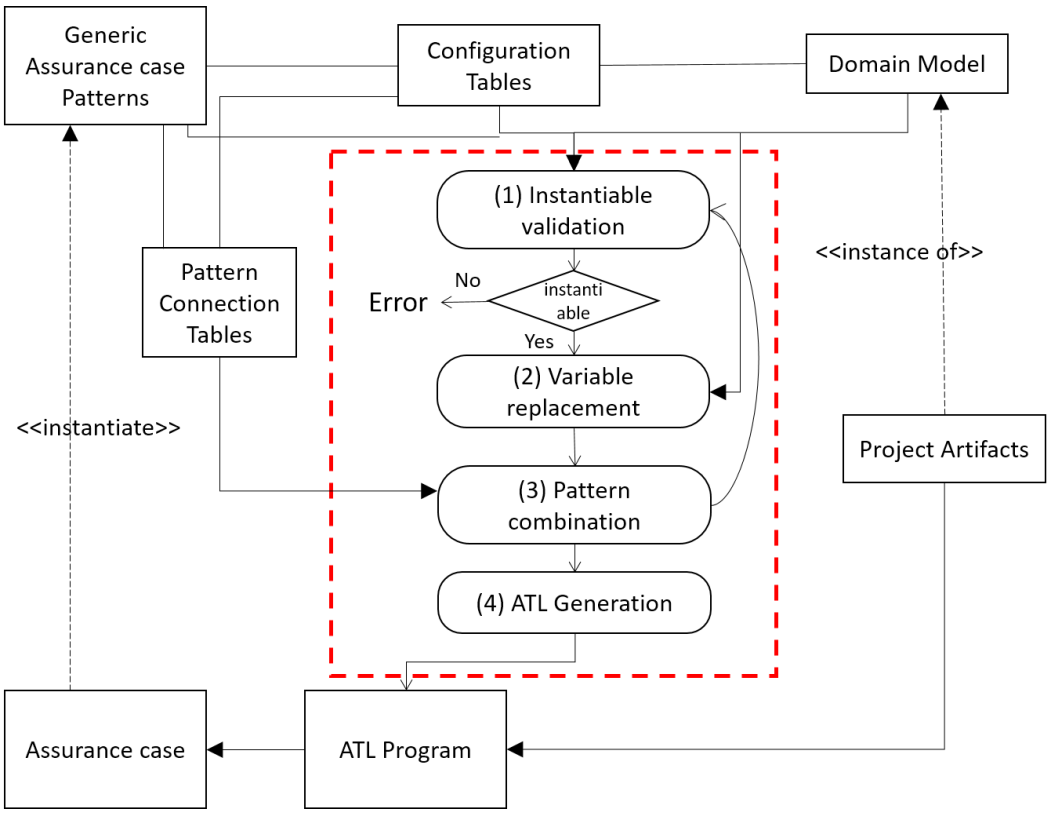


Figure 25 Assurance case generation flowchart

domain specific pattern is produced by an ATL program. For a specific project, the ATL program takes all artifacts, which are a valid instance of a domain model, as input and generates an assurance case as output.

5.4.1 Instantiable Validation and Variable Replacement

The instantiable validation sub-activity 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 appropriately mapped to elements in the domain model. During the validation sub-activity, SPIRIT uses configuration table 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 domain model according to the configuration table.
- C2: For a derived role r.x, if r and x are mapped to classes C_1 and C_2, in the domain model, then C_1 and C_2 should have (at least) one association between them.

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 domain model. 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).

An example shown in Figure 26 illustrates the different types of constraint violations. A node in this pattern example contains two roles {X} and {X.Y} shown in Figure 26 (1). A configuration table shown in (2) violate constraint C1. In this example, the mapping relationship between role {Y} and a domain class is not specified in the configuration table. A configuration table shown in Figure 26(3) fixes the error. In this configuration table, a new row is added to address the mapping relationship between {Y} and a domain class, *SystemDesign*. The domain model shown in Figure 26(4) violates constraint C2 based on the safety pattern in (1) and the configuration table (3). In this example, an association between *Hazard* and *SystemDesign* is required. The domain model shown in Figure 26(5) violates constraint C4 based on the safety pattern in (1) and the configuration table in (3). In this example, the role {X.Y} does not end with allInstances(). Therefore, the multiplicity at the end of class SystemDesign must be 1. A domain model example that can pass the instantiable validation is shown in Figure 26(6).

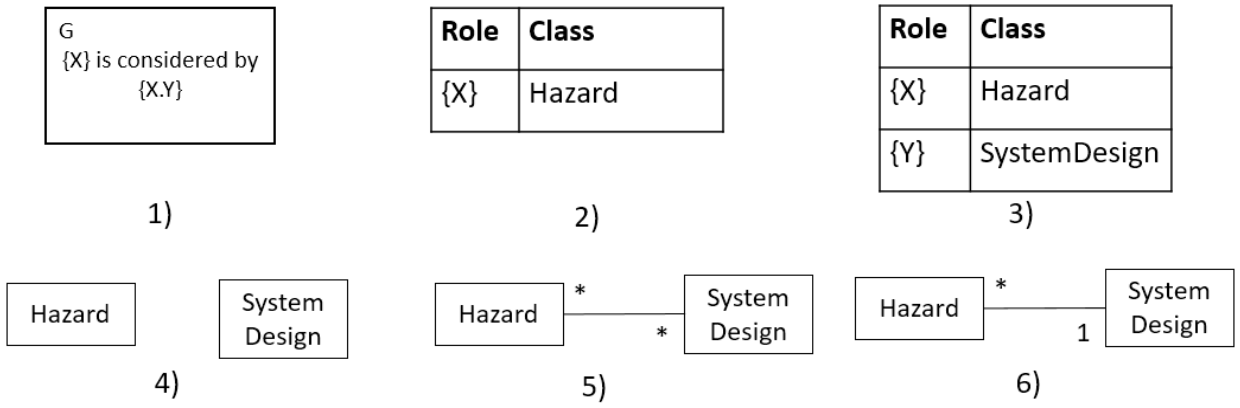
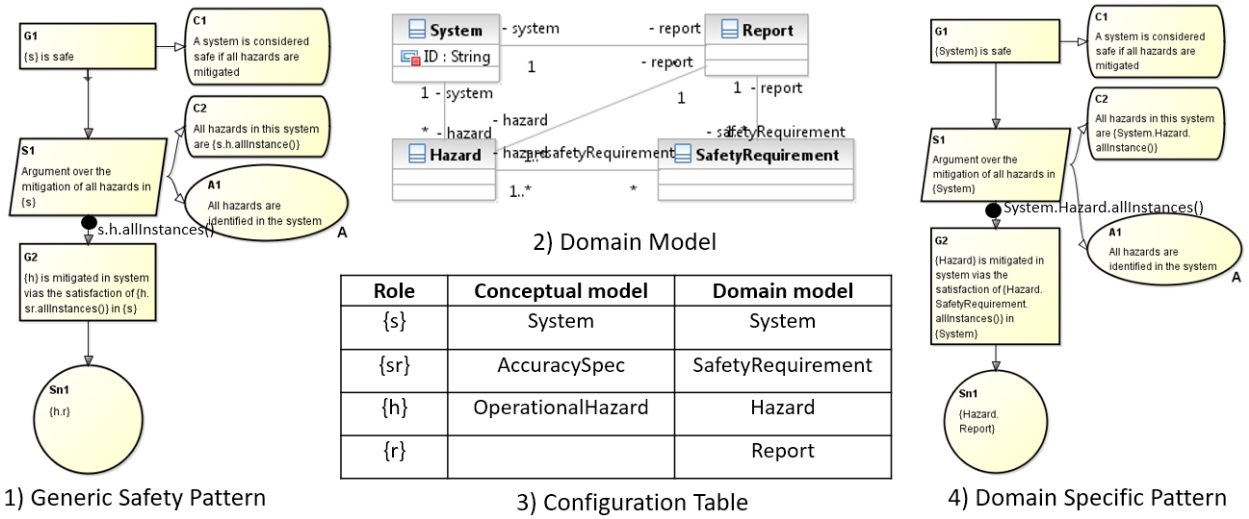


Figure 26 Instantiable validation example



Role	Conceptual model	Domain model
{s}	System	System
{sr}	AccuracySpec	SafetyRequirement
{h}	OperationalHazard	Hazard
{r}		Report

Figure 27 Domain specific pattern

Once a pattern is confirmed as instantiable by SPIRIT, a complete safety pattern for a specific domain is produced and this is called a domain specific safety pattern. All variables in each safety pattern are replaced with the classes in a specific domain model. An example shown in Figure 27 illustrates a domain specific pattern (4) is generated based on the generic safety pattern in Figure 27(1), domain model in Figure 27(2), and configuration table in Figure 27(3). In this example, role {s} defined in the generic safety pattern is replaced by {System}, role {s.h.allInstances()} is replaced by {System.Hazard.allInstances()}, role {h} is replaced by {Hazard}, role {h.sr.allInstances()} is replaced by {Hazard.SafetyRequirement.allInstances()}, and role {h.r} is replaced by {Hazard.Report} in the domain specific pattern shown in Figure 27(4).

5.4.2 Pattern Combination

To increase the flexibility of the generation of an assurance case from safety patterns, SPIRIT provides the pattern combination feature which allows a user to generate a domain specific pattern via the combination of the generic safety patterns. When two safety patterns are combined together, SPIRIT employs the information from a pattern connection to find how nodes in first safety pattern and second safety pattern are connected to each other. SPIRIT supports two types of pattern connections based on the type of a node involved in the connection to build a complete safety pattern.

The first type, which does not alter an argumentation structure of any safety pattern and has been widely used by current pattern-based approaches, combines two patterns via the duplicate nodes between the two patterns. This type of connection requires the identical nodes generated in the first pattern and a starting node of the second pattern. Figure 28 illustrates the generation of a combined safety pattern shown in Figure 28(2) via the connection of duplicate nodes between two patterns shown Figure 28(1). In this example, the combined safety pattern shown in Figure 28(2) is generated by connecting pattern 1 to itself via a link between node G2 and G1 in pattern 1 (shown in Figure 28(1)). In a domain specific pattern, all variables are replaced with a domain class. Note a domain specific pattern can be generated by the same safety pattern multiple times as shown in Figure 28(1). A variable can be replaced with different domain class when generating the domain specific pattern. Specifically, in Figure 28(2), variable {s} is replaced with domain class *Hazard* in the first application of pattern 1 while variable {s} is replaced with domain class *SWContribution* in the second application of pattern 1. When a combined safety pattern is generated via a connection of duplicate nodes, SPIRIT checks the identity between the two duplicate nodes involved in the connection after replacing all variables with the domain classes in the first and second pattern respectively. Two nodes are identity if the variables are mapped to same domain class or replaced by the same string. For example, {h} in G2 in the first application of pattern 1 is mapped to domain class *SWContribution*, and {$c} is

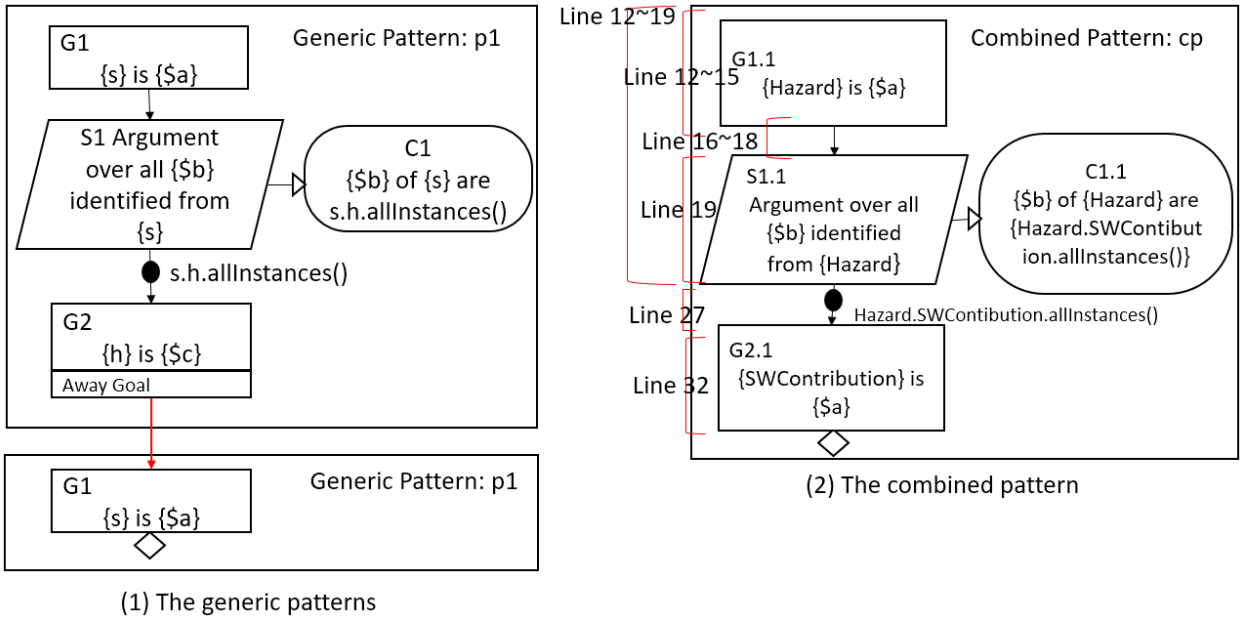


Figure 28 Pattern combination via duplicated nodes

replaced by string “addressed”. Variable {s} in G1 in the second application of pattern 1 is also mapped to domain class *SWContribution*, and {$a} is also replaced by “addressed”. In this case, these two nodes are identical. In the combined safety pattern, a new link is set between the parent node of the duplicate node in the first pattern and the starting node of the second pattern; and the duplicate node in the first pattern is discarded in the combined safety pattern. Therefore, in Figure 28(2), node G2 in first pattern is discarded in the combined safety pattern, and the starting node G1 in the second pattern in Figure 28(1) is added as a new node G2.1 to the combined safety pattern in Figure 28(2). A new link is set between S1.1 and G2.1 in the complete safety pattern in Figure 28(2) and the link between S1 and G2 in the first pattern is removed.

The second type of connection which alters an argumentation structure of a first safety pattern combines two patterns via the connection of a node in the first pattern to a starting node of the second pattern. In this case, a node of the first pattern is different from a starting node of the second pattern during combination. For example, Figure 29 shows how two safety patterns are combined to generate a domain specific safety pattern via the second type of connection. In this example, the combined safety pattern is generated via the connection between node S1 in the first pattern and starting node G3 of the second pattern (shown in Figure 29(1)), and a link between node S1.1 and G3.1 is added in the combined pattern (shown in Figure 29(2)). In SPIRIT, the type of pattern connections is an input provided by a client. If a client chooses an

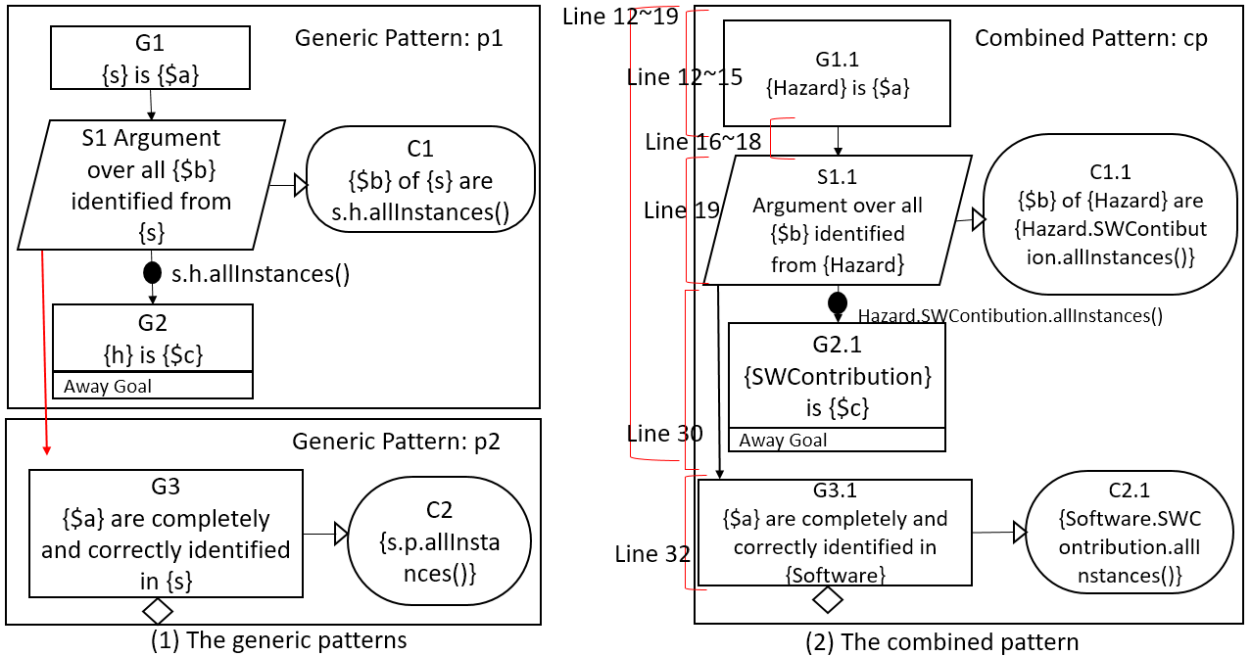


Figure 29 Pattern combination via node connection

away goal node in the first pattern, then the first type of connections is chosen. Otherwise, the second type of connection is carried out by our framework.

We outline the algorithm to generate a complete domain specific safety pattern in Figure 30. The algorithm takes a configuration table as input given by the *c* parameter and all pattern connections, given by the *PC* parameter. This algorithm generates an ATL program which is ready to produce an assurance based on a domain specific pattern such as the one in Figure 29(2). Note that a configuration table includes a safety pattern to be instantiated. The algorithm starts from generating an empty pattern denoted by variable *newp* at line 3. Next, it calls method *addNode* at line 5. Generally speaking, the *addNode* method copies a generic safety pattern whose root node is given by parameter n to a domain specific pattern denoted by parameter *newp* (from line 12 to line 19). At the same time, when a node is copied from a generic safety pattern to a domain specific pattern, the algorithm replaces each variable with a corresponding domain class (from line 12 to line 15). After a generic safety pattern is copied, the *addNode* method checks whether the current safety pattern is linked to some other safety patterns via the list of pattern connections given by parameter *PC* (lines 20 and 21). In particular, method *checkPatternConnection* mainly finds whether such a list of target safety patterns exist or not. If yes, the *checkPatternConnection* method returns the list. If a list is returned, the *addNode* method iterates each safety pattern connection (from line 22 to line 32). For each safety pattern connection, our framework identifies which type of pattern connections based on whether an away goal node is involved or not (on line 24). If an away goal node is involved in a connection, the combination of two patterns is set up via duplicate nodes (from line 25 to line 28) as the first type of pattern connections. In a duplicate node connection, our framework first checks if the two nodes involved in a connection are identical (on line 25) and get the parent node of the duplicate node in the first pattern and the link between the parent node and this duplicate node (on line 26). In a pattern connection via duplicate nodes, the *addNode* method creates a link from the parent node of the duplicate node in the first pattern to the root node in the second pattern. And the duplicate node will be removed from the domain specific pattern (from line 27 to line 28). If the node in the first pattern involved in the pattern connection is not an away goal node, the second type of pattern connection is applied to the pattern combination. In a second type of connection, the *addNode* method creates a link from a node in a source pattern to the root node in a target pattern and next establishes an appropriate link relationship based on the information

from a pattern connection (from line 29 to line 30). After that, the *addNode* method add the link to the domain specific pattern (on line 31) and calls itself to copy the second safety pattern to the domain specific pattern (on line 32).

```
Create the combined pattern
Input: The first configuration table used in the project, list of all Pattern Connection
PC
Output: An project specific combined assurance pattern newp

1    createCombinedPattern(c, PC)
2    {
3        Pattern newp = blank
4        identify the pattern c.p defined in c, and the root node n of c.p
5        addNode(newp, c.p, n, c, PC)
6    }
7    // newp a domain specific pattern to be generated
8    // p: current safety pattern; n: the root node of p;
9    // c: current configuration table; PC: a list of all pattern connections;
10   addNode(newp, p, n, c, PC)
11   {
12       for each variable v in n
13           if v is instantiable
14               replace v to dc via c
15       add node n to newp
16       for each link l in p   // for links in the same pattern p
17           if n = l.sn
18              add l to newp
19              addNode(newp, p, l.tn, c, PC)
20        List<PatternConnection> pcs = checkPatternConnection(p, PC, c, n)
21        if (pcs != null)   // node n has link(s) to other pattern(s)
22           for each PatternConnection pc in pcs
23                identify the root node rn of pc.tp.p
24                if type of n is away goal // first type of combination
25                   if n is identical to rn
26                      find the parent node pn of n and link pl between pn and n
27                         link nl = new link(pn, rn, pc.m)  // create a new link for pn to rn
28                         remove n and pl from newp // remove the duplicate node
29                else  // second type of combination
30                   link nl = new link(n, rn, pc.m)  // create a new link for n to rn
31                add nl to newp
32                addNode(newp, pc.tp,  rn, pc.tc, PC)
33   }
34   List<PatternConnection> checkPatternConnection(p, PC, c, n)
35   {
36   List<PatternConnection> pcs = blank // the matched pattern connections
37   for each Pattern Connection pc in PC
38   if ( p = pc.sp.p AND n = pc.n AND c = pc.sc)
39              remove pc from PC // update the pattern connections
40              add pc to pcs
41   return pcs
42   }
```

Figure 30 Pattern combination algorithm

To further illustrate how the algorithm works, we take two safety patterns shown in Figure 28 and Figure 29 into account and show how two patterns are combined together via the first and second type of pattern connection. Assume the algorithm takes configuration tables (Figure 31(1)) and pattern connection table (Figure 31(2)) as input. First, the new pattern *cp* in (Figure 29 (2)) is generated from line 3 in Figure 30. The pattern used in this example is p1, therefore, the root node of p1 is identified from line 4 in Figure 30 as G1. For node G1, variable {s} is replaced by domain class {Hazard} from line 14 in Figure 30. Node G1.1 is added to *cp* from line 15 in Figure 30. Node G1 links S1 in p1. Therefore, this link is added to *cp* from line 18 in Figure 30 and method *addNode* is called from line 19 in Figure 30 which adds S1.1 to *cp*. Line 20 in Figure 30 checks if G1 is links to another pattern via method *checkPatternConnection* from line 34 to line 42 in Figure 30, since there is no link between G1 to any other pattern, method *checkPatternConnection* returns an empty list, and the *addNode* method called on node G1 is finished.

Similarly, our algorithm adds the rest of nodes, S1and C1, in p1 to *cp* (S1.1 and C1.1) via the recursion of method add Node. Since node S1 links to p2 as defined in the Pattern Connection Figure 31(2), a second type pattern connection is identified by our algorithm when the *addNode* method is called for adding node S1 to *cp*, the *checkPatternConnection* returns a *PatternConnection* in line 41 of Figure 30. The root node G3 of p2 is identified from line 23 in

Configuration Table: c1 (Pattern1)	
Role	**WBS Domain class**
s	Hazard
h	SWContribution

Configuration Table: c2 (Pattern2)	
Role	**WBS Domain class**
s	Software
p	SWContribution

Configuration Table: c3 (Pattern1)	
Role	**WBS Domain class**
s	SWContribution
h	SafetyRequirement

(1) Configuration tables

Source Pattern	Source Node	Source Pattern Configuration Table	Target Pattern	Target Pattern Configuration Table	Multiplicity
p1	S1	c1	p2	c2	1
p1	G2	c1	p1	c3	s.h.allInstances()

(2) Pattern connection

1. LI = {“is”, “Argument over all”, “identified from”, “of”, “are”, “are completely and correctly identified in”}
2. V = { s, $a, $b, h, s.h.allInstances(), p, s.p.allInstances()}
3. G1 = < {“is”}, {s, $a} >, S1 = <{“Argument over all”, “identified from”}, {$b, s}>, G2 = < {{h, $c}}>, C1 = <{“of”,”are”},{$b,s,s.h.allInstances()}>, G3 =<{“are completely and correctly identified”},{s,$a}>, C2 = <{},{s.p.allInstances()}>. VN = {N1.1, N1.2, N1.3, N1.4, N2.1, N2.2}
4. N = {}
5. l1 = <G1, S1, 1>, l2 = <S1, G2, s.h.allInstances()>, l3 = <S1, C1, 1>, l4 = <G3, C2, 1> , L = { l1, l2, l3, l4 }
6. vn1 = {G1, S1, G2, C1}, vn2 = {G3, C2}, p1 = <vn1, {l1, l2, l3}>, p2 = <vn2, {l4}> P = { p1, p2 }
7. DC = {Hazard, SWContribution, Software, SafetyRequirement}
8. mp1 = <s, Hazard>, mp2 = <h, SWContribution>, mp3 = <s, Software>, mp4 = <p, SWContribution>, mp5 = <s, SWContribution>, mp6 = <h, SafetyRequirement>, MP = {mp1,mp2,mp3,mp4,mp5,mp6}
9. c1 = <p1, {mp1, mp2}>, c2 = <p2, {mp3, mp4}>, c3 = <p1, {mp5, mp6}>, C = {c1, c2, c3}
10. pc1 = <c1, c2, S1, 1>, pc2 = <c1, c3, G2, s.h.allInstances()>, PC = {pc1, pc2}

(3) Data types

Figure 31 Pattern combination inputs

Figure 30, and a new link between S1.1 to G3.1 is added to *cp* from line 31 in Figure 30 which connects node S1 in p1 to node G3 in p2. Node G3.1 is added to *cp* from line 32 in Figure 30. Our algorithm stops when all nodes in p1 and p2 are added to the combined pattern cp.

A first type pattern connection is identified when method *addNode* is called for node G2 in the first pattern (Figure 28(1)), node G2 is temporary added to the combined safety pattern *cp* from line 15 in Figure 30. Line 20 in Figure 30 checks if G2 is links to another pattern via method *checkPatternConnection* from line 34 to line 42 in Figure 30. Since node G2 links to p1 is defined in the Pattern Connection Figure 31(2), the *checkPatternConnection* returns a *PatternConnection* in line 41 of Figure 30. The root node G1 of p1 is identified from line 23 of Figure 30. Line 24 identifies the type of G2 is away goal node. Therefore, Line 25 in Figure 30 checks G2 in the first application is identical to G1 in the second application after replacing all variables in both nodes. The parent node of G2, S1, is identified from Line 26 in Figure 30, and a new link between S1.1 and G2.1 is generated from line 27 in Figure 30. After the new link is generated, the duplicate away goal node G2 is removed from the combined safety pattern *cp* from line 28 in Figure 30. The new link between S1.1 and G2.1 is added to *cp* from line 31 in Figure 30, and node G2.1 is added to *cp* from line 32 in Figure 30.

5.4.3 ATL Program Generation

To model an assurance case in GSN, the GSN working group develop the GSN metamodel conforms with GSN standard [37]. The GSN metamodel is an extension of the Structured Assurance Case Metamodel (SACM) [38]. Therefore, the generation of an assurance case can be done by a transformation from the instance model of a domain model (specific system) to an instance model of the GSN metamodel (assurance case). Atlas Transformation Language (ATL) is a model transformation language and toolkit which provide a mechanism to produce target models from a number of source models. In SPIRIT, we translate a domain specific safety pattern into an ATL program.

An ATL program is composed of a set of transformation rules and helpers. Each rule defines how the target model element is generated from the source model element. ATL defines two types of transformation rules: matched rule and called rule. An ATL matched generates target model elements based on the matched model element in the source model. An ATL matched rule is composed of two required section, *from* section and *to* section, and two optional section, *using* section and *do* section. The *from* section specifies the type of source model

element matched by a matched rule. The *to* section specifies the target model elements generated by a matched rule. The *using* section allows the declaration of local variables of a matched rule. A local variable declared in the *using* section can be used in the *using* section, *to* section, and *do* section within the same matched rule. The *do* section specifies the ATL imperative statements that is executed after the generation of the target model elements is complete in a matched rule. An ATL called rule is executed when the called rule is invoked from the action block within a matched rule or the body of a called rule. A called rule is composed of three optional sections, *to* section, *using* section, and *do* section. Compared with the ATL matched rule, an ATL called rule does not have *from* section and the *to* section is optional and an ATL called rule can accept parameters.

ATL allows a user to define methods in an ATL program, a method in ATL is called *helper*. ATL provides two different kinds of *helper*: the functional helpers and attribute helpers. An attribute helper is referred to as an attribute, and a functional helper is referred to as a helper. Similar to a method in an object-oriented language such as Java, a functional helper can accept parameters and has a return value. An attribute helper allows a user to define a global variable in an ATL program by specifying the name, type, and an initial value of a variable.

```
1  module sacm_gsn;
2  create OUT : SACM from IN : WBSModel;
3
4  helper def: varR1 : Integer = 1;
5
6  rule R1 {
7   from
8    s : WBSModel!"WBSModel::Software"
9   to
10   arg1: SACM!"SACM::Argumentation::Argumentation"(
11    id <- 'MyAssuranceCase'
12   ),
13   g1 : SACM!"SACM::Argumentation::Claim" (
14    id <- 'G1.'+ thisModule.varR1.toString()
15   )
16  do {
17    thisModule.called001(s, g1, arg1);
18  }
19 }
20
21 rule called001(p0 : WBSModel!Software, c : SACM!"SACM::Argumentation::ArgumentElement",
22                arg : SACM!"SACM::Argumentation::Argumentation") {
23  to
24   t : SACM!"SACM::Argumentation::ArgumentReasoning" (
25    id <- 'S1'
26   )
27 }
```

Figure 32 An ATL program example

An ATL program example is shown in Figure 32. Line 1 in this program specifies the name of this ATL module, *sacm_gsn*. Line 2 specifies the input metamodel is *WBSModel*, and the output metamodel is *SACM*. That is, an instance model of *WBSModel* is transformed to an instance model of *SACM*. An attribute helper is defined in line 4. In this example, an Integer type global variable called *varR1* is declared with initial value 1 in this ATL program. An ATL matched rule is defined from line 6 to line 19 in Figure 32. In this example, the name of this ATL matched rule is *R1*. Line 7 and line 8 specifies the from section of rule *R1*. In this example, the *from* section indicates the matched type of source model element is class Software of the *WBSModel* and this matched type is called *s*. That is, this matched rule is executed for every Software in the source model. Line 9 to line 15 specifies the *to* section of rule *R1*. In this example, two target model elements are generated when rule *R1* is executed. The first target element generated by this rule, *arg1*, is an instance of Argumentation class of SACM model specified shown in line 10, the id attribute of this Argumentation instance is assigned a value "*MyAssuranceCase*" in line 11. The second target element generated by rule *R1*, *g1*, is an instance of class Claim of SACM model in line 13. As mentioned in the previous section, GSN metamodel is an extension of SACM metamodel, the mapping relationships between GSN metamodel elements and SACM metamodel elements is shown in Table 1. A GSN assurance case is represented as an instance of Argumentation, and a GSN goal node is represented as an instance of Claim. Therefore, a GSN assurance case and a Goal node are generated in line 10 and line 11 when this matched rule is executed. Line 14 set the value of the id attribute of this claim instance based on the value of the global variable varR1. In this example, the value of id is set to "*G1.1*". Line 16 to line 18 specifies to do section of rule *R1*. The do section is executed after the execution of the *to* section of rule *R1* is complete. In this example, a called rule *called001* is invoked from the do section of *R1* in line 17, and three arguments are passed to the called rule, which are the software instance matched by *R1*, and the two target model elements generated by *R1*. An ATL called rule is define from line 21 to line 27 in this ATL program. Line 21 and line 22 specifies the name and the parameters of this called rule. In this example, the name of this called rule is *called001*. Three parameters are defined in this called rule: the first parameter *p0* whose type is Software in *WBSModel*, the second parameter *c* whose type is *ArgumentElement* in SACM model (Claim is a subclass of *ArgumentElement*), and the third parameter *arg* whose type is Argumentation. In this example, the called rule *called001* only contains a *to* section which

Table 1 Mapping relationships between GSN elements and SACM 1.0 elements

GSN Element	**SACE Element**	**Attributes of SACM Element**
GSN:Goal	SACM:Claim	assumed = false
GSN:Strategy	SACM:ArgumentReasoning	
GSN:Solution	SACM:InformationElement	
GSN:Context	SACM:InformationElement	
GSN:Assumption	SACM:Claim	assumed = true
GSN:Justification	SACM:Claim	assumed = false
GSN: X SupportedBy Y	SACM:AssertedEvidence	source = Y, target = X
GSN: X SupportedBy Y	SACM:AssertedInference	source = Y, target = X
GSN: X InContextOf Y	SACM:AssertedContext	source = Y, target = X

generates an instance of class *ArgumentReasoning* in the target model. Based on the mapping information shown in Table 1, a GSN strategy node is represented as an instance of *ArgumentReasoning*. Therefore, a GSN strategy node is generated in line 24 when this called rule is invoked. And the id attribute of this strategy node is set to "*S1*" in line 25. In summary, when a source model contains a software instance is loaded by this ATL program, a target model is generated with three elements: 1) an instance of Argumentation with id "*MyAssuranceCase*" to represent a GSN assurance case, 2) an instance of Claim with id "*G1.1*" to represent a Goal node in the generated assurance, and 3) an instance of *ArgumentReasoning* with id "*S1*" to represent a Strategy node in the generated assurance case.

As the mapping information shown in Table 1, different GSN elements may map to the same SACM element. These GSN elements are distinguished based on the attribute value and the relationships involved by these elements. For example, GSN Goal node, Assumption node, and Justification node are mapped to the same SACM class, *Claim*, as shown in Table 1. If the *assumed* attribute of an SACM *Claim* instance is set to true, then this *Claim* instance represents a GSN *Assumption* node. Otherwise, if this *Claim* instance is involved in an *AssertedInference*

relationship to support another node in an assurance case, then this *Claim* instance represents a GSN *Goal* node. Otherwise, if this *Claim* instance is involved in an *AssertedContext* relationship as a context of another node, then this *Claim* instance represents a GSN *Justification* node. In GSN, a *Goal* node can be supported by *Goal* node(s) or supported by *Solution* node(s). If a *Goal* node A is supported by another *Goal* node B, an instance of *AssertedInference* is used to represent the *SupportedBy* relationship where the *source* attribute of this *AssertedInference* instance is set to B and the *target* attribute of this *AssertedInference* instance is set to A. If a GSN *Goal* node C is supported by a GSN *Solution* node D, an instance of *AssertedEvidence* is used to represent the *SupportedBy* relationship where the *source* attribute of this *AssertedEvidence* instance is set to D and the *target* attribute of this *AssertedEvidence* instance is set to C. When a GSN *Goal* node is supported by another *Goal* node via a *Strategy* node, an *AssertedInference* instance is used to represent the *SupportedBy* relationship where the *source* is the child *Goal* node and the *target* is the parent *Goal* node, and the *describedInference* attribute of this *Strategy* node is set to this *SupportedBy* relationship. For example, a GSN *Goal* node A is supported by GSN *Goal* node B via a GSN *Strategy* node S. An instance of *AssertedInference, r*, represents the *SupportedBy* relationship where *r.source* = B and *r.target* = A and *S.describedInference = r*.

In SPIRIT, we propose an approach to automatically generate an ATL program from a domain specific pattern. The basic ideal of the ATL program generation is illustrated as below:

1. An ATL matched rule is generated as the entry point of an ATL program to generated root goal node of an assurance case in GSN.
2. Three called rules are generated in the ATL program to create the GSN *SupportedBy* and *InContextOf* relations between the GSN nodes in an assurance case.
3. The algorithm visits all nodes defined in the domain specific pattern from the root node to the leaf nodes. An ATL called rule is generated for each visited node (besides the root node), and the parameter of a called rule is generated based on the role expressions between a parent node and its child nodes(s).
4. The links between any two nodes in a domain specific pattern are generated by invoking the two called rules specified in 2.
5. Role(s) and Literal(s) defined in a node in the domain specific pattern is translated as assignment statements in the corresponding ATL rules.

An algorithm shown in Figure 33 illustrates the top-level pseudo codes for ATL generation from a domain specific safety pattern. First, the root node of a pattern is identified in line 3 and then the root node is visited in line 4. When the root node is visited, and ATL matched rule is generated from line 9 where the name of this rule is based on the *ID* of the visited node. Our approach requires the root node of a domain specific pattern must have at least one variable role so a matched type of source model element can be specified in the matched rule. A goal node may contain more that on variables, in this case, our algorithm finds the appropriate source model class via the *findMatchedSourceElement* method in line 10. After the source model class is identified for the root node, the *from* section of this matched rule is generated and the matched source model class is set in the *from* section in line 11. The *to* section of a matched rule is created in line 13. Two target model elements are generated in this matched rule: the first element generated from this rule is an instance of SACM *Argumentation* to represent a GSN assurance case via the *addElement* method in line 14, and all elements within an assurance case are owned by an *Argumentation* instance. The second element generated a GSN node based on the type of "node" via the *addElement* method in line 15. In this example, an instance of SACM *Claim* is generated to represent a GSN goal node in the generated assurance case. Next, the *do* section of this matched rule is generation in 16. In SPIRIT, three types of statements are added in the *do* section of an ATL rule: 1) Set the value of an attribute of a node, 2) Add the statements of a node, the statements including variables and literals in a node, and 3) Add a statement to invoke an ATL called rule. Line 17 set the value of *ID* attribute of the generated GSN *Goal* node, and Line 18 set the *assumed* attribute to false of the *Goal* node. For each role and literal defined in the root node, line 19 converts the element to an ATL statement in the *do* section. From line 21 to line 26, three ATL called rule are generated for the three SACM relationship elements: *AssertedInference, AssertedEvidence,* and *AssertedContext*. These ATL called rules are invoked when a relation defined in a safety pattern is identified. Line 27 identifies all outgoing relationships of the root node, that is, all relationships between the root node and its child nodes. For each outgoing relation, the child node of the root node is visited via the *VisitNode* method in line 29, and line 30 add an expression in the *do* section to invoke the ALT called rule generated by the *VisitNode* method.

Line 34 to line 68 illustrates the *VisitNode* method of our algorithm. The *VisitNode* method is called when a node defined in a safety pattern is visited. The main purpose of this

VisitNode method is to generate an ATL called rule for a node identified from the safety pattern. An ATL called rule is generated in line 36 where the name of this rule is based on the *ID* of the

```
1 main()
2 {
3     root = findRoot(pattern)        // find the root node of the domain specific pattern
4     VisitRoot(root)                 // visit the root node of the pattern
5 }
6
7 VisitRoot (node)                    // visit the root node of a pattern
8 {
9     rule = createMatchedRule(node)  // create a matched rule for root node
10    sourceClass = findMatchedSourceElement(node)    // find the matched source model class
11    rule.createFromSection(sourceClass)             // create the "from" section and set the matched
12                                                    // source model element in the from section
13    to = rule.createToSection()                     // create the "to" section
14    to.addElement(Argumentation)                    // generate a new assurance case
15    to.addElement(node.type)                        // create a Goal node
16    do = rule.createDoSection()                     // create the "do" section
17    do.addStatement(setID(node))                    // set the ID of the Goal node
18    do.addStatement(setAssumed(node))               // set the assumed attribute to false
19    for each element "e" (role and literal) in node     // add ATL statement for every contents in "node"
20        do.addStatement(e)
21    createRelationRule(calledAssertedInferenceRelation) // create a called rule for
22                                                        // setting up the SupportedBy relation
23    createRelationRule(calledAssertedEvidenceRelation)  // create a called rule for setting up the SupportedBy
24                                                        // relation between Goal and Solution
25    createRelationRule(calledAssertedContextRelation)   // create a called rule for setting up the
26                                                        // InContextOf relation
27    for each outgoing relation r of node                // Visit all outgoing relations of "node"
28    {
29        VisitNode(r.outGoing)                           // visit each child node of "node"
30        do.addStatement(r)                              // add a invoke called rule statement
31    }
32 }
33
34 VisitNode (node)
35 {
36    rule = createCalledRule(node)       // create a called rule for "node"
37    roles = node.getRoles()             // get all roles defined in "node"
38    parameters = findParameters(roles)  // generate a list of parameters based on the roles defined in "node"
39    rule.setParameter(parameters)       // set the parameter list of this called rule
40    to = rule.createToSection()         // create the "to" section
41    to.addElement(node.type)            // generate a GSN node based on the type of "node"
42    do = rule.createDoSection()         // create the "do" section
43    do.addStatement(setID(node))        // set the ID of "node"
44    do.addStatement(setAssumed(node))   // set the assumed attribute based on the type of "node"
45    get relation r between node and node.parent             // find the relation between "node" and its parent node
46    if (r instanceOf InContextOf)                           // if "node" is involved in an InContextOf relation
47        calledAssertedContextRelation(node, node.parent)    // set a InContextOf relation between "node"
48                                                            // and the parent node of "node"
49    else                                                    // node is involved in an SupprotedBy relation
50    {
51        if (node instanceOf Solution)                           // if node is a solution node
52            calledAssertedEvidenceRelation(node, node.paretn)   // set a SupportedBy relation between "node"
53                                                                // and the parent node of "node"
54        else if (node.parent instanceOf Strategy)
55            calledAssertedInference(node, node.parent.parent)   // set a SupportedBy relation between "node"
56                                                                // and the parent node of the parent node of "node"
57        else
58            calledAssertedInference(node, node.parent)          // set a SupportedBy relation
59                                                                // between "node" and the parent node of "node"
60    }
61    for each element "e" (roles and literals) in node           // add ATL statement for every contents in "node"
62        do.addStatement (e)
63    for each outgoing relation r of node                    // Visit all outgoing relations of "node"
64    {
65        VisitNode(r.outGoing)                               // visit each child node of "node"
66        do.addStatement(r)                                  // add a invoke called rule statement
67    }
68 }
```

Figure 33 Top level pseudo code for ATL generation

visited node. As described previously, an ATL called rule can have parameters. In a generated ATL program, a target model element can be referenced in different ATL called rule via parameters. The parameters of an ATL called rule are identified in line 38 based on the roles defined in a safety node. The identified parameters are set to this ATL called rule in line 39. An ATL called rue does not have the *from* section, the *to* section of an ATL called rule is generated in line 40, and an SACM element is generated based on the type of the visited node in line 41. Line 42 generates the *do* section of this rule. Similar to *VisitRoot* method, the *ID* and *assumed* attributes of the generated SACM element are set in line 43 and line 44. From line 46 to line 60, the algorithm identifies the relationship between this visited node and its parent node in the pattern in order to build a correspond link in a generated assurance case. If the relationship between this visited node and its parent node is an *InContextOf* relation, an SACM *AssertedContext* instance is generated via method *calledAssertedContextRelation* in line 47. The *source* and *target* attributes of this *AssertedContext* instance are set via the two arguments, *node* and *node.parent*, of this method call in line 47. A *SupportedBy* in GSN can map to SACM *AssertedInference* or *AssertedEvidence* as shown in Table 1. Line 52 in Figure 33 generates an instance of *AssertedEvidence* via *calledAssertedEvidenceRelation* method if the visited node is a GSN *Solution* node. Otherwise, two different situations need to be considered separately if the visited node is a GSN *Goal* node: 1) If this visited *Goal* node supports its parent *Goal* node via a *Strategy* node, an instance of *AssertedInference* is generated where the *source* attribute is this visited *Goal* node *node*, and the *target* attribute is the parent *Goal* node of *node* retrieved via *node.parent.parent* in line 55, where *node.parent* returns the *Strategy* node links to *node* and the parent node of this *Strategy* node is the parent *Goal* node of *node.* 2) If this visited *Goal* node is directly support the parent *Goal* node, and instance of *AssertedInference* is generated where the *source* attribute is this visited *Goal* node *node*, and the *target* attribute is the parent *Goal* node of *node* retrieved via *node.parent* in line 58. For each role and literal defined in the root node, line 62 converts each role and literal to an ATL statement in the *do* section. For each outgoing relation, the child node of this visited node is visited via the *VisitNode* method in line 65, and line 66 add an expression in the *do* section to invoke the ALT called rule generated by the *VisitNode* method. This program stops when all nodes defined in a domain specific pattern are visited.

The description of the methods used in the top-level algorithm in Figure 33 is shown in Figure 34. Method *findMatchedSourceElement* is illustrated from fine 1 to line 13 in Figure 34. This method is called from line 10 in Figure 33 to find the appropriate domain model class as the source model element specified in the *from* section of an ATL matched rule. The root *Goal* node in a pattern must contain at least one variable role to represent a specific system artifact where a claim is being evaluated. But a root *Goal* node may contain more than one variable roles. In that case, the *findMatchedSourceElement* finds the appropriate variable role and return the domain class mapped by this role as the source model element specified in the *from* section of an ATL matched rule. Line 4 creates an empty list to store all domain classes mapped by the roles defined in the root node of a domain specific pattern. Recall that each variable role in a domain specific pattern is replaced by the name of a domain class, line 5 finds the domain class for each role in this node and add the domain class to the class list. An appropriate domain class is defined as a class that has associations to all other domain classes in the class list. Line 11 finds this appropriate domain class and returns this class to the caller of this method. Method *addElement* is illustrated from line 15 to line 19 in Figure 34. This method is called from line 15 and line 41 in Figure 33 to generate an assurance case element defined in the *to* section of an ATL rule. The target model element generated by an ATL rule is represented as syntax " varName : varType" where *varName* specifies the variable name of a generated element and the varType specifies the type of the generated target model element. An example shown in Figure 32 line 10 illustrates a variable called *arg1* whose type is *Argumentation* in *SACM* model is generated in rule *R1*. In our algorithm, we use a variable to generate a unique variable name for each target model element generated in an ATL rule, and the type of the generated element is provided by the parameter *type*. An ATL statement is generated in Figure 34 line 18 in the *to* section of an ATL rule. Method *createdRelationRule* is illustrated from line 21 to line 33 in Figure 34. This method is called from line 21, 23, and 25 in Figure 33 to generate an ATL called rule for specifying a relationship between two assurance case nodes. Recall that the GSN *SupportedBy* and *InContextOf* are represented by three SACM classes, *AssertedInference*, *AssertedEvidence*, and *AssertedContext*. The purpose of this method is used to generate the called rules in an ATL program. A relationship between two nodes in an assurance case is generated by invoking the called rule from another ATL rule. This method takes two parameters, the name of the called rule and the type of the target model element generated by this rule. Line

```
1 findMatchedSourceElement(node)
2 {
3     create a domain class list "clist" = empty  // create a list store domain
4                                                 // classes mapped by roles in "node"
5     for each roles "r" in node
6     {
7         c = findMappedClass(r)                  // find the domain model class "c"
8                                                 // mapped by "r" via configuration table
9         clist.add(c)                            // add "c" to the "clist"
10    }
11    find class "m" in clist where "m" has associations with all other classes in clist
12    return m                                    // return the matched class
13 }
14
15 addElement(type)
16 {
17    generate ATL statement:
18        var : SACM!"type"   //create an ATL statement to generate a target model element
19 }
20
21 createRelationRule(ruleName, type)
22 {
23    rule = createCalledRule(ruleName)           // create a called rule
24    rule.setParameter(s)                        // set parameter "s" for "rule"
25    rule.setParameter(t)                        // set parameter "t" for "rule"
26    to = rule.createToSection()                 // create the "to" section
27    to.addElement(type)                         // generate a new target model element
28    do = rule.createDoSection()                 // create "do" section
29    d = description(source, s)                  // create a new description which
30    do.addStatement(d)                          // assigns value "s" to attribute "source"
31    d = description(target, t)                  // create a new description which
32    do.addStatement(t)
33 }
34
```

Figure 34 Pseudo code of methods description 1

23 in Figure 34 create an ATL called rule and set the rule name via the *ruleName* parameter. The relationship between two assurance case nodes is set via the *source* and *target* attributes of the *AssertedInference*, *AssertedEvidence*, and *AssertedContext* classes as shown in Table 1. Line 24 and line 25 in Figure 34 specifies the two parameters of this called rule. The *to* section of this called rule is generated from line 26, and one target model element is generated in this called rule in line 27. The type of this generated target model element is provided from parameter *type* of the *CreateRelationRule* method in line 21. The *do* section of this called rule is generated in line 28. Two descriptions are generated in line 29 and line 31 to set the value of *source* attribute and *target* attribute. And these two descriptions are added to the *do* section of this called rule in line 30 and line 32.

Table 2 ATL statements for different types of roles

Type	Role/Literal	Mapped Class	ATL Statements
R1	{a}	a -> X	content <- content + X.ID;
R2	{a.b}	a -> X b -> Y	content <- content + X.y.ID;

R3	{a.allInstances()}	a -> X	for(e in DomainModel!X.allInstances()) {
			IDs <- IDs + e.ID+' ';
			}
			content <- content + IDs;

R4	{a.b.allInstances()}	a -> X
		b -> Y
			for(e in X.y)) {
			IDs <- IDs + e.ID+' ';
			}
			content <- content + IDs;

L	Literal “L”		content <- content + L;

When a role defined in a node within the safety pattern is identified by our algorithm, an ATL statement is generated from the *addStatement* method shown in Figure 35 based on the type of a role expression as discussed in section 4.4.1. The role expression and its corresponding ATL statements is summarized in Table 2. Line 38 in Figure 35 identifies the type of a role. Based on the type of a role, this method creates different ATL statements within the *do* section of an ATL rule from line 44 to line 86. The mapped domain model class is identified and stored to variable *X* via the *findMappedClass* method in line 46 and line 51. If a role expression is a derived role (e.g. *a.b* or *a.b.allInstances()*), two domain classes are identified and stored to variable *X* and *Y* in line 52, 53 and line 71, 72. The association *a* between *X* and *Y* is retrieved from the *findAssociation* method in line 54 and line 73. Since two domain classes may have more than one association, SPIRIT allows a user to provide which association is selected by the algorithm by specifying the association role name of the second mapped class. Otherwise, SPIRIT will prompt a user to select an association during the generation of an ATL program. Line 55 and line 74 finds the role name *r* of association *a* at the end of the second mapped domain class of a

derived role, and the ATL statements are generated from in line 48 for a role whose type is *R1*, line 57 for a role whose type is *R2*, line 63 to line 67 for a role whose type is *R3*, line 76 to line 80 for a role whose type is *R4*, and line 85 for role whose type is *L*.

```
35 addStatement(role)
36 {
37     content = ""
38     type = role.getType // identify the type of a role. type R1: a, type R2: a.b,
39                         // type R3: a.allInstances(), type R4: a.b.allInstances()
40                         // type L: literal
41                         // find the domain classes "X", "Y" mapped by
42                         // role "a", "b" via configuration table
43
44     if(type = R1)       //generate an ATL binding statement for "role"
45     {
46         X = findMappedClass(role.first)
47         generate ATL statement:
48             content <- content +  X.id;
49     }
50     if(type = R2)
51     {
52         X = findMappedClass(role.first)
53         Y = findMappedClass(role.second)
54         a = findAssociation(X, Y)
55         y = a.getroleName(Y)
56         generate ATL statement:
57             content <- content + X.y.id;
58     }
59     if(type = R3)
60     {
61         X = findMappedClass(role)
62         generate ATL statements:
63             for(e in DomainModel!X.allInstances())
64             {
65                 IDs <- IDs + e.ID + ',';
66             }
67             content <- content + IDs;
68     }
69     if(type = R4)
70     {
71         X = findMappedClass(role.first)
72         Y = findMappedClass(role.second)
73         a = findAssociation(X, Y)
74         y = a.getroleName(Y)
75         generate ATL statements:
76             for(e in X.y)
77             {
78                 IDs <- IDs + e.ID + ',';
79             }
80             content <- content + IDs;
81     }
82     if(type = L)
83     {
84         generate ATL statements:            //generate an ATL binding statement for "literal"
85             content <- content + role;
86     }
87     generate ATL statement:
88         var.Content <- content              //set the Content of the target element
89 }
```

Figure 35 Pseudo code of methods description 2

Figure 36 illustrates the *addStatement* method for setting the value of an attribute of an assurance case element and adding a statement to invoke an ATL called rule from another rule. Line 91 to line 97 illustrates the *addStatement* method for setting the value of an attribute. This method takes a parameter *description* which composed of the name of an attribute and the value being set. Line 93 gets the attribute name from *description* and stored the information to variable *attributeName*, and line 94 gets the attribute value from *description* and store the information to variable *attributeValue*. An ATL statement is generated in the *do* section to assign the *attributeValue* to *attributeName* in line 96. Line 99 to line 120 illustrates the *addStatement* method for adding a statement to invoke an ATL called rule from another rule. Recall that our algorithm generates ATL called rules for every node defined in a safety pattern (besides the root node), but the assurance case elements are generated only when a called rule is invoked. This method is called when a relation is identified from a safety pattern during the visiting of a node. A parameter *relation* is required for this method. Line 101 gets the outgoing node of a relation and stored the information to a variable *n*. Line 102 finds the ATL called rule, *r*, for node *n*, which is the called rule being invoked. The arguments are identified based on the parameters of *r*, and the multiplicity of *m* of *relation* is identified in line 104. If the multiplicity of a relation is

```
 91 addStatement(description)
 92 {
 93     attributeName = description.getAttribute    // get the attribute name
 94     attributeValue = description.getValue       // get the attribute value
 95     generate ATL statement                      // generate an ATL statement to
 96         var.attributeName <- attributeValue;    // assign value to attribute
 97 }
 98
 99 addStatement(relation)
100 {
101     n = relation.outGoing                                  // get the outgoing node "n" of "relation"
102     find the corresponding ATL called rule "r" for "n"
103     generate "arguments" based on the parameters of "r"
104     m = relation.multiplicity                              // get the multiplicity of "relation"
105     if(m = 1)
106     {
107         generate ATL statement:                            // generate a simple rule binding statement
108             thisModule.r(argument);
109     }
110     else
111     {
112         X = findMappedClass(m.first)
113         Y = findMappedClass(m.second)
114         a = findAssociation(X, Y)
115         y = a.getroleName
116         generate ATL statements:                           // generate iterative rule binding statements
117             for(e in A.y)
118                 this.Module.r(argument);
119     }
120 }
```

Figure 36 Pseudo code of methods description 3

1, an ATL statement is generated in line 108 to invoke the called rule *r* for one time. Otherwise, iterative ATL statements are generated from line 117 to line 118 to invoke the called rule multiple times based on the number of system artifacts.

5.4.4 ATL Model Transformation

Once an ATL program is generated from a domain specific safety pattern, an ATL transformation generates an assurance case for a specific system by executing the ATL program. Three input files are required for an ATL transformation: 1) the domain model represented in an Ecore model [39] as the Source metamodel, 2) the SACM metamodel represented in an Ecore model as the Target metamodel and 3) the system model represented in an XMI file, which is an instance of the domain model, as the Source model. The output of this ATL transformation is a Target model, which is an assurance case for a specific system. The generated assurance case is formatted as an XMI file, which is an instance of the SACM metamodel. The assurance case file is represented in a standard format which can be recognized by the commercial GSN editor tools such as Astah GSN [40]. By loading the assurance case file from the Astah GSN editor, a graphical representation of an assurance case can be easily generated to show an assurance case as a diagram as an example shown in Figure 46.

CHAPTER VI
ASSURANCE CASE MAINTENANCE AND EVALUATION

While the judgement of software assurance is a decision finally made by humans, some effort has been made to automatically identify the areas where a decision must be rendered. Specifically, various models have been proposed to calculate the confidence of the root node in an assurance case. While various calculations have been carried out, most existing approaches use a model to calculate the confidence of nodes affected by the evidence which is linked to system artifacts. They then set up a threshold to show and determine which nodes may be below the threshold. These nodes confidence is then in doubt. Therefore, the confidence of all nodes along the path to the root of an assurance is reduced and the software assurance about the system is in jeopardy. If the confidence of a node is not below the threshold, then the node's confidence is not reduced. This implies it should not affect the confidence of the other nodes in an assurance case. To this end, we take a most conservative model in this paper where once the framework detects a node whose linked artifact is modified, then the framework highlights all nodes along the path from this node to the root node of the assurance case. The rational behinds this idea comes from the following: To support the evolution of an assurance case, we propose that any miss of highlighting nodes whose confidence is in jeopardy is a costly mistake when evaluating software assurance of a system. Therefore, the system should highlight all potential nodes whose confidence could be undermined by modifications.

6.1 Assurance Case Maintenance

An assurance case can be generated via SPIRIT during the system development. Therefore, the system artifacts may be changed after an assurance case is generated. When a system artifact is changed, SPIRIT helps the user to identify the nodes in an assurance case that may be affected by the change of system artifact. In SPIRIT, the nodes potentially affected by the change are highlighted and report to the user. We identify the affected nodes in an assurance case based on three major aspects: 1) Identify the affected nodes based on the artifact references. If a node references to the changed artifact, this node is considered as an affected node. When an assurance case is generated via SPIRIT based on the safety pattern, all roles specified in the

safety pattern are instantiated to specific system artifacts. When a role in node is instantiated by a specific system artifact, we consider the system artifact is referenced by this node. SPIRIT stores the referenced system artifacts information in the generated assurance case. During the maintenance of an assurance case, SPIRIT identifies the nodes reference to specific system artifacts in the assurance case. 2) Propagation of affected nodes based on the assurance case structure. Once a node is identified as an affected node, the parent node of this affected node is also considered as an affected node.

Therefore, when a node is considered as an affected node, all ancestor nodes of this node are highlighted as affected nodes. 3) Identify the affected nodes based on traceability of system artifacts. When a system artifact is changed, all other artifacts derived from the changed artifact may be affected by the change. The traceability information is provided in the domain model. When a system artifact is changed, SPIRIT identifies the type, which is a domain class of the changed artifacts, and then find other classes that are derived from this class. Once the derived domain model classes are identified, SPIRIT find the instances of these domain model classes which are link to the changed system artifact, and all nodes in the assurance case that reference to these system artifacts are considered as affected nodes.

The algorithm for maintenance of an assurance case is outlined in Figure 37. The maintenance algorithm takes an assurance case generated by SPIRIT, all artifacts to be monitored, and a computation model as inputs to detect affected nodes in an assurance case. The algorithm uses another method *detect_modification* on line 1 which returns a list of modified artifacts. Next, the algorithm is enumerating all modified artifacts starting from line 2. For each modified artifact, the algorithm calls method *detect_affected_nodes* to find a list of nodes in an assurance which are affected by the modified artifacts. The algorithm then employs the

```
Input: an assurance case sc: AssuranceCases;
Input: an artifacts to be monitored allArtifacts: AllArtifacts;
Input: a model to calculate the affected nodes
        Model: Node X AssuranceCases -> P(AllArtifacts);
Output: a list of highlighted nodes highlighted_nodes: P(N)

1.  affected_elements = detect_modification(AllArtifacts);
2.  for each art in affected_elements do
3.      affected_nodes = detect_affected_nodes(art);
4.      for each node in affected_nodes do
5.          highlighted_nodes.add(Model(node, sc));
6.  return highlighted_nodes;
```

Figure 37 Assurance case maintenance algorithm

computation model provided as input to calculate a list of affected nodes. The returned list of the affected nodes is added to variable *highlighted_nodes* which is finally returned as output of the algorithm.

6.2 Assurance Case Evaluation

Software certification has become an important step for a software system in safety critical sectors due to the potential impact on society. Assurance cases have been increasingly considered by many emerging standards or governmental guidelines as an important argument structure linking the claims about software assurance properties to the evidence supporting these claims [41]. However, the evaluation of assurance cases demands the certifiers' domain knowledge and experience. Thus, most software certification has been conducted manually; and this hinders the development of a software system in safety critical sectors, delays the deployment of a new system to market, and finally stifle developers' creativity in safety critical sectors.

But the latest development in uncertainty theories and software traceability has given us aspiration to leverage the capability of software certification by means of some automation. Notice safety patterns have been employed to generate new assurance cases so many assurance cases have some similar structure. For a regular system certification, a certifier starts with the evaluation of each evidence node in an assurance case and then propagate the confidence information up towards the root claim of an assurance case. The goal of this feature is to monitor how a certifier chooses an acceptable assurance case and then infer an accurate model, where confidence can be calculated from leaf claims upwards into a root claim. Next, the accurate model can be automatically applied to a second assurance case to deduce its confidence. Many existing approaches employ some theories, then manually configure the values for variables in these theories, and finally calculate a confidence value of an assurance case [20, 21, 24, 42, 43, 44]. In SPIRIT, we employ the Dempster Shafer (D-S) theory as a main calculation model to illustrate some variables' values can be configured in automation.

To reduce the manual setting values for variables when applying the D-S theory, SPIRIT takes two assurance cases in GSN as input, i.e. an acceptable assurance case and an unknown assurance case. For an acceptable assurance case, SPIRIT generates a set of assurance cases, which is called a set of learning data, with the same structure except for the leaf claims which are

directly supported by an evidence node in GSN. When using the D-S theory, the values for a set of disjoint contributing weights, which are supposed to be manually provided when the confidence value of a root node can be deduced from its child nodes, can be inferred by means of making the first acceptable assurance case to be better than most assurance cases in the set of learning data. Last, the values for the set of disjoint contributing weights are applied to the second assurance to deduce its confidence. In this way, SPIRIT provides some automation to leverage certifiers' capability to perform the software certification.

SPIRIT supports the confidence evaluation of an assurance case whose generation is based on a safety pattern. The goal of the evaluation feature of SPIRIT is to aid a certifier to infer the confidence of a second assurance case, if the certifier carefully evaluates the first acceptable assurance case when both assurance cases are instantiated from a same safety pattern. Here, a careful evaluation of the acceptable assurance case means that a certifier compares the acceptable assurance case with some other assurance cases that have the similar structure. This comparison allows us to infer the calculation model which is used by a certifier and can be applied to the second assurance case.

To calculate the confidence of a second assurance case, SPIRIT requires the following inputs: for each argument: the type of the argument and discounting factor, which will be explained in the later section on the application of the D-S theory in assurance cases. Next, the framework carries out the following tasks:

- Convert an assurance case into a confidence calculation model where D-S theory can be applied.
- Use Vector Space Model to calculate similarity values as confidence for all leaf nodes in a confidence calculation model.
- Apply the D-S theory to infer all the disjoint contributing weights in the top structure, which is shared by the two assurance cases, using the fact that the first assurance case is acceptable.
- Apply all the disjoint contributing weights in the second assurance case to deduce its confidence.

Confidence evaluation of an assurance case starts with leaf nodes in its corresponding confidence calculation model. We notice that many assurance cases have the traceability information as leaf node. So, instead of asking for confidence as input for each leaf claim, which

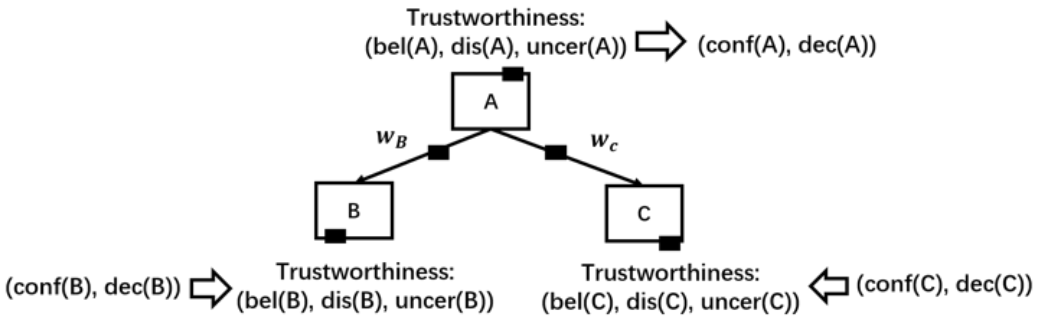


Figure 38 D-S theory for confidence calculation

is directly supported by an evidence node, we employ some similarity calculation models such as the Vector Space Model to calculation a similarity value for each leaf node. In SPIRIT, the evaluation feature mainly follows Wang et al.'s formulation for the D-S theory calculation [45]. When performing calculation propagation, the confidence of a top claim is based on the evaluation of the *trustworthiness* of each child claim and the *appropriateness* of all children claims. Thus, there are two assessment parameters corresponding to these two aspects. The first assessment parameter denotes the trustworthiness of a claim, say *P*, denoted as a 3-tuple (*bel*(*P*), *dis*(*P*), *uncer*(*P*)) showing belief, disbelief, and uncertainty of claim P, respectively. The second assessment parameter represents the appropriateness of a claim, *i.e.*, the degree that the claim can independently contribute to the trustworthiness of its parent claim. The degree is called a *disjoint contributing weight* denoted as *w*. A simple assurance case with the trustworthiness and appropriateness information is shown in Figure 38 where w_B and w_C, that are called disjoint contributing weights of B and C respectively, denote the appropriateness of B and C to support A. Thus, a disjoint contributing weight of a claim is also called an appropriateness value of the claim.

SPIRIT abstracts an assurance case into a calculation model which includes the main claim nodes and evidence nodes in the assurance case. In a calculation model, a claim node, denoted as *P*, is measured by three tuples, i.e. belief, disbelief, and uncertainty, and denoted as *bel(P)*, *bel*($\bar{P}$), and *uncer(P)* respectively. The 3-tuple format is used when applying the D-S theory. A frame of discernment Ω_P is $\{P, \bar{P}\}$, where $\bar{P}$ denotes false of P. The mass $m^{\Omega_P}(P)$ shows the degree of belief committed to the hypothesis that truth lies in P [45]. The 3-tuple is thus defined as follows:

$$\begin{cases} bel(P) = m^{\Omega_P}(P) = g_P \\ dis(\bar{P}) = m^{\Omega_P}(\bar{P}) = f_P \\ uncer(P) = m^{\Omega_P}(\Omega_P) = 1 - m^{\Omega_P}(P) - m^{\Omega_P}(\bar{P}) = 1 - g_P - f_P \end{cases} \quad (1)$$

The confidence calculation of a claim depends on its sub-claims. Two types of sub-claim contributions can be used to support a parent claim, *i.e*., a dependent argument and redundant argument. A dependent argument is the contribution of one child claim to its parent claim depends on another (*i.e.*, sibling) child claim. For example, in an argument, claim A: *"the system is acceptably safe*" is supported by claim B: *"the system is passed by verification"* and claim C: *"the system is passed by testing"*. In this case, the type of this argument is redundant since, being two different techniques, either B and C can support A in some degree without being dependent on the other one. A dependent argument denotes the contribution of a child claim to support its parent claim has some degree of overlap with another child claim. For instance, in another argument, claim A: *"The system is acceptably safe"* is supported by claim B *"the test process is sound"* and claim C *"the test results are correct"*. In this argument, the type is dependent since the test results given by C to support claim A depends on the test process given by claim B. Since it is not possible to infer that the trustworthiness of a claim can be derived from all its children claims via the appropriateness, *i.e*., the completeness of all children claims mentioned earlier, we use the *discounting factor* v to denote the uncertainty about the appropriateness of all children claims to support their parent claim. For instance, for the latter argument, claims B and C cannot have a large value of *v* since a system being acceptably safe cannot be highly ensured when only the testing approach is adopted.

Next, we consider how the trustworthiness of a parent claim can be deduced by the trustworthiness of its all sub-claims in a calculation model by using the formula from [46]. The trustworthiness of a claim denoted as A with an n-claims dependent argument is given as follows:

$$\begin{cases} bel(A) = v[\left(1 - \sum_{i=1}^{n} w_i\right)\prod_{i=1}^{n} g_i + \sum_{i=1}^{n} g_i w_i] = g_A \\ dis(\bar{A}) = v[\left(1 - \sum_{i=1}^{n} w_i\right)[1 - \prod_{i=1}^{n}(1 - f_i)] + \sum_{i=1}^{n} f_i w_i] = f_A \\ uncer(A) = 1 - g_A - f_A \end{cases} \quad (2)$$

where each w_i (i=1, 2, …, n) denotes a disjoint contributing weight of the i_{th} sub-claim to support its parent claim. Again, due to space constraints, the formula for the trustworthiness of a

claim with an n-claims redundant argument is omitted and can be found in [45]. The research goal of this paper is to find the appropriate values of w_i (i=1, 2, …, n) when the total appropriateness value of all children claims in an argument is given, *i.e.*, $\sum_{i=1}^{n} w_i = c$ where *c* is a preset value. Similarly, the trustworthiness of a claim denoted as A with an n-claims redundant argument is given as below:

$$\begin{cases} bel(A) = v[\left(1 - \sum_{i=1}^{n} w_i\right)[1 - \prod_{i=1}^{n}(1 - g_i)] + \sum_{i=1}^{n} g_i w_i] = g_A \\ dis(\bar{A}) = v[\left(1 - \sum_{i=1}^{n} w_i\right)\prod_{i=1}^{n} f_i) + \sum_{i=1}^{n} f_i w_i] = f_A \\ uncer(A) = 1 - g_A - f_A \end{cases} \quad (3)$$

Another format of trustworthiness of a claim represents a certifier's evaluation on claim *P* using a 2-tuple (*dec(P), conf(P)*) where *dec* denotes a certifier's confidence value and *conf* represents the confidence about the method used by a certifier to have a value for *dec*. We thus call a *dec* value of a claim a trustworthiness value of the claim in that a *dec* value directly denotes a confidence value given by a certifier. Since the 2-tuple and 3-tuple formats reflect two different perspectives on confidence of a claim, some conversion is necessary. For instance, a certifier's evaluation on a leaf claim via a 2-tuple is converted to a 3-tuple so the D-S theory can be carried out. Likewise, a 3-tuple of a root claim is converted to a 2-tuple after the calculation, mimicking a certifier's evaluation. Formulas (4) and (5) show the conversions between a 3-tuple and a 2-tuple of a claim. Following the common practice, we accept an assurance case or not based on a trustworthiness value of a root in the assurance case and we use 0.7 as a threshold value.

$$\begin{cases} bel(P) = conf(P) \times dec(P) \\ dis(P) = conf(P) \times (1 - dec(P)) \\ uncer(P) = 1 - bel(P) - disb(P) \end{cases} \quad (4)$$

$$\begin{cases} conf(P) = bel(P) + disb(P) \\ dec(P) = bel(P) \,/\, (bel(P) + disb(P)), \ if\ bel(P) + disb(P) \neq 0 \\ dec(P) = 0, \quad if\ bel(P) + disb(P) = 0 \end{cases} \quad (5)$$

The derivation of all disjoint contributing weights for the top structure which is shared by the two assurance cases is the key step performed by SPIRIT. Since the first assurance case is acceptable, a certifier should compare the first assurance case with some other assurance cases which have the same structure except for leave sub-goals. For instance, assume leaf goal node

contains a claim statement "*artifact a1 traces to artifact b1 at the System"*, a certifier considers the claim statement as "*artifact a1 traces to artifact b2 at the System*". In this case, the certifier uses the evidence node to figure out a new confidence value for the leaf goal node. Based on this fact, the framework generates a certain number of assurance cases to infer the disjoint contributing weights for the top structure. In the coupled tanks control system, we generate 300 assurance cases using the following procedure.

To change the claim from "*artifact a1 traces to artifact b1 at the System*" to "*artifact a1 traces to artifact b2 at the System*" for the goal node, SPIRIT first generates a similarity table for each leaf claim, sorts the similarity value in a descending order, and finally chooses the best, average, and worst similarity values as a candidate set to generate a new assurance case. The reason to do so is to make sure some so-called best similarity values can be chosen but the disjoint contributing weights can still be correctly derived. Next, for each leaf goal node, SPIRIT randomly chooses a value from the candidate set and an assurance case is generated once all leaf goal nodes are considered.

Besides values of disjoint contributing weights, SPIRIT calculates the trustworthiness for all sub-goals before formula (2) is used to calculate the trustworthiness of the parent goal node. Likewise, the framework calculates the trustworthiness for each sub-goal. As such, the trustworthiness calculation of our framework uses the bottom up strategy, starting with leaf goal nodes. But since the two assurance cases have different structures in the bottom portions, respectively, the framework assigns equal weights to all disjoint contributing weights for an argument. Finally, the confidence of a leaf node is converted by formula (4) using the similarity value as a *dec* value and 0.8 as a *conf* value. Next, SPIRIT finds a set of values for all disjoint contributing weights to assign the first assurance case with the highest *dec* value among the set of assurance cases used as learning data. Finally, the framework applies the values to the second assurance case and calculates a 3-tuple for the root claim and then converts the three-tuple back to a 2-tuple element using formula (5), thereby automatically generating the confidence of the second assurance case that would otherwise be performed manually by a certifier.

6.2.1 Evaluation Process

The goal of the SPIRIT assurance case evaluation process is to automatically deduce values for the assessment parameters such as the trustworthiness and appropriateness, i.e., disjoint contributing weights, of claims from the first assurance case, which has been determined

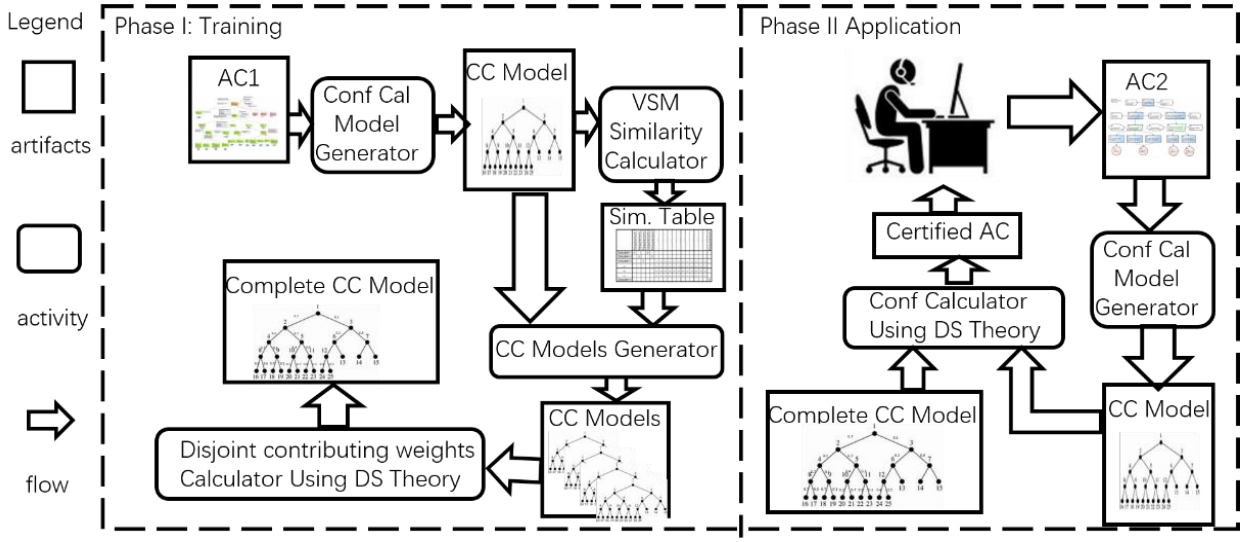


Figure 39 Overview of SPIRIT assurance case evaluation process

to be acceptable. Since the second assurance case is derived based on the safety pattern where the first assurance case is derived, the derived disjoint contributing weights are applied to the second assurance case to finally deduce its confidence. To do so, the process has two phases shown in Figure 39:

- The Training Phase. First, the framework converts an assurance case into a confidence calculation model where D-S theory can be applied. Second, the framework uses Vector Space Model to calculate similarity values as confidence for all leaf claims in a confidence calculation model. Third, the framework generates a set of similar assurance cases as learning data using the similarity values. Last, the framework applies the D-S theory to infer all the disjoint contributing weights in a common structure of two input assurance cases, using the fact that the first assurance case is acceptable.
- The Application Phase. The framework initially converts a second assurance case into a confidence calculation model. Next, the framework employs the disjoint contributing weights to the confidence calculation model so the D-S theory is applied to deduce the confidence of the second assurance case. Fourth, apply all the disjoint contributing weights in the second assurance case to deduce its confidence.

6.3 Implementation of Assurance Case Evaluation

In this section, we introduce the technical details for implementing the assurance case evaluation feature.

6.3.1 Confidence Calculation Model and Vector Space Model

Since an assurance case includes auxiliary information about how an argument structure is formed (e.g., a strategy node, justification node, and context node in an assurance case), we remove this information, such that only claims and evidence nodes are included in a confidence calculation model. Furthermore, if a claim called c_1 is supported by only one child claim c_2 that is further supported by child claim c_3, then we have claim c_1 is directly supported by child claim c_3 without including child claim c_2 in a confidence calculation model. In this case, an appropriateness value of a child claim is equal to the value given in a safety pattern.

Confidence evaluation of an assurance case starts with leaf claims, i.e., evidence nodes, in its corresponding confidence calculation model. We notice that many assurance cases have traceability information at the leaf claim level. Therefore, instead of asking for a confidence value as input, we use the Generalized Vector Space Model (GVSM) [47] to calculate a similarity value to mimic how a certifier evaluates a leaf claim.

6.3.2 Generation of a Set of Learning Data

Since the first assurance case is acceptable, a certifier usually compares it with some other assurance cases which have the same structure except for leave claims. Based on this fact, the framework generates a certain number of assurance cases to infer the disjoint contributing weights for the same structure of two GSNs.

SPIRIT first generates a similarity table for each leaf claim in the first input assurance case using GVSM [47] Next, SPIRIT sorts the similarity value in a descending order, and finally chooses the best, ideal, average, and worst similarity values as a candidate set to generate a new assurance case. The reason to do so is to make sure no extreme learning data can be generated. A candidate set is stored as a table attached to each leaf claim and used to generate new assurance cases.

The *createTrainingData* method shows how a training data set is created as shown in Figure 40. The method initializes the output variable list TD by adding the acceptable assurance case ac as the list first element at line 1. For each of the two case studies in this paper, we generate 300 assurance cases for the training data set. Next, the *creatingTrainingData* method enters a loop

statement whose execution generates one assurance case and add it to the TD list from line 2 to line 12. For each generation of an assurance case, the *creatingTrainingData* method gets a random number, i.e., variable x, at line 3 such that x leaf claims in ac are modified to produce a new assurance case. To do so, at line 4, the method initializes the list *CN*, that stores a list of leaf claims to be modified in a new assurance case, to be empty. Starting at line 5, the method enters another loop to collect x leaf claims to be modified in a new assurance case. Specifically, at lines 6 and 7, a leaf claim should be randomly chosen from all leaf claims of ac such that the chosen leaf claim should not be duplicate to any current existing leaf claim in *CN*. At line 8, the method copies ac to a new variable ac' that stores a new generated assurance case. From line 9, the method iterates each leaf claim in *CN* so that the leaf claim is modified in ac'. At line 10, each modification is carried out using the similarity table attached to the leaf claim of ac. After the loop statement of line 9, the method generates a new assurance case ac' that has x different leaf claims from ac. Next, the method updates *TD* by adding ac' at line 11 and replaces ac with ac' at line 12 since the next round of generation is based on the currently generated assurance case. At the end of the loop, a training data set is created.

6.3.3 Derivation of Disjoint Contributing Weights

Once types of arguments as well as total appropriateness values of child claims together are set based on a structure of a safety pattern, SPIRIT starts the confidence calculation with leaf

procedure CreateTrainingData
input:1. *ac:* an assurance case used to generate training data and;
2. *s:* the size of training data to be generated
output: TD=$\{ac, ac_1, ac_2, \dots, ac_{s-1}\}$

1 initialize TD = $\{ac\}$;
2 for i =1 to s-1
3 initialize x = randomly select a value between 1 and *ac.leafs.size*
4 initialize CN =∅
5 for i = 1 to x
6 CN =CN ∪$\{ac.leaf_i\}$ where $ac.leaf_i is\ ramdonly\ chose\ from$
7 $ac.leafs\ and\ \ ac.leaf_x \neq ac.leaf_y,\ \ x, y \in \{1,2, \dots, i\}$
8 initialize ac'= ac;
9 for each $ac.leaf_i$ in CN
10 $ac'.leaf_i =$ ramdonly select one value from $ac.leaf_i.SimilarityTable$
11 TD = TD ∪ $\{ac'\}$
12 *ac* = ac'

Figure 40 Create training data algorithm

claims via the D-S theory algorithm. Next, SPIRIT employs the following schema for each argument. For instance, consider a goal node has three sub-goals in an assurance case, three disjoint contributing weights are defined as w_1, w_2, and w_3 respectively. To find the best configuration of these disjoint contributing weights, the framework uses a preset step value, which is 0.1, and then employs the following formula (6) to find the value for w_1, w_2, and w_3.

$$\sum_{i=1}^{3} w_i = 0.9, where\ w_i \in \{0.1, 0.2, \dots, 0.7\} \quad (6)$$

The framework iterates each configuration and assurance case in a training data set to ensure that the acceptable assurance case is ranked as top as possible. To do so, method *IdentifyConfiguration* as shown in Figure 41 takes the accepted assurance case as parameter ac, and the training data as parameter *trainingData*. The method starts with calling method *find_all_confs(ac)* to find all possible configuration according to the structure of ac at line 1. Then, the method initializes the variables including the outputs from line 2 to line 4. From line 5, the method finds a best configuration by entering a loop statement. For each loop, i.e., each

```
Procedure IdentifyConfiguration(trainingData, ac)
Input: trainingData: a set of training data
       ac: the accepted assurance case
Output: wc: a winning set of disjoint contributing weights
        best_rank: the best rank of ac in trainingData
        trustworthiness: a trustworthiness value of ac in wc
1   Initialize wcs = find_all_confs(ac);
2   initialize best_rank = trainingData.size();
3   initialize wc = ∅
4   initialize best_dec = 0;
5   for each weight configuration c in wcs
6       initialize declist = ∅
7       for each assurance case ac in trainingData
8           initialize trustw = Calculate_Using_DS(c,ac)
9           declist.add()
10      new_rank = findRankofFirstAC(declist)
11      if(new_rank < best_rank  OR (new_rank == best_rank &&
                declist.get(0).getDec() > trustworthiness.getDec()))
12          best_rank = new_rank
13          wc = c
14          trustworthiness = declist.get(0)
```

Figure 41 Identify disjoint weights algorithm

configuration c, the method retrieves each assurance case ac from *trainingData*, calls method *Calculate_Using_DS (..)* to calculate a trustworthiness value of ac under configuration c, i.e., a 2-tuple element, and adds the value to list *declist* from line 7 to line 9. Next, the method gets the rank of the accepted assurance case among *trainingData* via calling method *findRankofFirstAC (..)* at line 10. If the new rank is better than the previous best one, i.e., variable *best_rank*, or the new rank is equal to the previous best one but the current configuration has a large *dec* value than the previous best one's, i.e. the trustworthiness variable, then the method updates the outputs so the current configuration, rank and trustworthiness are assigned to the corresponding output variables. Note that the confidence calculation of our framework uses the bottom up strategy, starting with leaf claims. So disjoint contributing weights for all arguments can be derived and the framework finds a set of values for all disjoint contributing weights to assign the first assurance case with the highest *dec* value among the set of assurance cases used as learning data. There exist scenarios where sections of two assurance cases have different structures even though they are both derived from the same safety pattern. In this case, we skip the calculation of disjoint contributing weights. Instead, the framework assigns equal weights to all disjoint contributing weights for all children claims instead of considering all sets of disjoint contributing weights satisfying formula (6).

CHAPTER VII
CASE STUDY

7.1 Coupled Tanks Control System

In this dissertation, we applied SPIRIT to an aircraft fuel tank system called coupled tanks control system [48] to illustrate the generation, maintenance, and evaluation of an assurance case in support mission critical software development. The coupled tanks control system is proposed by the Verification and Validation of Complex Systems (VVCS) group within AFRL/RQQA [48]. The coupled tanks control system is a cooling tanks system stores liquid for some process and then releases the liquid into a bottomless sink. The system contains coupled tanks, each of which is controlled by independent controllers that operate four actuators and receive feedback from five sensors. The development process of the coupled tanks control system consists of three major activities. The requirement analysis activity uses the Specification and Analysis of Requirements (SpeAR) framework to prove that a given model meets its formal specification. The architecture analysis activity uses the Assume Guarantee Reasoning Environment (AGREE) to evaluate the behavior of the subsystem components with the greater system. The model analysis activity uses Simulink Design Verifier (SLDV) to prove that the guarantees (and original requirements) hold true throughout the modeling phase. In this case study, we build an assurance case mainly focuses on the requirements elicitation and the SpeAR model properties elicitation of the coupled tanks control system. The coupled tanks control system considers two types of requirements: 1) the concept of operations (CONOPS) requirements illustrates the high-level requirements target on the challenging problems of a cooling tank system [49] and 2) The system requirements define the system behaviors of the coupled tanks control system in different aspects based on the compositional architecture of the system [48]. The compositional architecture of the coupled tanks control system is shown in Figure 42. The root claim of the first assurance case claims that all system requirements are adequately elicited and documented in the corresponding document(s) based on the different

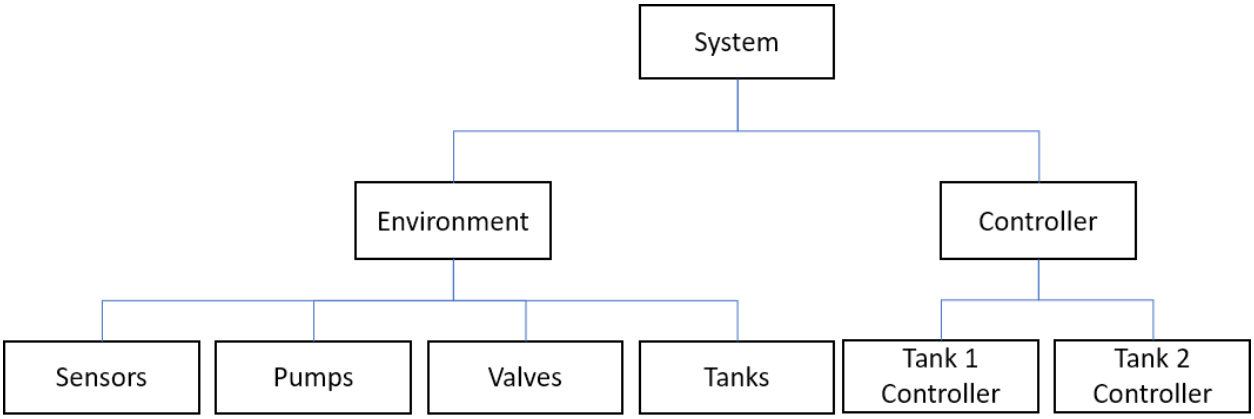


Figure 42 Coupled tanks control system compositional architecture

aspects of the coupled tanks control system architecture. For the system aspect requirements, the requirements should address the challenging problems targeted by the CONOPS requirements. Therefore, each CONOPS requirement is addressed by the coupled tanks control system requirement(s) at the system aspect, and each requirement at the system aspect is derived from the CONOPS requirement(s). The requirements at the system aspect are further refined to the requirements at the environment aspect and controller aspects. Therefore, each requirement at the system aspect addressed by the requirement(s) at environment aspect or controller aspect, and each requirement at environment aspect and controller aspect should derived from the requirement(s) at the system aspect.

To verify that the behaviors of the coupled tanks control meets the requirements, the coupled tanks control system SpeAR models are generated and the system requirements are formalized to the SpeAR model properties. An analysis can be performed to identify the problems of the SpeAR models including the Directly property conflict, Requirement conflict, and Under specification [50] based on the model checking techniques. The root claim of the second assurance case claims that all SpeAR model properties are adequately elicited and documented in the document(s) based on the different aspects of the coupled tanks control system architecture. For each aspect of the coupled tanks control system, a system requirement is formalized to the SpeAR model property(properties). And each SpeAR model property is derived from the system requirement(s).

7.2 Safety Pattern

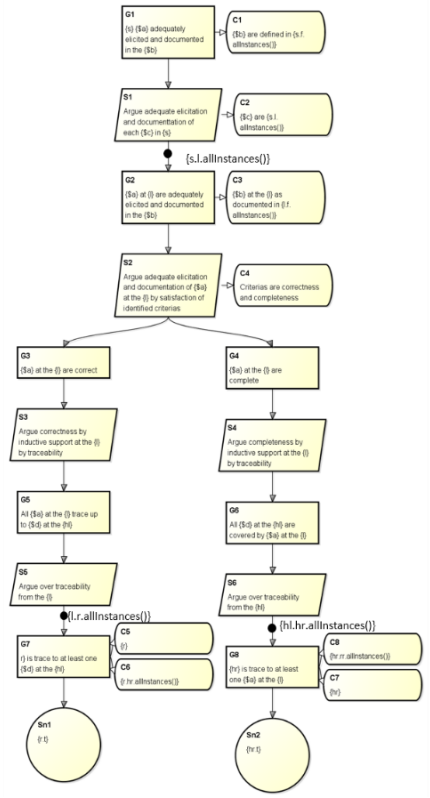
A safety pattern catalog shown in Figure 43 illustrates a safety pattern to support the argument structure of the claim of the adequate elicitation of the requirements and SpeAR model properties of the coupled tanks control system. The pattern catalog contains a catalog metamodel. The metamodel denotes the structure of a system (class *S*) is organized by its compositional architecture element (e.g., levels or aspects of a system) denoted by class *L*. A type of system artifact denoted by class *R* is associated with a specific architecture level or aspect denoted by class *L*. Class *HL* represents another architecture level or aspect in *S*, and class *HR* represents another type of system artifacts associated with *HR*. The document to store *R* and *HR* related to *L* and *HL* is represented by class *F*. A safety pattern illustrated in the pattern catalog defines seven variable roles and four string roles. Each variable role denotes a variable whose type which is the corresponding to the upper-case letter in the catalog metamodel. The four string roles are used to illustrate a domain class. In this example, *$a* illustrates a description of domain class mapped by *r*, *$b* illustrates a description of domain class mapped by *f*, *$c* illustrates a description of domain class mapped by *L*, and *$d* illustrates a description of domain class

Metamodel Description (for the CPS domain):

- The metamodel denotes the structure of a system (class S) is organized by its compositional architecture element (e.g, levels or aspects of a system) denoted by class L. A type of system artifact denoted by class R is associated with a specific architecture element L. Class HL represents another architecture element in S, and class HR represents another type of system artifacts associated with HR. The document to store R and HR related to L and HL is represented by class F.
- Each role denotes a variable whose type is the corresponding upper case letter in the metamodel.
- $a: a description of class R
  $b: a description of class F
  $c: a description of class L
  $d: a description of class HR

Safety Pattern Description:

- In this pattern, the top goal is that all instances of R (represented by {$a} of a cyber physical system S are adequately elicited and documented in F. To do so, the pattern considers the compositional structures L in S. For each instance of L, the claim of adequate elicitation of {$a} is decomposed by the claim of the completeness and correctness of R related to L. The claim of correctness is argued by each instance of R related to L is derived from a source, which is an instance of HR related to HL. The claim of completeness is argued by each instance of HR related to HL is covered by an instance of R related to L. In the end, the report to check the traceability support the claim of correctness and completeness of each L in S.



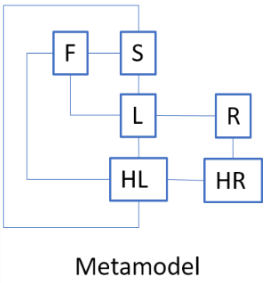


Figure 43 Safety pattern catalog for artifacts elicitation

mapped by *HR*. In this pattern, the top goal is that all instances of R (represented by {$a} of a cyber physical system S are adequately elicited and documented in F. To do so, the pattern considers the compositional structures *L* in *S*. For each instance of *L*, the claim of adequate elicitation of {$a} is decomposed by the claim of the completeness and correctness of *R* related to *L*. The claim of correctness is argued by each instance of *R* related to *L* is derived from a source, which is an instance of *HR* related to *HL*. The claim of completeness is argued by each instance of *HR* related to *HL* is covered by an instance of *R* related to *L*. In the end, the report to check the traceability support the claim of correctness and completeness of each *L* in *S*.

The mapping table shown in Table 3 illustrates the mapping relationships between roles defined in the safety pattern and the Cyber Physical System (CPS) domain model classes and the string values for generating the assurance case to argue all coupled tanks control system requirements are adequately elicited and documented. The mapping table shown in Table 4 illustrates the different mapping relationships which maps the same roles defined in Figure 43 to another sets of CPS domain classes for generating the assurance case to argue all coupled tanks control system SpeAR model properties are adequately elicited and documented. A partial CPS domain model is shown in Figure 44 to illustrate the relationships between the CPS domain classes mapped by the roles via the mapping relationship shown in Table 3. Two domain specific patterns are generated as shown in Figure 45. The generated domain specific patterns replace all roles defined in the safety pattern shown in Figure 43 to the CPS domain model classes based on the mapping relationships specifies in Table 3 and Table 4.

Table 3 Mapping table for requirement elicitation

Variable Role	**Domain Model Class**	**Association Role Name**
s	CyberPhysicalSystem	NULL
f	Documentation	documentation
l	Level	level
r	Requirement	requirement
hr	Requirement	sourceRequirement
hl	Level	upperLevel
t	TraceInfo	traceInfo
String Role	**Value**	
$a	requirements	
$b	requirement documents	
$c	aspects	
$d	requirements	

Table 4 Mapping table for SpeAR properties elicitation

Variable Role	**Domain Model Class**	**Association Role Name**
s	CyberPhysicalSystem	NULL
f	Documentation	documentation
l	Level	level
r	SystemProperty	systemProperty
hr	Requirement	sourceRequirement
hl	Level	level
t	TraceInfo	traceInfo
String Role	**Value**	
$a	SpeAR model properties	
$b	SpeAR model documents	
$c	aspects	
$d	requirements	

7.3 Assurance Case

Based on the generated domain specific patterns shown in Figure 45 and the CPS domain model shown in Figure 44, two ATL programs are generated via SPIRIT. After the ATL programs are generated, the ATL programs take the coupled tanks control system artifacts as shown in Table 5 and the CPS domain model shown in Figure 44 as inputs to generate the assurance cases for the coupled tanks control system. In this case study, we consider one instance of *CyberPhysicalSystem*, the *Coupled tanks control system*, four instances of *Level* are considered in this case study, which including the *CONOPS level* and three aspects of the Coupled tanks control system: *System aspect*, *Environment aspect*, and *Controller aspect*. Two instances of *Document* are considered in this case study, the *CONOPS document* stores the CONOPS requirements and the *Two Tanks Basic PA document* stores the system requirements and SpeAR model properties. Five CONOPS requirements are defined in the CONOPS document including *HR1*, *HR2*, *HR3*, *HR4*, and *HR5*. Eight system aspect requirements are defined in the coupled tanks control system including *SysR01 ~ SysR08*. Thirty-five environment aspect requirements are defined in the coupled tanks control system including *env_R01 ~ env_R35*, and eleven controller aspects requirements are defined in the coupled tanks control

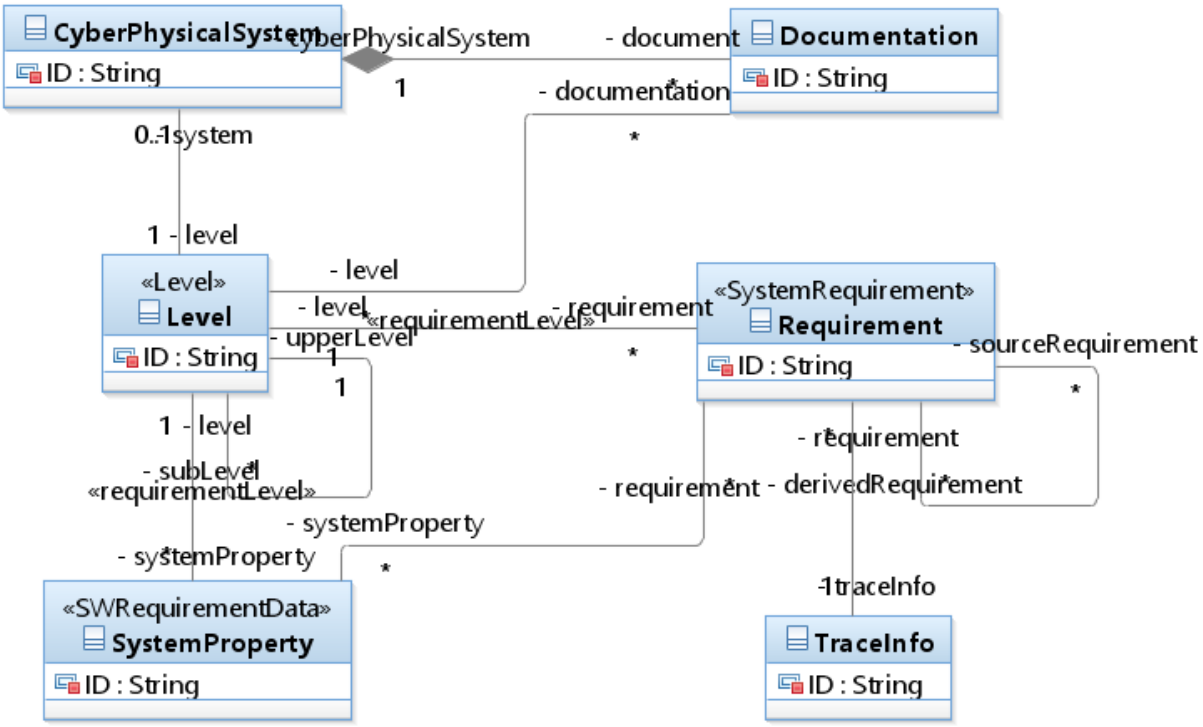


Figure 44 A partial CPS domain model

system including *Ctrl_R01 ~ Ctrl_R11*. Eight system aspect SpeAR model properties are defined in the coupled tanks control system including *p_sys_01 ~ p_sys_08*. Twenty-six system aspect SpeAR model properties are defined in the coupled tanks control system including *p_env_01 ~ p_env_10, p_pump_01, p_pump_02, p_pump_r01, p_pump_r02, p_sensor_01, p_sensor_02, p_t1_01, p_t1_02, p_t1_calculated_height, p_t2_01, p_t2_02, p_t2_03, p_t2_calculated_height,*

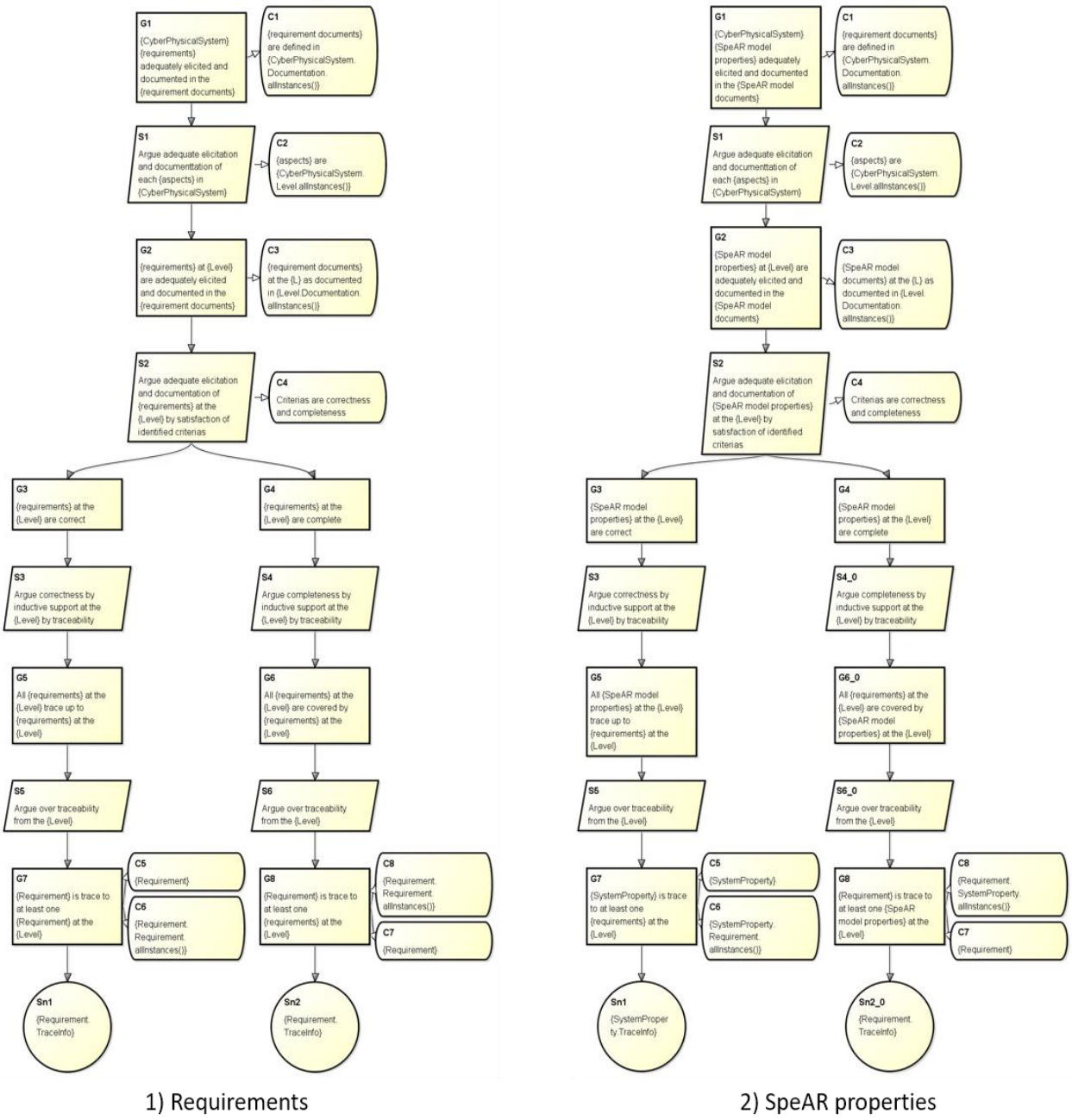


Figure 45 Domain specific pattern for requirement elicitation

Table 5 Coupled tanks control system artifacts

CyberPhysicalSystem	Level	Document	Requirement	SystemProperty
	CONOPS	Two Tanks CONOPS.pdf	HR1, HR2, HR3, HR4, HR5	
Two tanks control system	System aspect	Two Tanks Basic PA.docx	SysR01 ~ SysR08	p_sys_01 ~ p_sys_08
	Environment aspect	Two Tanks Basic PA.docx	env_R01 ~ env_R35	p_env_01 ~ p_env_10, p_pump_01, p_pump_02, p_pump_r01, p_pump_r02, p_sensor_01, p_sensor_02, p_t1_01, p_t1_02, p_t1_calculated_height, p_t2_01, p_t2_02, p_t2_03, p_t2_calculated_height, p_valve_01, p_valve_02, p_valve_r01, p_valve_r02
	Controller aspect	Two Tanks Basic PA.docx	Ctrl_R01 ~ Ctrl_R11	p_ctrl_01, p_ctrl_02, p_ctrl_03, p_ctrl_04 p_t1ctrl_p01, p_t1ctrl_r00, p_t1ctrl_r01, p_t1ctrl_r02, p_t2ctrl_01, p_t2_ctrl_02, p_t2ctrl_r00, p_t2ctrl_r01, p_t2ctrl_r02, p_t2ctrl_r03 p_t2ctrl_r04

p_valve_01, p_valve_02, p_valve_r01, and p_valve_r02. Fifteen controller aspect SpeAR model properties are defined in the coupled tanks control system including *p_ctrl_01, p_ctrl_02, p_ctrl_03, p_ctrl_04 p_t1ctrl_p01, p_t1ctrl_r00, p_t1ctrl_r01, p_t1ctrl_r02, p_t2ctrl_01, p_t2_ctrl_02, p_t2ctrl_r00, p_t2ctrl_r01, p_t2ctrl_r02, p_t2ctrl_r03, and p_t2ctrl_r04*.

The generated assurance cases for the requirements elicitation and SpeAR model properties elicitation are represented graphically using the Astah GSN editor. Due to the large number of the nodes in the assurance case, a simplified assurance diagram for the requirements elicitation is shown in Figure 46. In this assurance case diagram, we highlight some key nodes to illustrate the major claims of this assurance case. The root goal node of this assurance case claims that Coupled tanks control system requirements are adequately elicited and documented in the requirement document. The System aspect, Environment aspect and Controller aspect are considered in the coupled tanks control system as shown in Table 3, three goal nodes, *0G1.2.1*, *0G1.2.2*, and *0G1.2.3*, are generated in the assurance case, each of which claims the system requirements at a specific aspect are adequately elicited and documented. The claim of the adequate elicitation of each aspect in the coupled tanks control system is supported by the two

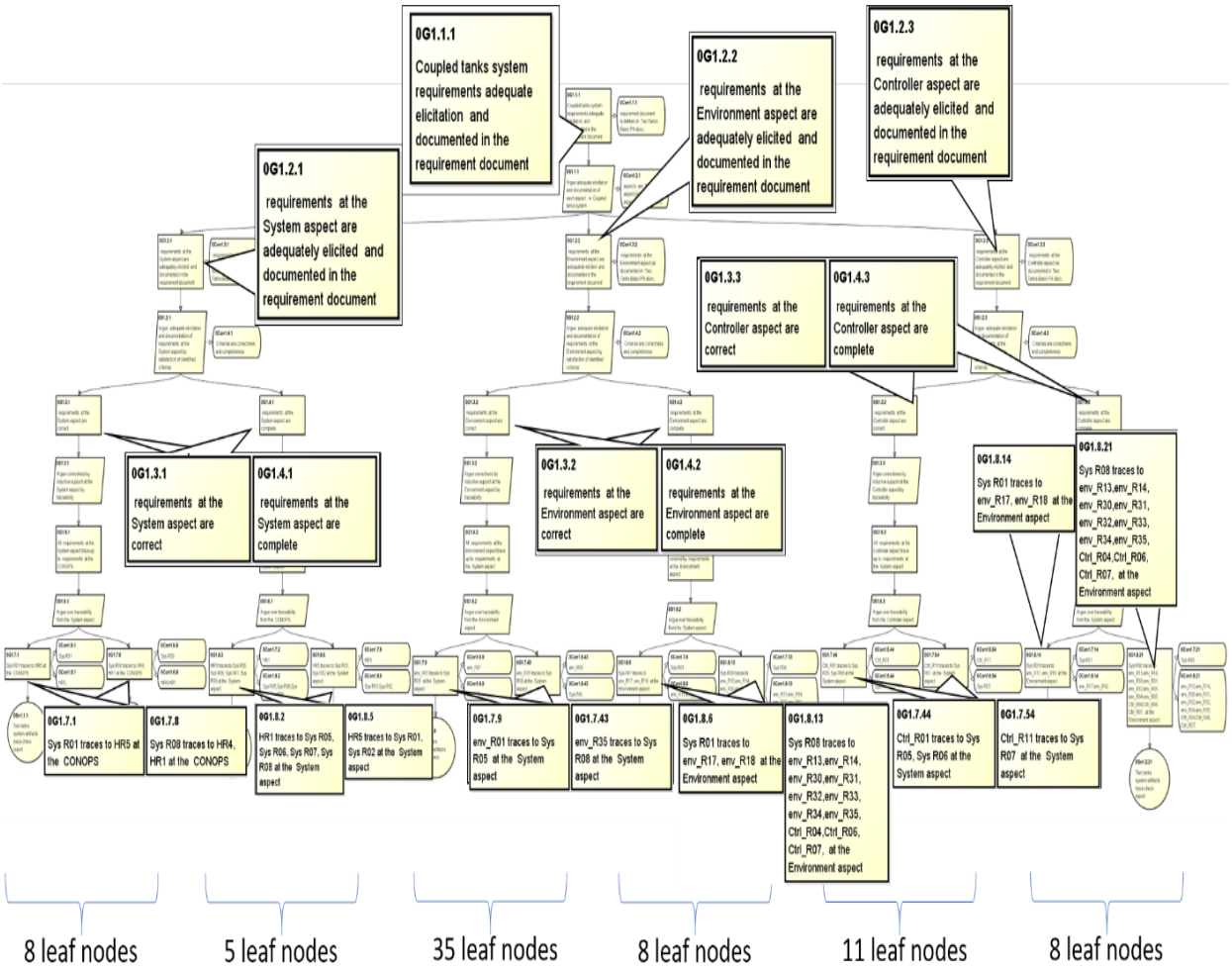


Figure 46 Coupled tanks requirements elicitation assurance case

identified criteria, completeness and correctness. Therefore, goal nodes *0G1.2.1*, *0G1.2.2*, and *0G1.2.3* are supported by two sub-goal nodes respectively, the first sub-goal node, *0G1.3.1*, *0G1.3.2*, and *0G1.3.3* claims the requirements at a specific aspect are correct, and the second sub-goal node, *0G1.4.1*, *0G1.4.2*, and *0G1.4.3* claims the requirements at a specific aspect are complete. The definition of correctness and completeness is derived from [51]. In this assurance case, the correctness is defined as all requirements are derived from a source. For the requirements at the system aspect, the claim of correctness is supported by the sub-claims that every requirement at system aspect traces to the CONOPS requirement(s) as goal nodes *0G1.7.1* to *0G1.7.8*. For the requirements at the environment aspect, the claim of correctness is supported by the sub-claims that every requirement at environment aspect traces to the requirement(s) at system aspect as goal nodes *0G1.7.9* to *0G1.7.43*. For the requirements at the controller aspect,

the claim of correctness is supported by the sub-claims that every requirement at controller aspect traces to the requirement(s) at system aspect as goal nodes *0G1.7.44* to *0G1.7.54*. The completeness is defined as all requirement are covered by the requirement(s) at different aspect. For the requirements at the system aspect, the claim of completeness is supported by the sub-claims that every CONOPS requirement traces to the requirement(s) at system aspect as goal nodes *0G1.8.1* to *0G1.8.5*. For the requirements at the environment and controller aspect, the claim of completeness is supported by the sub-claims that every requirement at system aspect traces to the requirement(s) at environment and controller aspects as goal nodes *0G1.8.6* to *0G1.8.21*. The coupled tanks system artifacts trace check report is considered as the evidence to support the traceability between these requirements are correct and complete.

The simplified assurance diagram for the second assurance for SpeAR model properties elicitation is shown in Figure 47. Similar to the assurance case diagram shown in Figure 46, we highlight the key goal nodes in the generated assurance case as shown in Figure 47. In the coupled tanks control system, the system requirements are formalized to the SpeAR model properties. The root goal node of this assurance case claims that Coupled tanks control system SpeAR model properties are adequately elicited and documented in the SpeAR model documents. The System aspect, Environment aspect and Controller aspect are considered in the coupled tanks control system as shown in Table 3, three goal nodes, *0G1.2.1*, *0G1.2.2*, and

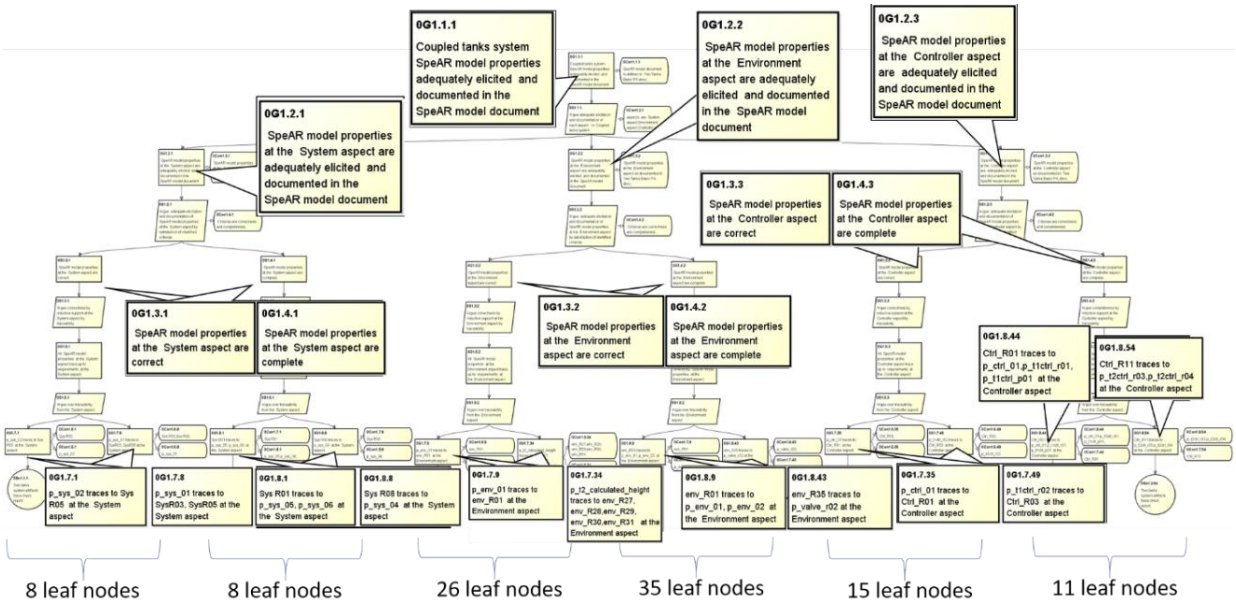


Figure 47 Coupled tanks SpeAR properties elicitation assurance case

0G1.2.3, are generated in the assurance case, each of which claims the SpeAR properties at a specific aspect are adequately elicited and documented. The same criteria, correctness and completeness are applied to this assurance case for the claim of the adequate elicitation of each aspect in the coupled tanks control system. Goal nodes *0G1.2.1*, *0G1.2.2*, and *0G1.2.3* are supported by two sub-goal nodes respectively, the first sub-goal node, *0G1.3.1*, *0G1.3.2*, and *0G1.3.3* claims the SpeAR properties at a specific aspect are correct, and the second sub-goal node, *0G1.4.1*, *0G1.4.2*, and *0G1.4.3* claims the SpeAR properties at a specific aspect are complete.

In this assurance case, the correctness is defined as all SpeAR properties are derived from coupled tanks control system requirements at the same aspect. For the SpeAR properties at the system aspect, the claim of correctness is supported by the sub-claims that every SpeAR property at system aspect traces to the requirement(s) at the system aspect as goal nodes *0G1.7.1* to *0G1.7.8*. For the SpeAR properties at the environment aspect, the claim of correctness is supported by the sub-claims that every SpeAR property at environment aspect traces to the requirement(s) at environment aspect as goal nodes *0G1.7.9* to *0G1.7.34*. For the SpeAR properties at the controller aspect, the claim of correctness is supported by the sub-claims that every SpeAR property at controller aspect traces to the requirement(s) at controller aspect as goal nodes *0G1.7.35* to *0G1.7.49*. The completeness is defined as all system requirements are covered by SpeAR properties at the same aspect. For the SpeAR properties at the system aspect, the claim of completeness is supported by the sub-claims that every system requirement at system aspect traces to the SpeAR properties at system aspect as goal nodes *0G1.8.1* to *0G1.8.8*. For the SpeAR properties at the environment aspect, the claim of completeness is supported by the sub-claims that every requirement at environment aspect traces to the SpeAR properties at environment aspect as goal nodes 0G1.8.9 to 0G1.8.43. For the SpeAR properties at the controller aspect, the claim of completeness is supported by the sub-claims that every requirement at controller aspect traces to the SpeAR properties at controller aspect as goal nodes *0G1.8.44 to 0G1.8.54*.

7.4 Maintenance

In this case study, we consider a coupled tanks control system requirement (ID) is changed and analyze how the nodes in assurance cases are affected by the change. To

demonstrate the case study example, two partial assurance cases for the coupled tanks control system requirements elicitation and SpeAR properties elicitation are shown in Figure 48. When coupled tanks system requirement *Sys R01* is changed, SPIRIT first identifies the assurance case nodes reference to *Sys R01*. In this example, node *0G1.7.1* are identified as affected nodes based on the change of (Sys R0x), these affected nodes are marked with ① in Figure 48(i). Secondly, SPIRIT identifies the affected nodes based on the structure of the assurance case. The ancestor

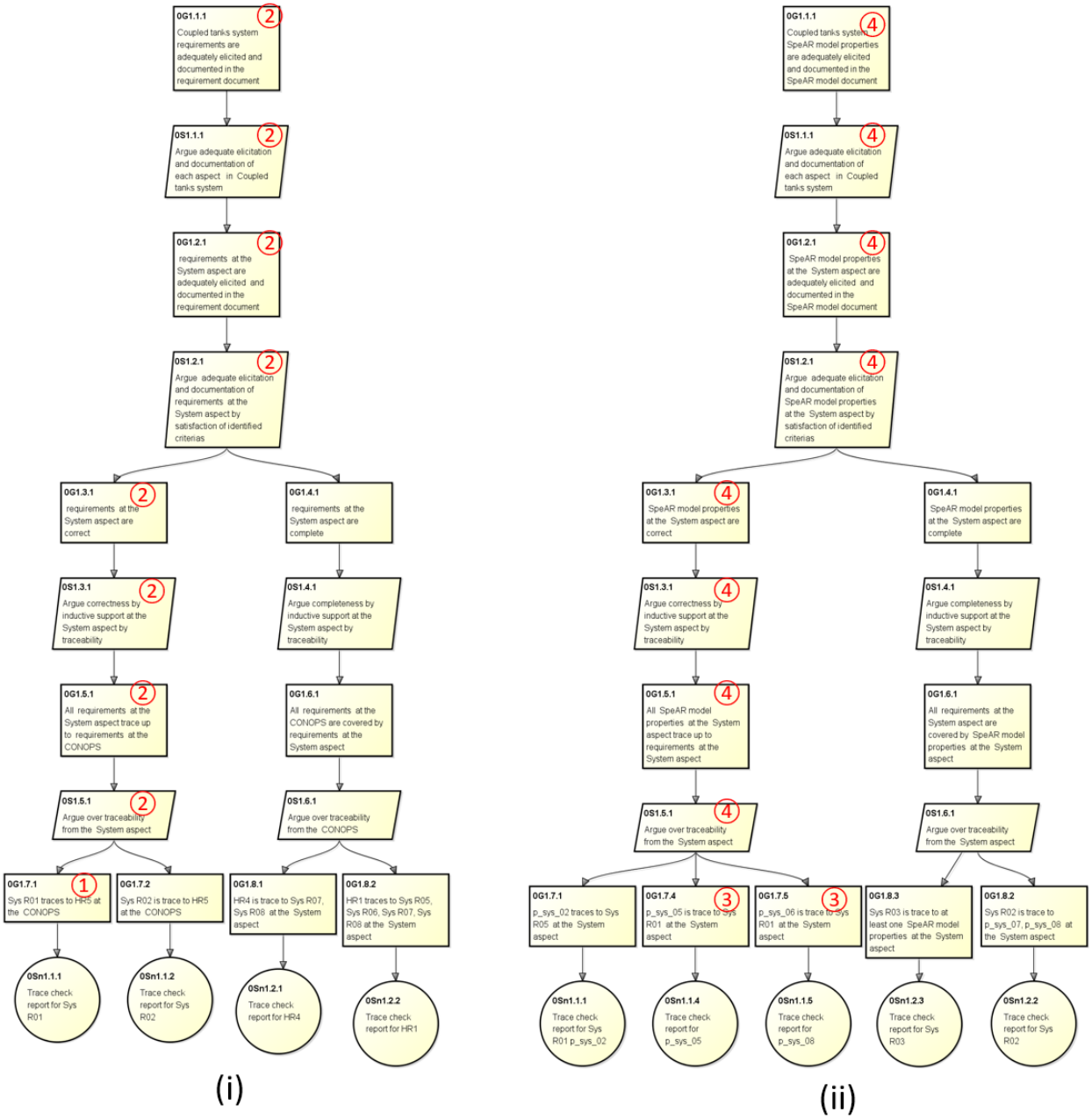


Figure 48 Coupled tanks control system assurance case maintenance example

nodes of *0G1.7.1* are identified. Therefore, these nodes are considered as affected nodes and marked with ② in Figure 48(i). Thirdly, SPIRIT identifies the affected nodes based on the traceability of system artifacts. In this case study example, the type of requirement *Sys R01* is domain model class *Requirement* as shown in Table 5. Based on the CPS domain model as shown in Figure 44, domain class *SystemProperty* is derived from *Requirement*. SPIRIT identifies SpeAR model properties *p_sys_05* and *p_sys_06* are derived from *Sys R01* based on the traceability information between the coupled tanks control system artifacts. SPIRIT identifies node *0G1.7.3* and *0G1.7.4* in the SpeAR model properties elicitation assurance case are affected by the change and marked with ③ in Figure 48(ii). Lastly, the ancestor nodes of *0G1.7.3* and *0G1.7.4* in Figure 48(ii) are also considered as affected nodes. These affected nodes are marked with ④ as shown in Figure 48(ii).

7.5 Evaluation Result

To support the certification of a generated assurance case, SPIRIT evaluates the assurance case for the coupled tanks control system via applying the Dempster Shafer (D-S) theory for the assurance case confidence calculation. To calculate the confidence of the coupled tanks control system requirement elicitation assurance case shown in Figure 46, SPIRIT generates the confidence calculation model for the requirement elicitation shown in Figure 49

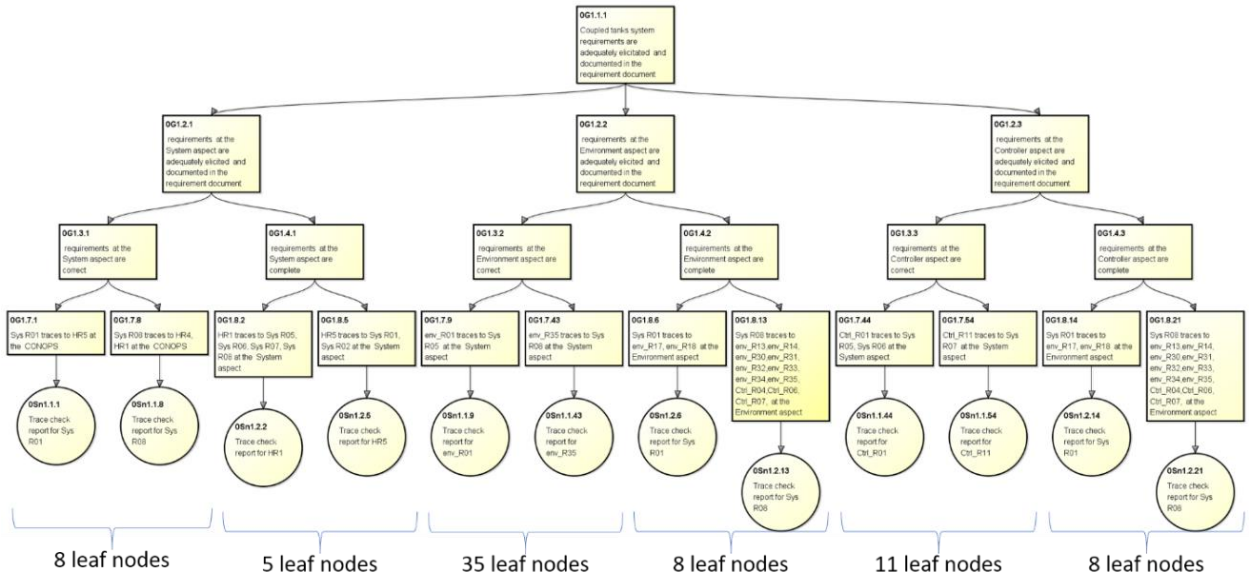


Figure 49 Confidence calculation model for requirements elicitation

based on the assurance case. In a confidence calculation model, all context nodes, assumption nodes, justification nodes, strategy nodes, and the intermediate goal nodes are eliminated from the assurance case. The confidence calculation model of the coupled tanks control system requirements elicitation is built from the highlighted goal nodes and the solution nodes in Figure 46. To calculate the confidence of an assurance case, the confidence of the evidence (solution nodes) must be known by SPIRIT. The confidence of evidence is usually decided by the certifier. But in some cases, the confidence for the evidence can also be calculated. In this case study example, the evidence to proof the traceability between requirements are correctly set up is based on how close these requirements are related to each other. One way to measure how close these requirements are related to each other is calculating the similarity between the description of a requirement and the description of another requirement. In other words, if two requirements are claimed to have a traceability between each other, the similarity value between these two documents provides a measurement that how confidence that the traceability is set correctly. In this case study, we use the Vector Space Model [47] to calculate the similarity between the requirements and use the similarity values to calculate the confidence of a solution node. Different models are applied to the assurance case confidence calculation such as Dempster-Shafer Theory (D-S theory) and Bayesian belief network (BBN). In this case study, we apply a confidence assessment framework proposed by Wang [45] to calculate the confidence of the generated assurance cases based on the D-S theory. In this confidence assessment framework, the confidence of the root node is based on two factors: 1) The *trustworthiness* to each sub-goal nodes in the confidence calculation model and 2) the *appropriateness* of the sub-goal nodes related to the root node. In this case study, the *trustworthiness* of a node is calculated based on the VSM similarity results. To specify the *appropriateness* of the sub-goal nodes related to its parent goal nodes and how the *trustworthiness* propagates from the sub-goal nodes to the parent goal node, additional information is required by the framework to construct a confidence calculation model. First, the *appropriateness* of a sub-goal node to its parent node is represented by a *disjoint contributing weight* called *w*. This value indicates that the degree that a sub-goal can independently contribute the *trustworthiness* to its parent goal. Second, the confidence assessment framework defines two argument types to specify how the *trustworthiness* of the sub-goal nodes contributed to the parent goal node. The two argument types are defined as *Dependent Argument* and *Redundant Argument*. The *Dependent Argument* indicates that the

contribution of a sub-goal to its parent node is depends on the *trustworthiness* of other sub-goal nodes. The *Redundant Argument* indicates that the sub-goal nodes have overlapping to each other to contribute the *trustworthiness* to the parent node.

In our case study, we manually review the structure of the generated assurance case and assign the type of argument for every level of the confidence calculation model. After we review the structure of the safety pattern, we assign the *Dependent Argument* type for all levels of the confidence calculation model. For the disjoint contributing weight of each node, we adopted a machine learning approach to find the most appropriate configuration for a confidence calculation model. To identify the most appropriate configuration of a confidence calculation model, we made an assumption that with the most appropriate configuration setting, the assurance case with correct traceability links between requirements at the different aspects should have higher root node confidence compared with an assurance case contain incorrect traceability links. Therefore, we define the most appropriate configuration of a confidence calculation model as following: 1) a configuration of a confidence calculation model is called an appropriate configuration if the calculation of the confidence based on the configuration of this confidence calculation model results in a higher value of the root node confidence compared with the root node confidence values of other confidence calculation models with the same structure but contains incorrect traceability links in the claim statements. 2) If there exist more than one appropriate configurations, the most appropriate configuration is the appropriate configuration that calculates the highest confidence value of the confidence calculation model. After the root node confidence of a confidence calculation model is calculated based on the most appropriate configuration, a decision scale is assigned to the root node based on its confidence value.

To compare the root node confidence between the assurance case with correct traceability and other assurance cases combined with correct and incorrect traceability, we randomly generate a set of assurance cases based on different traceability setting of the coupled tanks control system requirements elicitation assurance case. These random generated assurance case has the same structure compared with the original assurance case, but the leaf nodes confidences are different to the original assurance case. SPIRIT takes the original assurance case and the randomly generated assurance cases as the training set to identify the most appropriate configuration for the coupled tanks control system requirements elicitation confidence

calculation model. To generate an assurance case with random traceability, we consider the representative traceability sets for each goal node in the confidence calculation model that is supported by the solution nodes. The coupled tanks control system requirements elicitation assurance case contains 75 goal nodes that are supported by the solution nodes as shown in Figure 46. Recall that the claims that are supported by the solution nodes in the requirements elicitation assurance case contains a statement that a requirement, *R*, traces to a set of requirements, *S*, at aspect *A*, and *S* contains *N* requirements. To generate the training set assurance cases, a similarity table, *ST*, is generated for every requirement *R* to all requirements at the aspect *A*. In a similarity table, each row represents a requirement at aspect *A*, called *A'*, and the similarity value between *R* to *A'*. The similarity table are sorted based on the similarity value from the highest value to the lowest value.

In this case study, we choose four representative sets based on the similarity values for every requirement R and all requirements at aspect A stored in a similarity table. The four representatives' sets are: 1) The set of requirements S that has the best similarity rank with R, that is, select the top N requirements from ST. This set is called *SBest*, 2) The set of requirements S that are defined in the original assurance case, called *SIdeal*, 3) The set of requirements S that has the average similarity rank with R, that is, select the middle N requirements from ST. This set is called *SAverage*, and 4) The set of requirements S that has the worst similarity rank with R, that is, select the bottom N requirements from ST. This set is called *SWorst*. In this case study, the four representative sets are generated for all 75 goal nodes supported by the solution nodes. When a new assurance case is randomly generated, each of the 75 goal nodes selects one out of the four representative sets as the set S in the node statement, and the corresponding similarity values are identified for calculating the confidence of the supported solution node.

In the confidence calculation model for requirements elicitation, the configuration of the disjoint contributing weights as shown in Figure 50 includes weight *W1.2.1, W1.2.2, W1.2.3* for node *0G1.2.1, 0G1.2.2*, and *0G1.3.3*. Weight *W1.3.1* and *W1.4.1* for node *0G1.3.1* and *0G1.4.1*, weight *W1.3.2* and *W1.4.2* for node *0G1.3.2* and *0G1.4.2*, and weight *W1.3.3* and *W1.4.3* for node *0G1.3.3* and *0G1.4.3*. For the goal nodes that are directly supported by the solution nodes in the confidence calculation model, we assign the equal weights for the goal nodes support the same parent node. Based on the definition of the disjoint contributing weights in [45], the sum of the disjoint contributing weights for all sub-goal nodes support the same parent node is less than

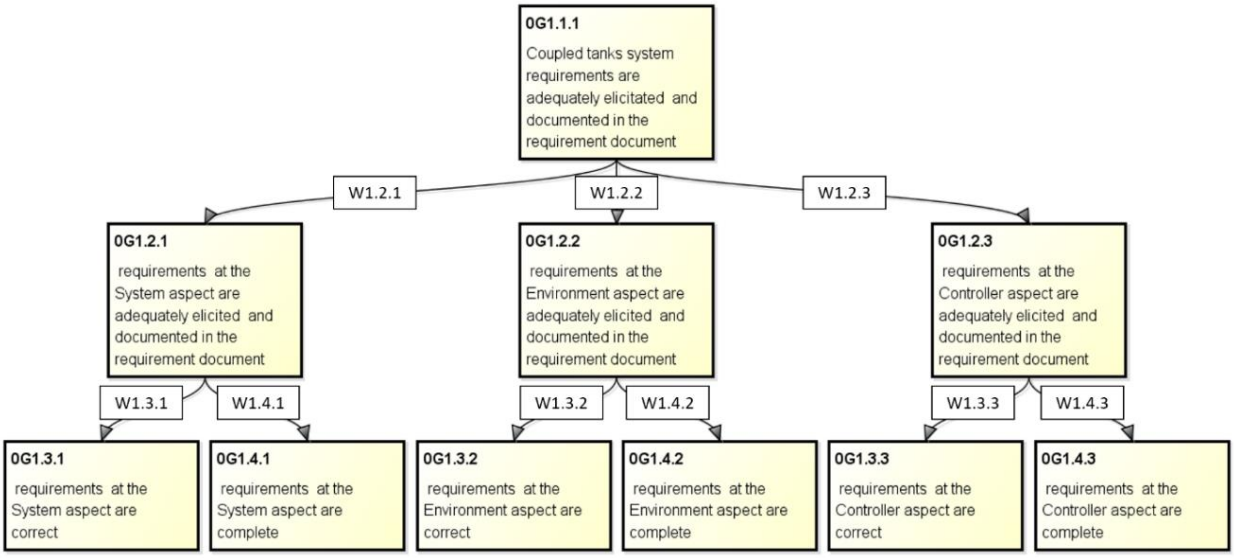


Figure 50 Disjoint contributing weights of the requirements elicitation confidence model

or equals to 1, and a factor called *CA* is introduced to represent the dependency and redundancy among sub-goals support the same parent node. The value of *CA* is between 0 and 1, and the sum of *CA* and the disjoint contributing weights for all sub-goal nodes support the same parent node is 1. In this case study, we define the value of *CA* to 0.1, and the sum for the disjoint contributing weights for all sub-goal nodes support the same parent node is 0.9 (e.g. W1.2.1 + W1.2.2 + W1.2.3 = 0.9, and W1.3.1 + W1.4.1 = 0.9). In order to identify the most appropriate configuration for these disjoint contributing weights, we define the minimum weight value for

Table 6 Configuration parameters table

Group	Node ID	Weight ID	Min value	Max value	CA	increments	# of weight combinations
G1	0G1.2.1	W1.2.1	0.1	0.7	0.1	0.1	28
	0G1.2.2	W1.2.2	0.1	0.7		0.1	
	0G1.2.3	W1.2.3	0.1	0.7		0.1	
G2	0G1.3.1	W1.3.1	0.1	0.8		0.1	8
	0G1.4.1	W1.4.1	0.1	0.8		0.1	
G3	0G1.3.2	W1.3.2	0.1	0.8		0.1	8
	0G1.4.2	W1.4.2	0.1	0.8		0.1	
G4	0G1.3.3	W1.3.3	0.1	0.8		0.1	8
	0G1.4.3	W1.4.3	0.1	0.8		0.1	
Constraint	**Expression**						
1	W1.2.1 + W1.2.2 + W1.2.3 + CA = 1						
2	W1.3.1 + W1.4.1 + CA = 1						
3	W1.3.2 + W1.4.2 + CA = 1						
4	W1.3.3 + W1.4.3 + CA = 1						

every node is 0.1, and we consider all possible combinations for all disjoint contributing weights with step 0.1. A table shown in Table 6 illustrates the configuration parameters of the coupled tanks control system requirements elicitation confidence calculation model. In this case study, SPIRIT verifies all 14,336 combinations (28*8*8*8) and identity the most appropriate configuration for the confidence calculation model based on the original assurance case and 299 randomly generated assurance cases as the training set for the identification of the most appropriate configuration. To evaluate the configuration identified by SPIRIT, we design an experiment to evaluate the most appropriate configuration result. In this case study, the experiment contains 100 runs. For each run of the experiment, we randomly generate 299 assurance case and compare the root node confidence with the original assurance case. We apply the most appropriate configuration for these 300 assurance cases, the result is a rank value indicates the root node confidence of the original assurance case among the 300 assurance cases. The rank value for each run is between 1 to 300, and we calculate the average rank for 100 runs. In this experiment, the average rank for the coupled tanks control system is 2.71. Compare with the ideal average rank, 1, this experiment result shows that with the most appropriate configuration identified by SPIRIT, the root node confidence for the confidence calculation model derived from the original assurance case with the correct traceability is higher than the confidence calculation models derived from the assurance cases with incorrect traceability, which is corresponding to our expectation. And the confidence value of this root node is around 0.772, which is corresponding to the *tolerate* decision scale.

7.6 Evaluation Result Analysis

In this section, we evaluate the SPIRIT framework based on two case studies. Based on the safety patterns used in the two case studies, we preset the value of c, *i.e.*, the total disjoint contributing weights of all child claims, to 0.9, each argument type to redundant, and the value of v to 1. For the training phase, we find that the first assurance case using the input of the training phase is acceptable based on the disjoint contributing weights derived by the framework. We further investigate the following questions:

Q1: Can the training phase produce an appropriate set of disjoint contributing weights based on the first acceptable assurance case?

In order to answer this question, we do experiments based on the following two criteria. The first criterion is the rank and the other is the relationship between the appropriateness and trustworthiness of a claim. Since the first assurance case is acceptable, an appropriate set of disjoint contributing weights should make the assurance case have a higher rank than most other assurance cases, independent of how a training data set is generated. As for the relationship criterion, we find that when a certifier compares two claims having the same parent claim, in most cases, if claim A has a larger trustworthiness value than claim B, then the appropriateness value of A to its parent claim has a larger value than the appropriateness value of B.

Q2: Can the disjoint contributing weights generated by the training phase be successfully used in the application phase?

We compare the framework with randomly generated disjoint contributing weights by adopting the Matthews Correlation Coefficient (MCC) that has been widely used by researchers [52]. The MCC is used in machine learning as a measure of the quality of binary (two-class) classifications. Taking into account true and false positives and negatives, the MCC is essentially a correlation coefficient between the observed and predicted binary classifications; it returns a value between −1 and +1. A coefficient of +1 represents a perfect prediction, 0 means no better than random prediction, and −1 indicates total disagreement between prediction and observation. A value is between 0 and 1 means the prediction is better than a random prediction while a value between -1 and 0 indicates the prediction is worse than a random prediction. In our scenario, the MCC is employed to indicate whether the derived set of disjoint contributing weights by the framework is better than a randomly generated set.

Before answering the two research questions, we briefly introduce the gear controller system that was designed according to the informal requirements delivered by Mecel AB to illustrate the applicability of UPPAAL [53]. In the gear controller system, there are five components, each of which was modeled by a timed automaton according to the original requirements from Mecel AB. Meanwhile, safety and liveness requirements are converted from the informal description of the system to UPPAAL queries. The UPPAAL queries are further validated against the timed automata to ensure all safety and liveness requirements are satisfied. In this case study, we employ two assurance cases that claim the correct design of time automata and derivation of the UPPAAL queries respectively shown in Figure 51 and Figure 52. Similar to the coupled tanks system, the assurance cases for the gear controller system are generated by SPIRIT based on the

domain model shown in Figure 53, the safety pattern shown in Figure 54, the mapping tables shown in Table 7, and the system artifacts shown in Table 8. For the assurance case to claim the

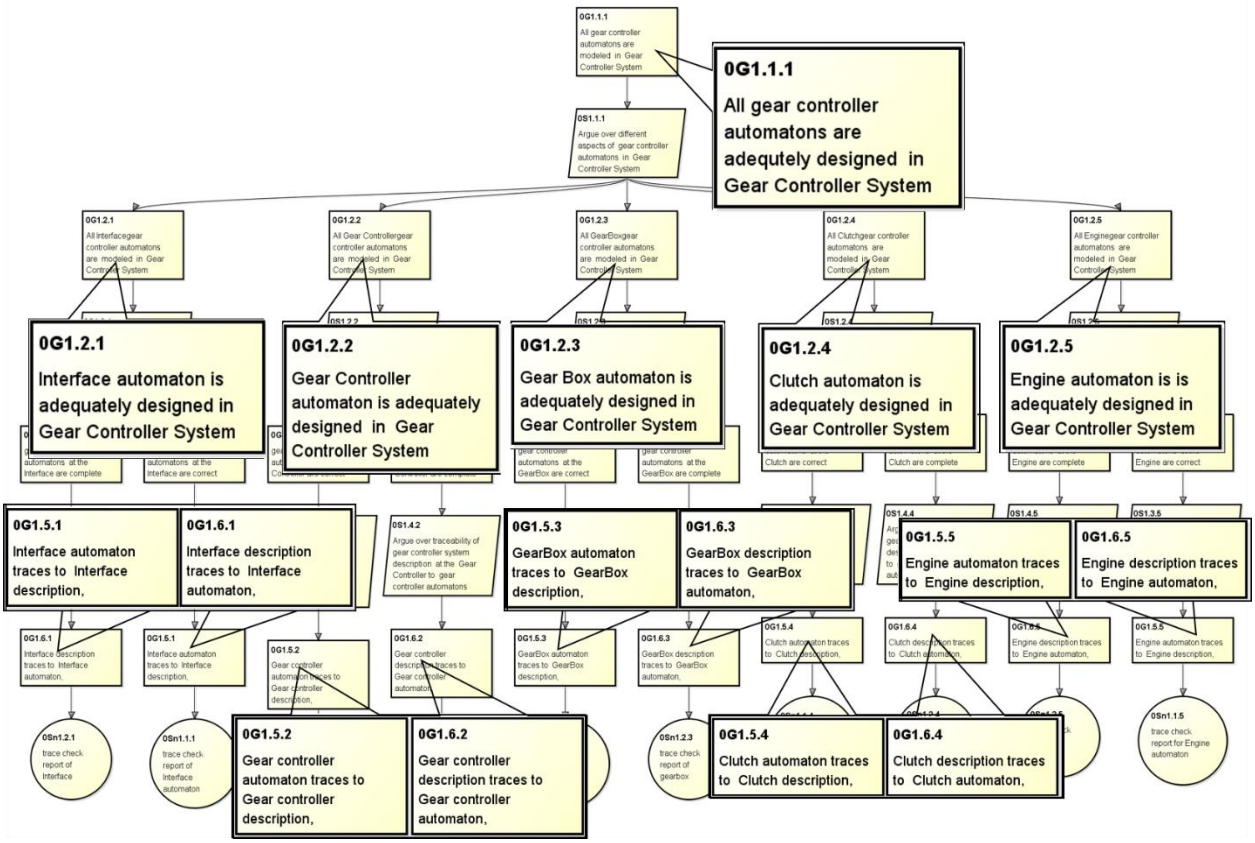


Figure 51 Gear controller system design of automatons assurance case

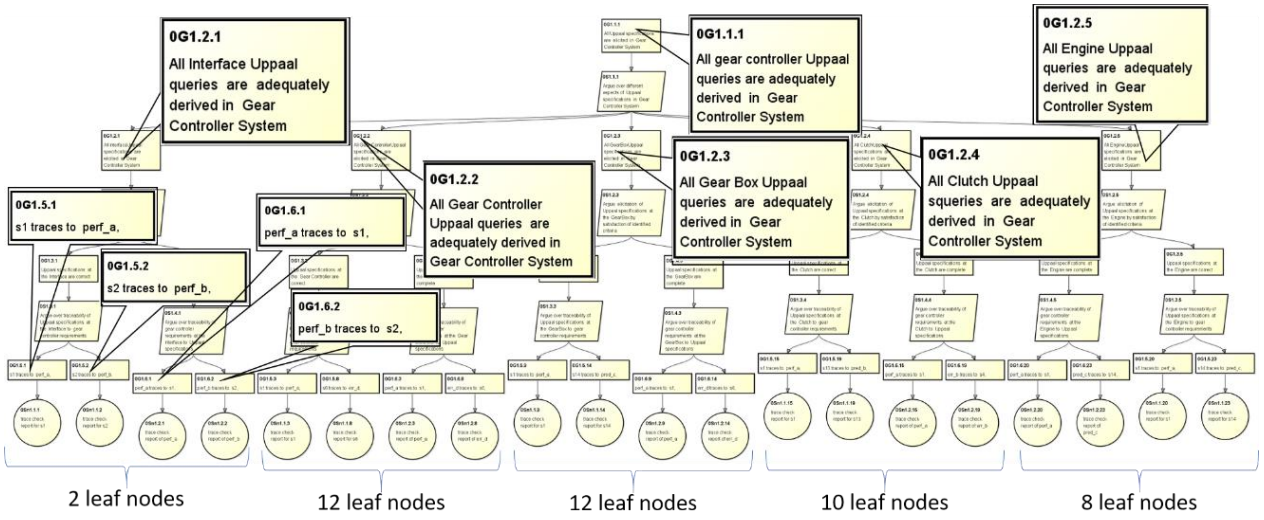


Figure 52 Gear controller system derivation of UPPAAL queries assurance case

design of time automata being adequate, the assurance case starts with the top claim, i.e., “All gear controller automatons are adequately designed in the gear controller system”. For the second assurance case to claim the derivation of UPPAAL queries being correct, we start with the top claim, i.e., “All gear controller UPPAAL queries are adequately derived in the gear controller system”.

To support the top claim, both assurance cases employ a strategy that targets the five components in the gear controller system, i.e., the Interface, Gear Controller, Gearbox, Engine, and Engine components. For each component, one child claim, labelled as *0G1.2.x* where *x* is {1, 2, …, 5}, is formed to ensure the automaton for the component is adequately designed. Due to space constraints, we only highlight the key nodes in Figure 51 and Figure 52.

For each sub-claim 0G1.2.x, two sub-claims are further developed to support its correctness and completeness via a strategy node 0S1.2.x. For the assurance case on automata, the correctness child claim ensures that the automaton of a component traces to the corresponding component description, and the completeness child claim asserts that the component description matches the automaton in terms of similarity. For the assurance case on the UPPAAL queries, the correctness child claim ensures that the UPPAAL queries related to a specific component can trace to the related system requirements, and the completeness child

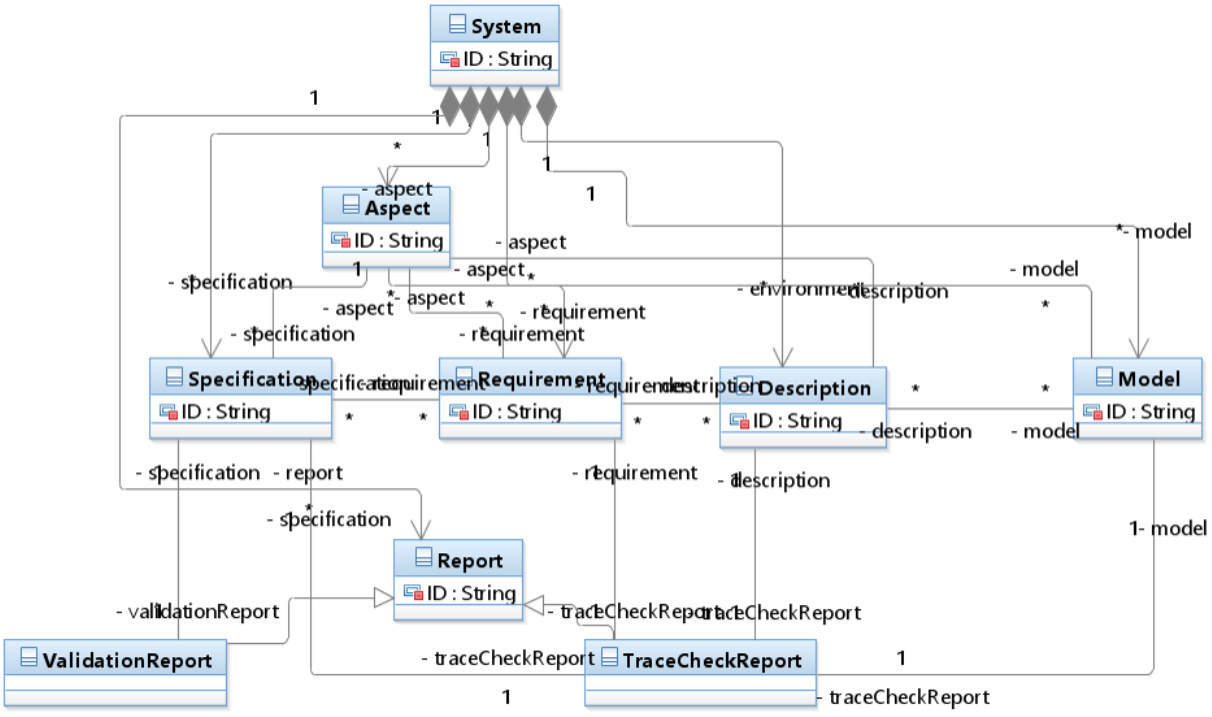


Figure 53 Domain model of the gear controller system

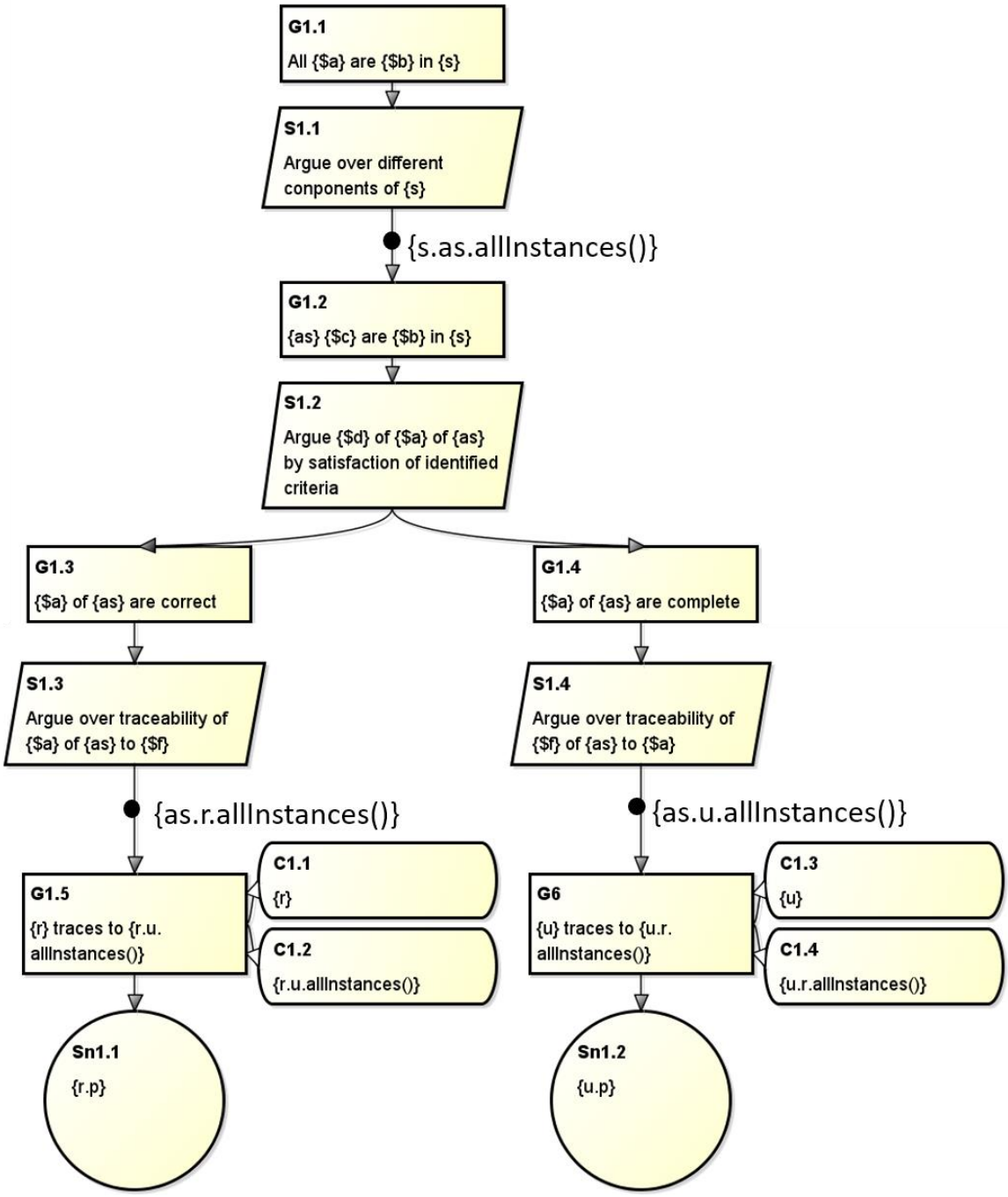


Figure 54 Safety patterns of the gear controller system

claim asserts that the system requirements related to a component match the UPPAAL queries in terms of similarity. At the bottom part of the two assurance cases, both the correctness claim and the completeness claim are supported by traceability check reports between the related system artifacts.

Table 7 Mapping tables of the gear controller system

Variable Role	Domain Model Class	Association Role Name
s	System	NULL
as	Aspect	aspect
u	Description	description
r	Model	model
p	TraceCheckReport	traceCheckReport
String Role	**Value**	
{$a}	gear controller automatons	
{$b}	adequately designed	
{$c}	automaton	
{$d}	design	
{$f}	gear controller descriptions	

i) Configuration table for the design of automatons

Variable Role	Domain Model Class	Association Role Name
s	System	NULL
as	Aspect	aspect
u	Requirement	requirement
r	Specification	specification
p	TraceCheckReport	traceCheckReport
String Role	**Value**	
{$a}	Gear controller UPPAAL queries	
{$b}	adequately derived	
{$c}	UPPAAL queries	
{$d}	derivation	
{$f}	gear controller system requirements	

ii) Configuration table for the derivation of UPPAAL queries

Answer to Q1: the rank aspect. To determine whether the first acceptable assurance case has a higher rank than the other assurance cases, we first fix the derived set of disjoint contributing weights given by the training phase and generate 100 groups of experimental data, each of which consists of 300 assurance cases including the first acceptable assurance case. Except for the acceptable assurance case, there is no assurance case that occurs in two or more groups. Next, we apply the disjoint contributing weights derived from the training phase to these 100 groups and then find the ranks of the acceptable assurance case in these groups.

Table 8 Gear controller system artifacts

System	Aspect	Description	Model
Gear Controller System	Interface	Interface description	Interface automaton
	Gear Controller	Gear Controller description	Gear Controller automaton
	Gear Box	Gear box description	Gear box automaton
	Clutch	Clutch description	Clutch automaton
	Engine	Engine description	Engine automaton

System	Aspect	Requirement	Specification
Gear Controller System	Interface	perf_a, perf_b	s1 s2
	Gear Controller	perf_a, perf_b, err_a, err_b err_c, err_d	s1, s2, s3 S4, s5, s6
	Gear Box	perf_a, perf_b pred_c, func_a err_c, err_d	S1, s2, s5 S6, s12, s14
	Clutch	perf_a, perf_b pred_b, err_a err_b	S1, s2, s3 S4, s13
	Engine	perf_a, perf_b pred_b, pred_c	S1, s2 S13, s14

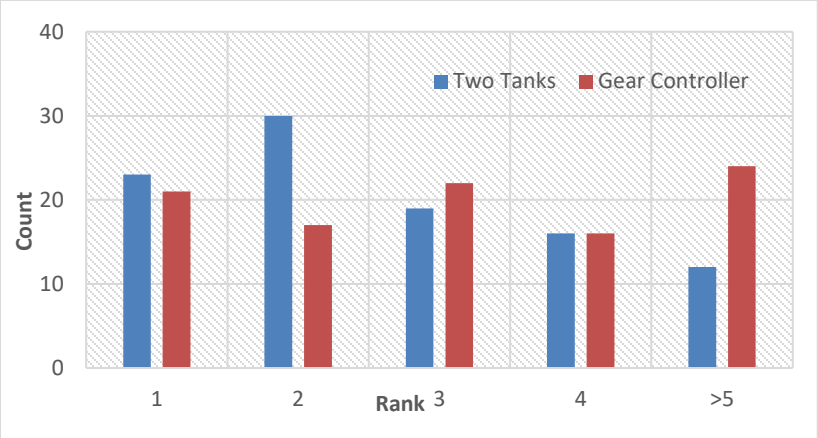


Figure 55 Training result

Table 9 Compare the derived weights with the randomly generated weights in training phase (Coupled tanks)

Node ID	Derived Weights	Random Weights 1	Random Weights 2	Random Weights 3	Random Weights 4	Random Weights 5
0G1.2.1	0.7	0.3	0.1	0.1	0.2	0.7
0G1.2.2	0.1	0.1	0.4	0.3	0.1	0.1
0G1.2.3	0.1	0.5	0.4	0.5	0.6	0.1
0G1.3.1	0.8	0.3	0.6	0.6	0.3	0.8
0G1.4.1	0.1	0.6	0.3	0.3	0.6	0.1
0G1.3.2	0.7	0.1	0.1	0.2	0.4	0.1
0G1.4.2	0.2	0.8	0.8	0.7	0.4	0.8
0G1.3.3	0.1	0.4	0.8	0.7	0.6	0.1
0G1.4.3	0.8	0.5	0.1	0.2	0.3	0.8
Avg Rank	**2.71/300**	**5.3/300**	**10.71/300**	**6.93/300**	**5.32/300**	**3.93/300**

Table 10 Compare the derived weights with the randomly generated weights in training phase (Gear system)

Node ID	Derived Weights	Random Weights 1	Random Weights 2	Random Weights 3	Random Weights 4	Random Weights 5
0G1.2.1	0.1	0.3	0.1	0.2	0.3	0.2
0G1.2.2	0.1	0.2	0.2	0.1	0.1	0.1
0G1.2.3	0.1	0.2	0.3	0.4	0.2	0.2
0G1.2.4	0.1	0.1	0.2	0.1	0.1	0.2
0G1.2.5	0.5	0.1	0.1	0.1	0.2	0.2
0G1.3.1	0.1	0.1	0.7	0.5	0.5	0.1
0G1.4.1	0.8	0.8	0.2	0.4	0.4	0.8
0G1.3.2	0.1	0.3	0.1	0.1	0.7	0.7
0G1.4.2	0.8	0.6	0.8	0.8	0.2	0.2
0G1.3.3	0.1	0.7	0.7	0.1	0.5	0.5
0G1.4.3	0.8	0.2	0.2	0.8	0.4	0.4
0G1.3.4	0.7	0.3	0.1	0.1	0.1	0.7
0G1.4.4	0.2	0.6	0.8	0.8	0.8	0.2
0G1.3.5	0.1	0.5	0.7	0.5	0.3	0.5
0G1.4.5	0.8	0.4	0.2	0.4	0.6	0.4
Avg Rank	**3.28/300**	**13.73/300**	**17.06/300**	**3.99/300**	**5.54/300**	**3.37/300**

For the coupled tanks system case study, the rank distribution of the acceptable assurance case among the 100 groups is shown in Figure 55, where the acceptable assurance case ranks first among 23 groups, second among 30 groups, third among19 groups, fourth among 16 groups, and fifth or higher among 16 groups. The result shows that the average trustworthiness value of the acceptable assurance case is 0.772, and the average rank is 2.71, which means the acceptable assurance case has a better trustworthiness value than more than 99% of the randomly generated assurance cases using the derived set.

For the gear controller system case study, the rank distribution of the acceptable assurance case among the 100 groups is also shown in Figure 55, where the acceptable assurance

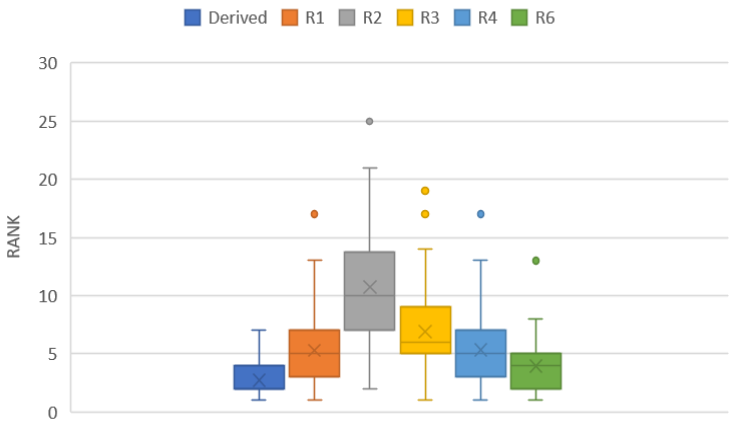


Figure 56 A boxplot of ranks for the coupled tanks system

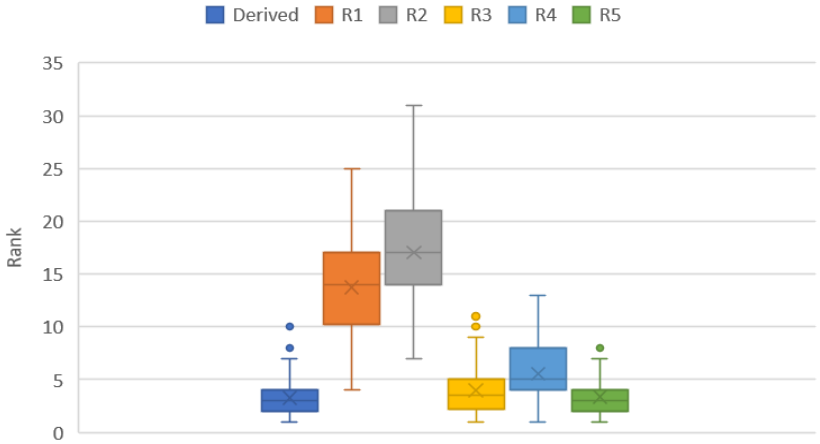


Figure 57 A boxplot of ranks for the gear controller system

case ranks first among 21 groups, second among 13 groups, third among 22 groups, fourth among 16 group, and fifth or higher among 24 groups. The average trustworthiness value of the acceptable assurance case is 0.752 and the average rank is 3.28, indicating the acceptable assurance case has a better trustworthiness value than more than 98.9% of the randomly generated assurance case.

Next, we generate 5 sets of disjoint contributing weights and apply to the above 100 sets of training data so we can check whether the derived set outperforms the random set in terms of the rank. The results for the two case studies are shown in Table 9 and Table 10 respectively. Furthermore, we employ the boxplot to demonstrate the rank distributions based on the experimental data for both case studies, which are shown in Figure 56 and Figure 57

respectively. The conclusion is that average rank of the derived set is higher than the randomly selected 5 sets.

Answer to Q1: the relationship aspect. To identify the relationship between a trustworthiness value, i.e., a *dec* value showing the certifier's perspective, and a disjoint contributing weight of a claim, we do two experiments. The first one is to randomly generate 5 assurance cases with different *dec* values for child claims to study how the appropriateness values of these children claims are distributed. The second experiment tries to fix the trustworthiness value of a child claim and change the trustworthiness values of the other children claims to see how the appropriateness values of children claims are distributed. For simplicity, we only consider the top-level assurances cases.

In the first experiment, we generate 5 assurance cases for both the coupled tanks system case study and the gear controller case study. To observe the relationship between the trustworthiness values of a claim and the derived weight value, we focus on the trustworthiness values as well as appropriateness values of claims related to the top argument. The experimental results in Table 11 confirms that a node with the highest *dec* value in an assurance case leads to a largest weight value for all 10 assurance cases.

The second experiment tries to fix the trustworthiness value of a child claim and change trustworthiness values of the rest child claims to see how the appropriateness value of fixed child claim is changed. For each experiment, we select one child claim and fix the trustworthiness value of the selected claim, and then change the trustworthiness value of the other child claims using the ideal similarity, average similarity, and worst similarity values. To do so, we can change the rank of the fixed branch so its appropriateness value change can be observed. The

Table 11 Relationship between trustworthiness and disjoint contributing weights

i) Coupled tanks system

	AC1		AC2		AC3		AC4		AC5	
	Dec	Weight	Dec	Weight	Dec	Weight	Dec	Weight	Dec	Weight
0G1.2.1	0.822784	0.7	0.743754	0.7	0.647367	0.1	0.647367	0.1	0.743754	0.7
0G1.2.2	0.706655	0.1	0.665824	0.1	0.49689	0.1	0.706655	0.7	0.706655	0.1
0G1.2.3	0.700371	0.1	0.700371	0.1	0.700371	0.7	0.514099	0.1	0.654852	0.1

ii) Gear controller system

	AC1		AC2		AC3		AC4		AC5	
	Dec	Weight	Dec	Weight	Dec	Weight	Dec	Weight	Dec	Weight
0G1.2.1	0.861944	0.5	0.822136	0.1	0.562081	0.1	0.067554	0.1	0.562081	0.1
0G1.2.2	0.680194	0.1	0.680194	0.1	0.680194	0.5	0.680194	0.5	0.385045	0.1
0G1.2.3	0.83019	0.1	0.761678	0.1	0.474456	0.1	0.091039	0.1	0.761678	0.5
0G1.2.4	0.840019	0.1	0.740005	0.1	0.354443	0.1	0.050245	0.1	0.354443	0.1
0G1.2.5	0.858963	0.1	0.835005	0.5	0.443067	0.1	0.06507	0.1	0.443067	0.1

root claim of coupled tanks system case study has 3 children claims so that there are 9 experiments to test all combinations. For the coupled tanks case study, we show the results when the three children claims are each fixed in Figure 58 (i), (ii), and (iii) respectively. For the gear controller system case study, the root claim has 5 children claims, and so we have 15 experiments for configurations. Likewise, for the gear controller case study, the results when the five children claims are each fixed are shown in Figure 59 (i) to (v) respectively. Note that the total appropriateness value of all children claims is 0.9 and we use 0.1 as the step function. So, when a rank of a fixed child claim is improved to 1, its disjoint contributing weight value is increased to reach a maximum value, i.e., 0.7 for the coupled tanks case study and 0.5 for the gear controller case study. Likewise, when a rank of a fixed child claim drops from 1, its disjoint contributing weight value is also decreased from a maximum value too and hits a lowest value when the rank drops to a bottom level.

Answer to Q2. To find whether a derived set of the disjoint contributing weights from the training phase can be successfully used in the application phase, we manually generate 100 assurance cases with the same structure as the second assurance case. The first criterion of the manual review is based on the average trustworthiness value of the entire assurance case. We first set a threshold value to be an average trustworthiness value of all leaf claims in the 100 assurance cases. The rationale behind this criterion is that if an average trustworthiness value is too low, then it is impossible for the case to be accepted. Next, we select all assurance cases whose average trustworthiness value is greater than the threshold value for further manual

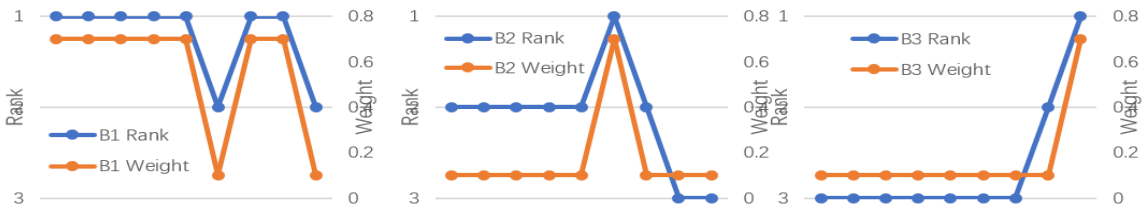


Figure 58 Relation between rank and weight (Coupled Tanks)

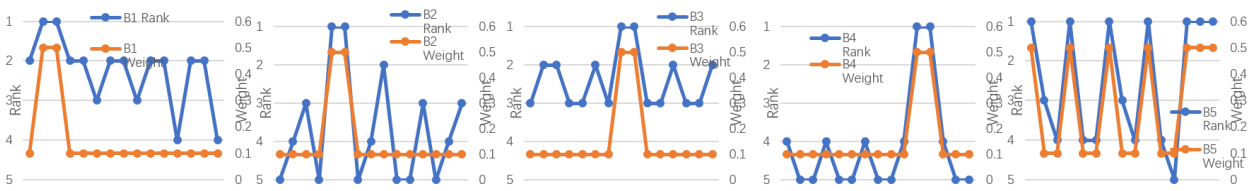


Figure 59 Relation between rank and weight (Gear Controller)

Table 12 Assurance cases that are accepted by the manual review

ID	Average Trustworthiness of root node	Average Trustworthiness of 0G1.2.1
1	0.8033282	0.8444193
2	0.7951467	0.8446155
3	0.8010616	0.8444193
4	0.7953813	0.8393353
5	0.7673455	0.8445965
6	0.7670239	0.8370047
7	0.7809744	0.8648005
8	0.763875	0.8444193
9	0.7782355	0.8444193
10	0.7553595	0.8225925
11	0.7636479	0.8296573
12	0.7560142	0.8444193
13	0.7887146	0.8567398
14	0.7537787	0.8607754
15	0.7609353	0.8432416
16	0.7834818	0.8663686
17	0.7444917	0.8360348
18	0.7372574	0.8189912
19	0.7333541	0.8516003
20	0.7487461	0.8394918
21	0.7470645	0.8182725
22	0.7476877	0.8422003
23	0.7523687	0.8294141
24	0.7457807	0.839777
25	0.7358557	0.8141716

i) Coupled tanks system

ID	Average Trustworthiness of root node	Average Trustworthiness of 0G1.2.2
1	0.890721463	0.876639553
2	0.911376032	0.876639553
3	0.916451572	0.949798468
4	0.890721463	0.876639553
5	0.92296373	0.839364599
6	0.88661995	0.876639553
7	0.926230508	0.855698489
8	0.904957743	0.949798468
9	0.871283271	0.876639553
10	0.86253593	0.876639553
11	0.918555057	0.817321233
12	0.87325483	0.876639553
13	0.901819789	0.876639553
14	0.841673643	0.876639553

ii) Gear controller system

review. For the coupled tanks system assurance cases, there are a total of 45 assurance cases selected for manual review and the other 55 assurance cases are immediately rejected. For the second phase of manual review, after communicating with the developers on the system design multiple times, we find the System aspect artifacts play an important role in the early design and analysis process in that the system aspect, as the first step of the process, is further decomposed to the environmental aspect and controller aspect. Thus, we further review the 45 assurance cases selected from the first phase. Among the 45 assurance cases, we manually select 25 assurance cases as the acceptable set based on the assurance structure under the first branch, i.e. the claim 0G1.2.1 in Figure 46.

Next, we calculate the MCC value of the coupled tanks case study by applying the derived set of the disjoint contributing weighs from the training phase to the 100 assurance cases. The experiment result shows that the set of derived disjoint contributing weights can correctly

identify 22 assurance cases that acceptable (True positive) among the originally accepted 25 assurance cases; and 71 assurance cases that are unacceptable (True negative) among the originally rejected assurance cases. Also, 3 acceptable assurance cases are rejected by the set of derived disjoint contributing weights (False negative) and 4 unacceptable assurance cases are accepted by the set (False positive). The acceptable assurance cases that are passed the manual review are shown in Table 12(i). The MCC value of our framework is 0.816. In this case, we conclude that the derived set of the disjoint contributing weights from the training phase is better than a random set for the coupled tanks case study.

For the gear controller system assurance cases, we employ the same criterion for the first phase by calculating a threshold value based on an average trustworthiness value of all leaf claims in the 100 assurance cases. Thus, there are totally 44 assurance cases selected for manual review and the rest 56 assurance cases are immediately rejected due to a low average trustworthiness value. Next, we study the five components in the gear controller system and concentrate on the gear controller artifacts since the gear controller is the main part in this case study, as is confirmed by one of the authors who has extensive experience in this case study since he actively involved to the design and verification in the past. Based on this criterion, after reviewing the 44 assurance cases, we manually select 14 assurance cases as the acceptable set.

Likewise, we use the same manner to calculate the MCC value for the gear controller case study. We calculate the MCC value by applying the derived set of the disjoint contributing weighs from the training phase to the selected 100 assurance cases. In this scenario, the experiment result shows that using the set of derived disjoint contributing weights the framework has 13 assurance cases acceptable (True positive) among the 14 acceptable assurance cases; and 77 assurance cases unacceptable (True negative) among the originally rejected assurance cases. Furthermore, 1 acceptable assurance case is rejected by the framework (False negative) and 9 unacceptable assurance cases are accepted by the framework (False positive). The acceptable assurance cases that are passed the manual review are shown in Table 12(ii). The MCC value of

Table 13 MCC value in application phase

	TP	TN	FP	FN	MCC
Coupled Tanks	22	71	4	3	**0.816072**
Gear Controller	13	77	9	1	**0.690144**

our framework is 0.69. The detailed results for both case studies are shown in Table 13. In this case, we conclude that the derived set of the disjoint contributing weights from the training phase is indeed better than a random set for the gear controller case study.

CHAPTER VIII
CONCLUSION

8.1 Concluding Remarks

This dissertation has developed the SPIRIT framework to support the validation and certification of a system during the development process. The validation of the domain model activity as described in chapter III helps the system developer to check the proposed domain model against to the standard documents. To do that, SPIRIT takes a standard conceptual model as input and automatically derive a UML profile with OCL constraints to represent the standard. When a user provides a domain model to SPIRIT, a validation report is generated to inform the user weather the domain model satisfies the standard document or not by evaluating the OCL constraints. To demonstrate whether a software system has been developed to satisfy software assurance, software engineers should provide an argumentation structure called assurance case which lays down all arguments made behind each step or activity during an SDLC as well as the relevant artifacts as evidence. The assurance case generation activity as described in chapter IV helps the system developer automatically builds an assurance case based on the system artifacts. To reach the automatic generation of an assurance case, SPIRIT first extends the GSN safety pattern to define the relation between roles and achieved unambiguous instantiation when an assurance case is generated from the safety pattern, which provides the user to reuse successful safety case patterns in constructing convincing safety cases for their own devices in an efficient manner. The safety pattern catalog provides the description of what a safety pattern does and thus the guidance about how to use a safety pattern. To improve the flexibility of the use of safety patterns, SPIRIT supports two types of pattern connections based on the type of a node involved in the connection to build a complete safety pattern for a specific domain, called domain specific pattern. To generate an assurance case from a domain specific pattern, SPIRIT converts the domain specific pattern as an ATL model transformation program. In a generated ATL program, each node defined in the domain specific pattern is represented as a rule and the relationship between nodes is captured via the invocation of rules. Once the ATL program is created by SPIRIT, the execution of ATL model transformation generates an assurance case for the specific system. Since system artifacts may be changed during the SLDC of a system, it is

important for the system developer to identify the impaction of the system caused by the modification of system artifacts. SPIRIT maintenance feature as described in chapter V identifies the system artifacts and the assurance case nodes that may be affected by the modification of a system artifact, and then generates a report to the system developer. Once an assurance case is generated, it is important for the certifier to review an assurance case to ensure the root goal of an assurance case is sufficiently supported based on the argument structure of the assurance case and the system artifacts as evidence. To help the certifier to review a generated assurance, the SPIRIT assurance case evaluation feature supports the calculation of the confidence of a generated assurance case. In SPIRIT, a confidence calculation model is automatically generated from an assurance case. With the use of D-S theory, the confidence of each node in a confidence calculation model is calculated by SPIRIT.

To evaluate the use of SPIRIT, we apply SPIRIT to the coupled tanks control system and the gear controller case studies as described in chapter VI. In this case study, a safety pattern catalog is generated as shown in Figure 43 to claim the system artifacts are correct and complete elicited and documented. With a domain model for the Cyber Physical System shown in Figure 44 and the mapping tables provide the mapping relationships between roles and domain classes as shown in Table 4 and Table 3 as inputs, SPIRIT create two domain specific patterns as shown in Figure 45 and derive two ATL programs to represent the domain specific patterns. The assurance cases are generated based on the coupled tanks control system artifacts as summarized in Table 5. After the assurance cases are generated for the coupled tanks control system, we analyze how the assurance case nodes are affected by the modification of a coupled tanks control system requirement. The evaluation of the coupled tanks control system assurance cases consists of two steps. The first step takes the confidence calculation model derived from the coupled tanks control system assurance case as input to identify the most appropriate configuration based on the machine learning approach. The experiment based on the most appropriate configuration shows that the assurance case with correct traceability settings has higher average ranks compared with the assurance case contains incorrect traceability settings. In the end, we analyze the configuration identified by SPIRIT and apply the configuration to the second assurance case. The result shows that the identified configuration can be applied to other assurance cases with similar structure.

8.2 Future Works

One important feature can be developed in the future is to support the run-time adaption of the assurance case. Since SPIRIT support the generation of an assurance case during different phases of software development life cycle, it is important to support the user to edit or update an assurance case to reflect the change of system. This work can include an interface to connect the domain model, system artifacts, and the assurance case. The interface will monitor any modification of the system artifacts and automatically update the assurance case to reflect the modification. Once an assurance case is updated, the confidence of the new assurance case will be calculated and report to the user for further analyze.

REFERENCES

[1] RTCA, "DO-178C Software considerations in airborne systems and equipment certification," 2011.

[2] P. Ammann and J. Offutt, Introduction to Software Testing, 1 ed., New York, NY, USA: Cambridge University Press, 2008.

[3] Adelard, "Claims, Arguments and Evidence (CAE)," [Online]. Available: https://www.adelard.com/asce/choosing-asce/cae.html.

[4] Goal Structuring Notation Working Group, GSN Community Standard Version 1, 2011.

[5] European Organisation for the Safety of Air Navigation, "Preliminary Safety Case for Airports Surface Surveillance," Eurocontrol, 2011.

[6] R. France and B. Rumpe, "Model-Driven Development of Complex Software: A Research Roadmap," in *Proceedings of Future of Software Engineering 2007*, 2007.

[7] A. Langari and T. Maibaum, "Safety Cases: A Review of Challenges," 2013.

[8] T. Kelly and J. McDermid, "Safety Case Construction and Reuse Using Patterns," in *Proceedings of SAFECOMP' 97*, 1997.

[9] 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.

[10] 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.

[11] 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.

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

[13] E. W. Denney and G. J. Pai, "Safety Case Patterns: Theory and Applications," NASA/TM–2015–218492, 2015.

[14] 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.

[15] E. Jee, I. Lee and O. Sokolsky, "Assurance Cases in Model-Driven Development of the Pacemaker Software," LNCS 6416, 2010.

[16] K. Attwood and P. Conmy, "Nuanced term-matching to assist in compositional safety assurance," in *1st International Workshop onAssurance Cases for Software-Intensive Systems (ASSURE)*, 2013.

[17] A. L. Juarez Dominguez, B. G. Partridge and J. J. Joyce, "Creating safety assurance cases for rebreather systems," in *1st International Workshop on Assurance Cases for Software-Intensive Systems (ASSURE)* , 2013.

[18] A. Ray and R. Cleaveland, "Constructing safety assurance cases for medical devices," in *1st International Workshop on Assurance Cases for Software-Intensive Systems (ASSURE)* , 2013.

[19] R. Hawkins, T. Kelly, J. Knight and P. Graydon, "A New Approach to Create Clear Safety Arguments," in *Nineteenth Safety-Critical Systems Symposium*, Southampton, UK, 2011.

[20] A. Ayoub, B. Kim, I. Lee and O. Sokolsky, "A systematic approach to justifying sufficient confidence in software safety arguments," in *Computer Safety, Reliability, and Security*, Springer, 2012, pp. 305-316.

[21] E. Denney, P. Ganesh and H. Ibrahim , "Towards measurement of confidence in safety cases," in *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2011.

[22] X. Zhao, D. Zhang, M. Lu and F. Zeng, "A new approach to assessment of confidence in assurance cases," in *Computer Safety, Reliability, and Security*, 2012.

[23] G. Shafer, A mathematical theory of evidence, Princeton: Princeton university press, 1976.

[24] J. Guiochet, Q. A. Do Hoang and M. Kaaniche, "A model for safety case confidence assessment," in *International Conference on Computer Safety, Reliability, and Security*, 2015.

[25] R. Wang, J. Guiochet and G. Motet, "Confidence Assessment Framework for Safety Arguments," in *Proc. of SafeComp'17*, Trento, Italy, 2017.

[26] L. Duan, S. Rayadurgam, M. Heimdahl, O. Sokolsky and I. Lee, "Representation of Confidence in Assurance Cases Using the Beta Distribution," in *IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*, 2016.

[27] P. Rook, "Controlling software projects," *Software Engineering Journal*, vol. 1, no. 1, pp. 7-16, 186.

[28] P. Graydon, I. Habli and R. Hawkins, "Arguing Conformance," *IEEE Software*, vol. 29, no. 3, pp. 50-57, 2012.

[29] O. M. Group, *UML Infrastructure Specification, v2. 4.1*, 2011.

[30] O. M. Group, *Object Constraint Language (OCL) Version 2.4*, 2014.

[31] C. Larman, Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Interative Development (3rd Edition), Prentice Hall, 2004.

[32] [Online]. Available: http://www.eclipse.org/atl/.

[33] R. Panesar-Walawege, M. Sabetzadeh and B. L., "Supporting the verification of compliance to safety standards via model-driven engineering: approach, too-support and empirical validation," *Information and Software Technology*, vol. 55, pp. 836-864, 2013.

[34] I. Eclipse, "Model Development Tools (MDT)," [Online]. Available: https://www.eclipse.org/modeling/mdt/?project=uml2.

[35] E. McKean, The new oxford American dictionary vol 2, New York: Oxford University Press, 2005.

[36] E. W. Denney, G. J. Pai and J. M. Pohl, "Automating the generation of heterogeneous aviation safety cases," 2011.

[37] Goal Structuring Notation Working Group, "GSN Metamodel," 2014.

[38] OMG, 2013. [Online]. Available: http://www.omg.org/spec/SACM/.

[39] D. Steinberg, F. Budinsky, E. Merks and M. Paternostro, EMF: eclipse modeling framework, Pearson Education, 2008.

[40] Astah, "Astah GSN Editor Overview," [Online]. Available: http://astah.net/editions/gsn.

[41] US Food and Drug Administration (FDA), "Guidance for Industry and FDA Staff-Total Product Life Cycle: Infusion Pump─Premarket Notification[510 (k)] Submissions.," 2010.

[42] C. B. Weinstock, J. Goodenough and A. Z. Klein, "Measuring assurance case confidence using Baconian probabilities," in *1st International Workshop on Assurance Cases for Software-Intensive Systems (ASSURE)*, 2013.

[43] F. Zeng, M. Lu and D. Zhong, "USING DS EVIDENCE THEORY TO EVALUATION OF CONFIDENCE IN SAFETY CASE," *Theoretical & Applied Information Technology*, vol. 47, no. 1, 2013.

[44] J. Goodenough, C. B. Weinstock and A. Z. Klein, "Toward a theory of assurance case confidence," CMU/Software Engineering Institute Technical Report CMU/SEI-2012-TR-002, 2012.

[45] R. Wang, J. Guiochet and G. Motet, "Confidence Assessment Framework for Safety Arguments," in *SafeComp* , Trento, Italy, 2017.

[46] R. Wang , J. Guiochet , G. Motet and W. Schön, "D-S Theory for Argument Confidence Assessment," in *Belief Functions: Theory and Applications*, 2016, pp. 190-200.

[47] G. Tsatsaronis and V. Panagiotopoulou, "A Generalized Vector Space Model for Text Retrieval Based on Semantic Relatedness," in *Proc. of EACL'09*, 2009.

[48] K. H. Gross, A. W. Fifarek and J. A. Hoffman, "Incremental Formal Methods Based Design Approach Demonstrated on a Coupled Tanks Control System," in *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*, 2016.

[49] AFRL-VVCAS, "Cooling Tank Challenge Problem CONOPS/High-Level Requirements," AFRL/88ABW-2017-0433, 2017.

[50] A. Fifarek, "Requirement Analysis using the SpeAR Framework," AFRL/88ABW-2015-2814, 2015.

[51] SoI Assurance Argument Group, Safe and Secure Systems and Software Symposium (S5), 2017.

[52] B. W. Matthews, "Comparison of the predicted and observed secondary structure of T4 phage lysozyme," *Biochimica et Biophysica Acta (BBA) - Protein Structure*, vol. 405, no. 2, pp. 44-451, 1975.

[53] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson and Y. Wang, "UPPAAL — a tool suite for automatic verification of real-time systems," in *Proc. of the International Hybrid Systems Workshop*, 1995.