RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

Université de Toulouse

Doctorat de l'Université de Toulouse

préparé à l'Université Toulouse - Jean Jaurès

Assurer la sécurité de l'architecture logicielle en utilisant des modèles d'argumentation: pattrons, templates, et outils

---

Thèse présentée et soutenue, le 13 décembre 2024 par

Marwa ZEROUAL

École doctorale
EDMITT - Ecole Doctorale Mathématiques, Informatique et Télécommunications de Toulouse

Spécialité
Informatique et Télécommunications

Unité de recherche
IRIT : Institut de Recherche en Informatique de Toulouse

Thèse dirigée par
Brahim HAMID

Composition du jury
M. Jean Michel BRUEL, Président, Université Toulouse - Jean Jaurès
Mme Régine LALEAU, Rapporteure, Université Paris-Est Créteil
Mme Barbara GALLINA, Rapporteure, Mälardalen University
M. Jason JASKOLKA, Examinateur, Université de Carleton
M. Simos GERASIMOU, Examinateur, University of York
M. Brahim HAMID, Directeur de thèse, Université Toulouse - Jean Jaurès
Mme Morayo ADEDJOUMA, Co-directrice de thèse, CEA Saclay

Dedicated to those I hold dearest: my parents. And to those they cherish most, the Zeroual juniors: Mohamed Bahaa Eddine, Sanad Abdessamed, and Noursine

Acknowledgements

With these lines, I would like to acknowledge all who became part of my doctorate journey.

First of all, I want to thank my thesis co-director, Doctor Morayo Adedjouma, who believed in me and offered me this opportunity to pursue this work in very welcoming conditions.

Then, I am very grateful to Professor Brahim HAMID who supervised my thesis in an impeccable manner. He always knew how to put me on the right track and boost my self-confidence. He taught me that there is always a more efficient way to accomplish tasks that initially seemed difficult to me.

This endeavor would not have been possible without our collaboration with Professor Jason JASKOLKA. He always had relevant answers to my questions and critical comments on my work, seeking for improvements. He taught me that enjoying the research makes it less of a burden and more of a passion.

I also want to thank all the members of the jury for their time and participation in my defense: Professor Régine LALLEAU, Professor Barbara GALLINA, Professor Jean Michel BRUEL, and Professor Simos GERASIMOU. Their questions and comments during the defense process turned out to be a chance for several perspectives.

I would then thank my colleagues from the LSEA team from the CEA-List and the ARGOS team from the IRIT-UT2 for all the stimulating discussions and exchanges we have shared. A special thanks to Doctor Megha QUAMARA who instilled in me the passion for perfection in work.

I will always be grateful to my friends for being all ears to all my problems.

Finally, my accomplishments were not mine alone. Nothing could be further from the truth. I have been encouraged, sustained, inspired, and tolerated by the greatest family anyone ever had.

Abstract

Context Nowadays, as systems grow larger and more complex, they invariably become more susceptible to an array of unforeseen security vulnerabilities. Therefore, security should be considered at all stages of their development. Thus, there is an ever-growing need to assure the security of critical software-dependent systems, and the information that they use, store, and communicate, in the face of cyber-attacks and failures. 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. The security (assurance) case assures the system users and regulators that the systems and their data are secure.

Problem It is often difficult, costly, and time-consuming to gather sufficient evidence supporting assurance claims about the security of a system after it has already been built and deployed. Thus, it is essential to incorporate security evaluation and assurance into the system development lifecycle. However, more explicit modeling techniques must be used to encourage the formal reasoning of assurance and security in the early design stages. Moreover, methodological support in this regard must be included, consolidating tools and techniques from the system, security, and assurance disciplines.

Contributions Considering the problems above, we propose an approach and tooled framework to better support and ease the security assurance process. Accordingly, the specific contributions of this work are many-fold:

- C.1) a method to build a security case modeling framework supporting the analysis of the security argument models.
- C.2) a modeling language and a design framework for modeling security cases of software architectures
- C.3) methods that reduce the cost of creating security cases by introducing security reusable argument patterns and templates.

- C.4) a catalogue of reusable security argument models (patterns and templates)
- C.5) an operational toolchain support for facilitating different phases of the approach.

Methodology The approach associates Model-Driven Engineering techniques to define a modeling language of security cases of software architectures. The results are provided as two complementary artifacts: (*a*) a modeling process of reusable formal model libraries for the security assurance of software architectures by security, process, and assurance experts; and (*b*) an application process of secure architectural assurance by an architect reusing the libraries specified in the process (*a*), within the validation of the assurance expert. Process (*a*) includes the following activities: (1) the definition of a modeling language for the security assurance cases; (2) the definition and formalization of the assurance concepts: claims, arguments, and evidence; and (3) the definition of reusable security argument models. Process (*b*) includes the following activities: (1) the security evaluation and measurement of a concrete architecture model to generate the artifacts and data required for the creation of a security case; (2) the selection of reusable argument models that fit with the security claims and the evidence provided by the evaluation activities; and (3) the generation of security case of the software architecture model.

Validation and illustration The specific security case modeling language is implemented using meta-models for the abstract syntax, and textual grammars for the concrete syntax. It provides a standardized modeling environment, relying upon the existing Unified Modeling Language. Furthermore, a specification of assurance semantic rules is proposed using a suitable formal language with automated tool support, namely OCL. As part of the assistance to the software architect, we propose a toolchain comprising Eclipse as a modeling framework and its OCL plug-in as a formal tool for semantic verification. The approach illustration and assessment are conducted via an autonomous drone (ACAS Xu) use case in the aeronautical domain, targeting a security-critical avoidance collision scenario.

Table of contents

Table of contents vii

List of figures xiii

List of tables xv

I Introduction 1

1 Introduction 3

1.1 Context . . . 3
1.2 Problem statement . . . 5
1.3 Contributions . . . 6
1.4 Publications . . . 7
1.5 Technical Framework . . . 10
1.5.1 Overview of Assurance Cases . . . 10
1.5.1.1 Terminology for Assurance . . . 10
1.5.1.2 Notations . . . 11
1.5.2 Software security engineering . . . 14
1.5.2.1 Incorporating security in software engineering . . . 15
1.5.2.2 Security threat Modeling . . . 15
1.5.2.3 Formal Verification of security requirements . . . 16
1.5.3 Model-Based Engineering (MBE) . . . 17
1.5.3.1 Model-Driven Engineering (MDE) . . . 18
1.5.3.2 Domain Specific Modeling Language (DSML) . . . 19
1.5.4 Development environment . . . 20
1.6 Thesis outline . . . 21

II Related Work 23

2 Related works: Approaches to creating Security Assurance case 25

2.1 Introduction . . . 25
2.2 Materials and Method . . . 26
2.3 Literature exploration . . . 28
2.4 Assessment and Key Takeaways . . . 35
2.4.1 Key finding . . . 35
2.4.2 Lacks in research . . . 37
2.5 Conclusion . . . 38

III Contribution 41

3 Approach 43

3.1 Introduction . . . 43
3.2 Need for Security Assurance – The Stakeholder's Perspective . . . 44
3.3 Method to Build the Security Assurance Modeling Framework . . . 45
3.3.1 Modeling framework development process . . . 46
3.3.2 Application development process . . . 47
3.4 Supporting the approach within SDLC . . . 47
3.5 Use Case: Autonomous Avoidance Detection System . . . 49
3.6 Research methodology . . . 51
3.7 Conceptual model . . . 51
3.8 Conclusion . . . 54

4 Security Cases Modeling Language (SCML) 55

4.1 Introduction . . . 55
4.2 Desired features of the language . . . 56
4.3 Abstract Syntax . . . 57
4.3.1 Assurance Package . . . 58
4.3.2 Process Package . . . 61
4.3.3 Security Package . . . 63
4.3.4 System Package . . . 64
4.3.5 Relationships between the different packages . . . 66
4.4 Semantics . . . 67
4.4.1 Static semantics . . . 67

4.4.2 Dynamic semantics 68
4.5 Concrete Syntax 69
4.6 Conclusion 71

5 Methodologies and Tools for the creation of Security Cases 73
5.1 Introduction 74
5.2 Inputs for the methodologies 75
5.3 Creating Security Cases from Scratch 76
5.3.1 Security case definition 77
5.3.2 Tool support 80
5.3.2.1 Modeling Framework Block 80
5.3.2.2 Application Development Block for Reuse 81
5.4 Creating Security Cases using Argument patterns 81
5.4.1 Argument Pattern definition 82
5.4.2 Pattern application 85
5.4.3 Tool support 88
5.5 Creating Security Cases using Argument templates 90
5.5.1 Argument Template definition 90
5.5.2 Template application 91
5.5.3 Tool support 92
5.6 Conclusion 93

6 Catalogue of reusable security argument patterns and templates 95
6.1 Introduction 95
6.2 Security argument patterns 96
6.2.1 Secure software architecture argument pattern 96
6.2.2 Argument pattern for threat modeling using STRIDE 98
6.2.3 Argument pattern for threat modeling using PASTA 101
6.2.4 Argument pattern for model checking 104
6.2.5 Argument pattern for well-formedness of the model 106
6.2.6 Summary 109
6.3 Reusable argument templates 110
6.3.1 Template for SCADA 110
6.3.1.1 Introduction to SCADA systems 110
6.3.1.2 Reference architecture for SCADA system 111
6.3.1.3 Securing a SCADA system 112

6.3.1.4 Security argument template for SCADA system 112
6.3.2 Template for MLBS 120
6.3.2.1 Introduction to MLBS systems 120
6.3.2.2 Reference architecture for MLBS system 121
6.3.2.3 Securing a MLBS system 122
6.3.2.4 Security argument template for MLBS system 124
6.3.3 Summary 128
6.4 Conclusion 129

7 Evaluation of the contributions 131
7.1 Introduction 131
7.2 Case study 132
7.2.1 Expressing the architecture of the ACAS Xu 132
7.2.2 Securing the software architecture of ACAS Xu 133
7.3 Definition of the reference data 135
7.3.1 SecureArchitecture reference data 136
7.3.2 Formal Verification reference data 137
7.3.3 ML formal verification reference data 140
7.3.4 ML monitoring reference data 142
7.4 Security case for ACAS Xu 145
7.5 Discussions 148
7.5.1 Key features of the approach 148
7.5.2 Applications of the proposed approach 150
7.5.3 Generalization of the proposed approach 151
7.6 Conclusion 151

IV Conclusion 153

8 Conclusion & future works 155
8.1 Summary and contributions 155
8.2 Limitations and future works 157
8.3 Perspectives 158

Bibliography 161

Appendices 173

A Security Engineering Activities 177
A.1 Threat modeling . . . 177
A.1.1 STRIDE . . . 177
A.1.2 P.A.S.T.A. . . . 178
A.2 Formal methods . . . 179
A.3 Evaluation Assurance Level . . . 180

B Concrete Syntaxes 181
B.1 Reference Data . . . 181
B.2 Process Package . . . 181
B.3 System Package . . . 182
B.4 Security Package . . . 182
B.5 Security case for the toy example . . . 182

C Catalogue of argument patterns 183
C.1 Graphical presentation of the patterns presented in the manuscript . . . 183
C.1.1 SecureArchitecture argument pattern . . . 183
C.1.2 STRIDEThreatModeling argument pattern . . . 183
C.1.3 PASTAThreatModeling argument pattern . . . 183
C.1.4 ModelChecking argument pattern . . . 183
C.1.5 ModelWellFormedness argument pattern . . . 183
C.1.6 FormalVerification argument pattern . . . 183
C.2 Argument patterns from our previous works . . . 183
C.2.1 Threat identification argument pattern . . . 184
C.2.2 Threat mitigation argument pattern . . . 190
C.2.3 Alloy Model well-fromedness argument pattern . . . 190
C.2.4 DNN secure development argument pattern . . . 193
C.2.5 DNN security requirements satisfaction argument pattern . . . 196

D Deep Neural Networks 201
D.1 Overview of DNN . . . 201
D.2 DNN LifeCycle . . . 201
D.2.1 Requirements engineering stage . . . 202
D.2.2 Data-oriented stage . . . 203
D.2.3 Model oriented process . . . 203

D.2.4 DNN operational process 204
D.3 DNN from the case study 204
D.4 Security verification of DNN 204
D.4.1 Test-based verification 204
D.4.2 Formal verification 205
D.4.3 Monitoring 206

Glossary **209**

List of figures

1.1 CAE main elements 12
1.2 GSN main elements and relationships 13
1.3 Components of SACM, from [124] 14
1.4 Modeling pyramid of the OMG 18

3.1 Stakeholders involved in the approach 45
3.2 Method to build security assurance framework 46
3.3 Integration of the approach in the development model 48
3.4 Architecture of ACAS Xu 50
3.5 Use Case Scenario: Collision caused by adversarial examples. Revised from [120] 51
3.6 Methodological framework 52
3.7 Conceptual model for the security assurance modeling language 53

4.1 SCML: Assurance Package 58
4.2 SCML: Process Package 61
4.3 SCML: Security Package 63
4.4 SCML: System Package 64
4.5 SCML: relationships between the packages 66
4.6 SCML structure adapted from [25] 71

5.1 From Scratch methodology overview 77
5.2 Tool support architecture and artifacts used for the scratch methodology . . 80
5.3 Pattern definition process 82
5.4 Pattern application methodology overview 86
5.5 Tool support architecture and artifacts used for creating and applying patterns 88
5.6 Template definition process 91

5.7 Template application methodology overview 92
5.8 Tool support architecture and artifacts used for creating and applying templates 94

6.1 A typical architecture of SCADA systems adopted from [111] 111
6.2 Patterns diagram for the SCADA Template 120
6.3 A typical architecture of MLBS systems 121
6.4 Patterns diagram for the MLBS template 128

7.1 Patterns diagram with the reference data for the MLBS template 144

C.1 Secure architecture argument pattern 184
C.2 STRIDE threat modeling argument pattern 185
C.3 PASTA threat modeling argument pattern 186
C.4 Model checking argument pattern 187
C.5 Model well-formedness argument pattern 188
C.6 Formal verification argument pattern 189
C.7 Argument pattern for threats identification 191
C.8 Argument pattern for the protection against threats 192
C.9 Pattern for the well-definedness of the system model 192
C.10 DNN Secure development argument pattern 195
C.11 Security requirement satisfaction argument pattern 198

D.1 Deep Neural Network Life cycle revised from [10] 202
D.2 ACAS Xu neural network 205

List of tables

1.1 Threat modeling methods . . . 16
1.2 Formal verification methods . . . 17

2.1 Summary of Approaches for Model-based Security cases Development . . . 31
2.2 Summary of Approaches for Standard-based Security cases Development . . . 32
2.3 Summary of Approaches for Asset-based Security cases Development . . . 33
2.4 Summary of Approaches for Formal-based Security cases Development . . . 35

6.1 Comparative summary of the argument patterns . . . 109
6.2 SCADA System Security: Threats and Security Patterns . . . 113
6.3 SCADA System Security: Threats, Requirements, and Security Mechanisms . . . 114
6.4 MLBS Security: Threats and Security Mechanisms . . . 123
6.5 Comparative summary of the argument templates . . . 129

7.1 ACAS Xu Security: Threats, Requirements, Security Patterns and Mechanisms . . . 135
7.2 Our result: Model-based approach for Security Case Development. . . . 150

8.1 Summary of the Thesis Contributions. . . . 156

List of listings

4.1 OCL constraint for Rule 17 . . . 69
4.2 Excerpt of the textual grammar of SCML . . . 70
5.1 Secure Architecture of a toy example . . . 75
5.2 Threat mitigation activity . . . 75
6.6 Reference architecture of SCADA system . . . 111
6.7 Secure reference architecture of SCADA system . . . 112
6.9 Reference architecture of MLBS system . . . 121
6.10 Secure reference architecture of MLBS system . . . 122
7.1 Secure Architecture of ACAS Xu system . . . 133
7.4 Theorem Proving Activity . . . 137
7.8 Interval Analysis Activity . . . 141
7.10 Monitoring Activity . . . 143
B.1 Excerpt of the textual syntax of SCML (Reference Data) . . . 181
B.2 Excerpt of the textual syntax of SCML (Process package) . . . 181
B.3 Excerpt of the textual syntax of SCML (Process package) . . . 182
B.4 Excerpt of the textual syntax of SCML (Security package) . . . 182

Part I

Introduction

Chapter 1

Introduction

Contents

1.1 Context **3**
1.2 Problem statement **5**
1.3 Contributions **6**
1.4 Publications **7**
1.5 Technical Framework **10**
1.5.1 Overview of Assurance Cases 10
1.5.2 Software security engineering 14
1.5.3 Model-Based Engineering (MBE) 17
1.5.4 Development environment 20
1.6 Thesis outline **21**

The structure of the introduction chapter is organized as follows. Section 1.1 introduces the context of the thesis. Section 1.2 states the proposed research problem. Section 1.3 summarises the contributions of this thesis. Section 1.4 notes the publications related to the work presented in this thesis. Section 1.5 presents elements of the formal and technical frameworks and some tools we use in the context of our research. Finally, Section 1.6 outlines the structure of the remainder of this thesis.

1.1 Context

Nowadays, the software systems deployed in critical domains, such as national security, financial, transportation, communication, and healthcare domains, are increasingly grow-

ing to become very large, complex, and connected. They have begun operating in open environments, which inevitably leaves them susceptible to various threats where failures could lead to harm, i.e., death, injury, or damage to property or the environment in their system context [52]. Security and safety present open challenges: they are complex and often subjective. Notably, the security is worsening because of the unpredictable nature of attackers' capabilities and knowledge. New insights about the system or its environment can emerge at any moment, requiring a reassessment of the system's safety and security measures [7]. Developing such critical systems demands robust evidence to consistently meet essential properties, including security, safety, and reliability.

Software assurance is often demonstrated by compliance with national or international security standards (e.g., ISO 26262 [2], ISOSAE 2143 [4], EN 50128 [20]). Compliance with standards provides assurance that the system behaves as it is intended to behave. Some standards called prescriptive standards, tightly depict processes and activities that must be followed to develop secure systems. In contrast, goal-based standards depict what developers and operators must achieve without specifying what and how to do it. Consequently, compliance with these standards can only be demonstrated by providing arguments. The arguments show how the evidence generated from the security engineering activities supports the claims about software security.

These arguments (together with supporting evidence) are typically called an 'assurance case'. In other words, the assurance case demonstrates why the results of the low-level analysis activities we have carried out (i.e., evidence) support and justify the high-level claims about the software (i.e., the top claim that we are interested in). This demonstration is made explicit using the assurance cases. Thus, different stakeholders can review, criticize, and learn from it (for example, if they want to change the system, they can use the assurance case to assess some of the impacts). Besides the argument and evidence, the assurance case captures assumptions about a system, viz. the domain application, a description of the environment, the development technologies, and how we will use that system.

When the assurance case claims about the security, it is called a *security case*. Security cases are formally mandated either by a certification process or by accepted sound practice principles in several security-critical industries. Typically, the security assurance of critical software-dependent systems requires providing early evidence of mitigating security risks, attacks, and vulnerabilities. Gathering sufficient evidence to support claims about system security is often complicated and time-consuming. It becomes more costly after the system has already been built and deployed [125]. Thus, there is a need to incorporate techniques for software evaluation and assurance into the system development lifecycle SDLC. An

argument for the security of a system should be developed in parallel with the system it supports [61].

The system architecture is one of the first artifacts produced by the development process and includes many design choices that should be reflected in the assurance case. Therefore, we have to understand which elements of the system architecture are important for the assurance case. The security of the software architecture in the early lifecycle stages has not yet been fully investigated. Nevertheless, to the best of our knowledge, the state-of-the-art approaches have not addressed the incorporation the security assurance cases in the software engineering process regarding the software architecture models.

1.2 Problem statement

Based on the previous discussion, we specify our general research problem from the lack of methodological tool support for security assurance cases. Our overall research problem is: ***How to define and assess a framework for generating security cases using reusable models?***

Specifically, we aim to address the following methodological issues in this regard:

- *Problem* **P1:** Despite the existing methodologies, techniques, and tools in safety assurance, a lack of methodological support exists for security assurance during the early design stages of the system software engineering process.
- *Problem* **P2:** Regarding software architecture modeling for assurance, there is a trend toward extending modeling languages (e.g., SysML) to better and explicitly support the concepts and needs of assurance standards. However, there is a lack of modeling language consolidating diverse knowledge from software architecture design, security engineering, and software assurance to effectively express the security assurance concepts in an integrated fashion.
- *Problem* **P3:** Creating security cases requires expertise from multiple domains: security, software engineering, and software assurance. It is a complex process that can be error-prone due to the delicateness of aligning security requirements with assurance objectives and ensuring comprehensive arguments supported with evidence. There is a need for means and rules that help the security case creators make security cases with strong arguments.
- *Problem* **P4:** Creating security cases takes a lot of time, and developers need to reduce the cost of producing secure systems. At the same time, they must also

lower the cost and risk associated with security vulnerabilities for both developers and end users. There is a lack of means that foster reusability and thus reduce the costs associated with the security assurance activity.

- *Problem* **P5:** The current landscape of security assurance case development is equally complex when approached from a standalone perspective. Existing works provide limited guidance for non-expert engineers, such as system architects and analysts involved in the security engineering process. These stakeholders may lack the expertise to effectively create and integrate comprehensive security assurance cases. This challenge is compounded by the lack of automated tool support for the systematic creation and validation of security assurance cases during the design phase, making it difficult to ensure that all necessary security requirements and mitigations are thoroughly addressed and documented.

1.3 Contributions

To address the problem above, this work strives for a joint design and assurance approach, along with a tool-supported framework, to better support and ease software security assurance. We adopt a systems perspective of security assurance, with software architecture serving as an essential basis [17]. Accordingly, the specific contributions of this thesis are five-fold:

- *Contribution* **C1:** The definition of a method to support the creation of an architecture-driven security assurance framework, i.e., an assurance that exploits and is linked to the software architecture to provide more structured evidence and arguments to show that the software architecture is secure and free of vulnerabilities.

- *Contribution* **C2:** The definition of a security case modeling language extended with some semantic rules that ensure the well-formedness of the security assurance models and make the assurance arguments less refutable. To this end, we propose a DSML called SCML defined within an abstract syntax, concrete syntax, and semantic rules.

- *Contribution* **C3:** The definition of methods that reduce the cost of creating security cases by introducing security argument patterns and templates. Moreover, we show how to define and apply them to create security cases for a specific system.

- *Contribution* **C4:** A catalogue of reusable argument models that contains reusable argument patterns for security engineering activities. Additionally, it presents reusable argument templates for reference architectures.

- *Contribution* **C5:** An operational tool-chain support integrating MDE and formal-based techniques for facilitating different phases of the proposed approach: modeling argument patterns and templates, semantic validation, and creating security cases from scratch or by applying reusable patterns and/or templates.

Thesis Approach Synopsis. Initially, we investigate the state-of-the-art approaches for incorporating security assurance in the software engineering process using security cases. Accordingly, features are defined to review these approaches from both standalone and co-engineering viewpoints. The lack and need for methodological support to an end-user, e.g., system designer, architect, and analyst, are thus identified. Accordingly, an approach is proposed in the context of the problems defined in Section 1.2. The specific contributions are made in the scope of the proposed approach. This is accompanied by tool-chain support that endows an environment for creating security cases while fostering the reuse of argument patterns and templates. Moreover, the approach's applicability is demonstrated with a use case from the aeronautics domain. Some of the outcomes of this work are published in [130, 132, 129, 131].

1.4 Publications

This section presents published/under-review papers related to the thesis. The publications are divided into two categories: (i) papers that are fundamental for the thesis contributions; (ii) papers that are related to the thesis. In the following, we list and provide concise overviews of those publications in chronological order.

Fundamental publications

- **Paper A.** [131] **Marwa Zeroual**, Brahim Hamid, Morayo Adedjouma, and Jason Jaskolka. 2024.
  Formal model-based argument patterns for security cases. In Proceedings of the 28th European Conference on Pattern Languages of Programs (EuroPLoP '23). Association for Computing Machinery, New York, NY, USA, Article 10, 1–12.
  doi: https://doi.org/10.1145/3628034.3628044
  Summary: This paper proposes an approach to constructing security assurance cases using formal methods. The proposed approach involves the following three steps: (1) decomposing security requirements and deriving security threats; (2) formalizing the system model and security threats; and (3) deriving the security

argument patterns supported by the results of the formal verification of the security requirements. We present the derived argument patterns using the Goal Structure Notation pattern notation. We apply the patterns to build security cases of an autonomous drone case study system.

- **Paper B.** [130] **Zeroual, M.**, Hamid, B., Adedjouma, M., Jaskolka, J. (2023). Constructing Security Cases Based on Formal Verification of Security Requirements in Alloy. In: Guiochet, J., Tonetta, S., Schoitsch, E., Roy, M., Bitsch, F. (eds) Computer Safety, Reliability, and Security. SAFECOMP 2023 Workshops. SAFECOMP 2023. Lecture Notes in Computer Science, vol 14182. Springer, Cham.
  doi: https://doi.org/10.1007/978-3-031-40953-0_2
  Summary: In this paper, we propose an approach that uses formal methods to construct security assurance cases. This approach takes a list of security requirements as input and generates security cases to assess their fulfillment. Furthermore, we define security argument patterns supported by the formal verification results presented using the GSN pattern notation. The overall approach is validated through a case study involving an autonomous drone.

- **Paper C.** [132] **Marwa Zeroual**, Brahim Hamid, Morayo Adedjouma, and Jason Jaskolka. 2024.
  Security Argument patterns for Deep Neural Network Development. In Proceedings of the 30th PATTERN LANGUAGES OF PROGRAMS CONFERENCE (PLOP '30). Association for Computing Machinery, New York, NY, USA, Article 10, 1–12.
  Summary: In this paper, we derive security argument patterns for DNN development, emphasizing verification of the fulfillment of security requirements. We present the argument patterns using the GSN pattern notation. We apply the patterns to build security cases of a DNN used in an autonomous drone case study system.

- **Paper D.** **Marwa Zeroual**, Brahim Hamid, Morayo Adedjouma, and Jason Jaskolka. 2024.
  A Tool-Supported Methodology for Creating Security Cases Using Argument Patterns. In Proceedings of the 13th International Conference on Model and Data Engineering - Naples - November 18 - 20, 2024 (MEDI 2024). **Summary:** In this paper, we propose to use Model Driven Engineering abstraction mechanisms to define a security assurance metamodel. Then, we apply formal techniques to specify expressions on models. These expressions reflect the semantic properties of

the metamodel. We also propose a process to define reusable argument patterns. The process is inspired by the metamodel and allows checking the formal semantic properties already defined. After building a catalogue of argument patterns, we show how they can be reused to construct security cases on a concrete system under assurance. To support the approach, we use the Eclipse Modeling Framework (EMF).

Publications related to the thesis

- **Paper F.** [132] **M. Zeroual**, B. Hamid, M. Adedjoumaa and J. Jaskolka, Towards logical specification of adversarial examples in machine learning, 2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Wuhan, China, 2022, pp. 1575-1580, doi: 10.1109/TrustCom56396.2022.00226.
  Summary: In this paper, we propose an approach to adversarial example threat specification and detection in component-based software architecture models. We use first-order and modal logic as an abstract and technology-independent formalism. The general idea of the approach is to specify the threat as property of a modeled system such that the violation of the specified property indicates the presence of the threat. We demonstrate the applicability of the method through a classifier used in a recommendation system.

- **Paper G.** **Marwa Zeroual**, Brahim Hamid, Morayo Adedjouma, and Jason Jaskolka.
  Formal security analysis of DNN architecture. In Proceedings of the 17TH INTERNATIONAL CONFERENCE ON VERIFICATION AND EVALUATION OF COMPUTER AND COMMUNICATION SYSTEMS (VECoS 2024) **Summary:** In this paper, we propose a formal approach to verify some security properties of neural networks. The overall approach goes through three steps: (1) specify security objectives as properties of a modeled neural network in a technology-independent specification language; (2) implement the developed model in a suitable tooled language for analysis; and (3) suggest a set of security requirements necessary to fulfill the targeted security objectives. We use first-order logic and modal logic as abstract and technology-independent formalism and Alloy as a tooled language. To validate our work, we explore a set of representative security objectives from the Confidentiality, Integrity, and Availability classification in a use case from the unmanned aircraft domain.

1.5 Technical Framework

This section presents the general concepts that are necessary to comprehend the problem tackled in this thesis and proposed contributions regarding methods and tools to build a framework for security assurance cases.

1.5.1 Overview of Assurance Cases

This section introduces the assurance case domain. In the first subsection, we set the terminology that will be followed in this thesis. Afterward, we present the different notations commonly used to document assurance cases.

1.5.1.1 Terminology for Assurance

This section provides definitions and explanations of key terms used in assurance cases.

Software assurance Software assurance is a process that combines all the practices that aim to ensure that the software behaves as specified and is free from intentional and unintentional failure modes. Among those practices, software engineers can use assurance cases.

Assurance case An assurance case is a structured argument that some system has some properties we desire, that it is safe, reliable, or secure against attack. Defining safety cases in particular, Kelly says "A safety case should communicate a clear, comprehensive and defensible argument that a system is acceptably safe to operate in a particular context" [66].

Security assurance case By generalizing the definition given by [66] to the security property, we obtain the following definition. A security case should communicate a clear, comprehensive, and defensible argument that a system is acceptably secure to operate in a particular system.

Assurance argument The argument in an assurance case demonstrates how a high-level claim (such as ”the system is sufficiently secure”) is supported by specific, detailed evidence related to lower-level properties. For instance, this might include statistical testing data that addresses a crucial security requirement for a particular software component.

Argument pattern An argument pattern is a reusable, generalized approach for addressing frequent issues encountered in the construction of argumentation frameworks, such as establishing the connection between a claim and supporting evidence within the Toulmin model [115].

Argument template An argument template serves as a reusable argument, offering a more concrete and detailed structure than argument patterns, which makes it easier to instantiate quickly. While it is more abstract and less complete than assurance cases, it is typically tailored to a specific domain or technology, providing a focused framework for argumentation [121].

1.5.1.2 Notations

Assurance cases must be documented clearly using arguments that should be structured to be comprehensible to all assurance-case stakeholders. It should also be clear how the evidence is asserted to support this argument. We can use textual or graphical notation. A free text notation [105] using natural language does not seem to be the most appropriate notation to address the above challenges due to the ambiguity or the poor structuring of the text, [55] presents techniques based on structured text for evidence specification: *Structured Prose, Argument Outline, Mathematical proof, LISP style.* In order to overcome the limitations of using free text in representing assurance cases, graphical representation approaches have been introduced. GSN and CAE are the most used ones, in addition to other notations presented in [75].

Claims-Argument-Evidence (CAE) CAE is a graphical notation for representing assurance cases by documenting a set of claims supported by the argument and the related evidence [105]. Figure 1.1 shows the core elements CAE notation.

- *Claim* is defined as a statement asserted within the argument that can be assessed to be true or false.
- *Argument* describes the argument approach presented in support of a claim.
- *Evidence* is described as a reference to the evidence being presented in support of the claim or argument.

Other than these elements, CAE presents different types of relationships such as:

- *Supports* is a relation between Argument and Claim.

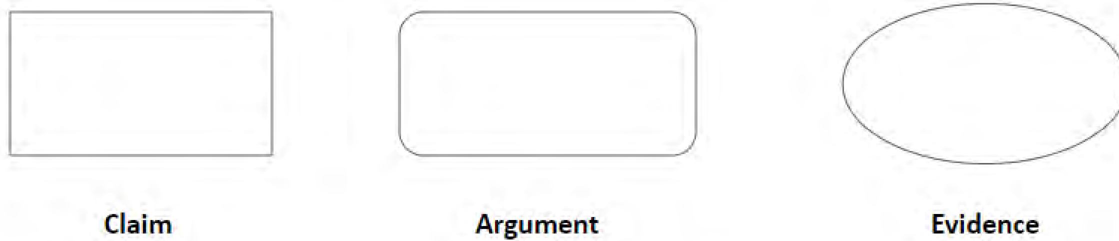


Figure 1.1: CAE main elements

- *Is a sub-claim of* is a relation between Sub-claim and Argument.
- *Is evidence for* is a relation between Evidence and Sub-claim.

Goal Structuring Notation (GSN) GSN [45] is a graphical argumentation notation that can be used to document explicitly the individual elements of any argument (claims, evidence and contextual information) and, perhaps more significantly, the relationships that exist between these elements (i.e. how claims are supported by other claims, and ultimately by evidence) and the context that is defined for the argument. GSN is a generic argument structuring language, which can be used to document arguments in any domain. In this document, we will consider it for assurance case domain.
Figure 1.2 shows the core elements and relationships of GSN notation:

- *Goal* presents a claim forming part of the argument.
- *Strategy* describes the inference that exists between a goal and its supporting goal(s).
- *Solution* presents a reference to an evidence item.
- *Context* presents a contextual artefact. This can be a reference to contextual information, or a statement.
- *Justification* rendered as an oval with the letter 'J' at the bottom-right, presents a statement of rationale.
- *Assumption* rendered as an oval with the letter 'A' at the bottom-right, presents an intentionally unsubstantiated statement.
- *In Context Of* declares a contextual relationship.

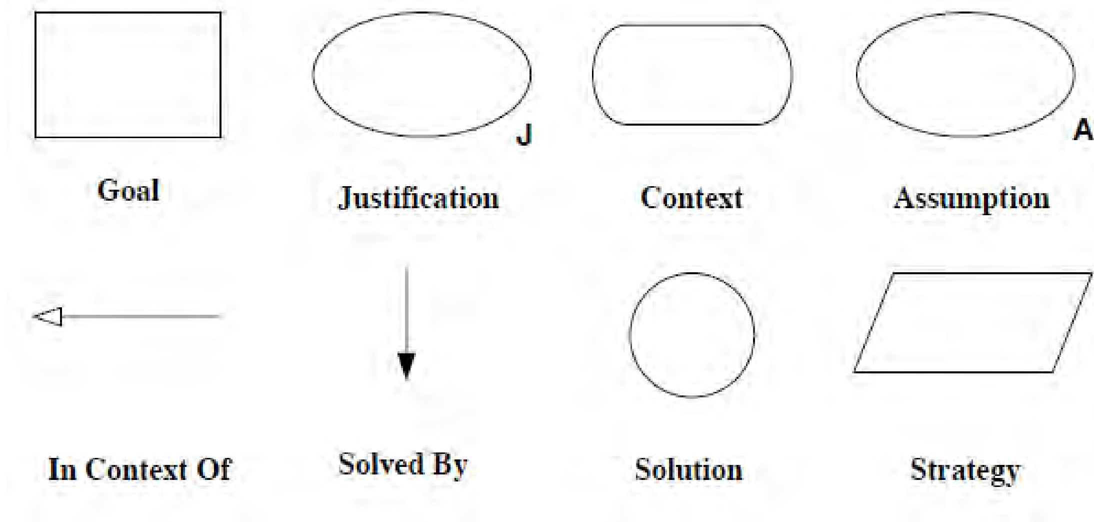


Figure 1.2: GSN main elements and relationships

- *Supported By* allows inferential or evidential relationships to be documented. Inferential relationships declare that there is an inference between goals in the argument. Evidential relationships declare the link between a goal and the evidence used to substantiate it.

Structured Assurance Case Metamodel(SACM) The Structured Assurance Case Metamodel (SACM) is standardized by the Object Management Group (OMG). The intention of the metamodel is to promote a model-based approach in the process of system assurance, which is currently a manual approach that produces artifacts (i.e. Assurance Cases) that are (mostly) not model-based, where higher-level operations (such as model validation) on these artifacts are not applicable. SACM is created to support the model-based paradigm with existing well-established graphical argumentation notations, the Goal Structuring Notation (GSN), and Claims-Arguments-Evidence (CAE)[124]. The SACM components are shown in Figure 1.3. There are five components in SACM[105], as shown in 1.3:

- *Base component* defines the fundamental elements of SACM, such as element names and descriptions.

- *AssuranceCase component* captures the concepts of Assurance Case in system assurance.

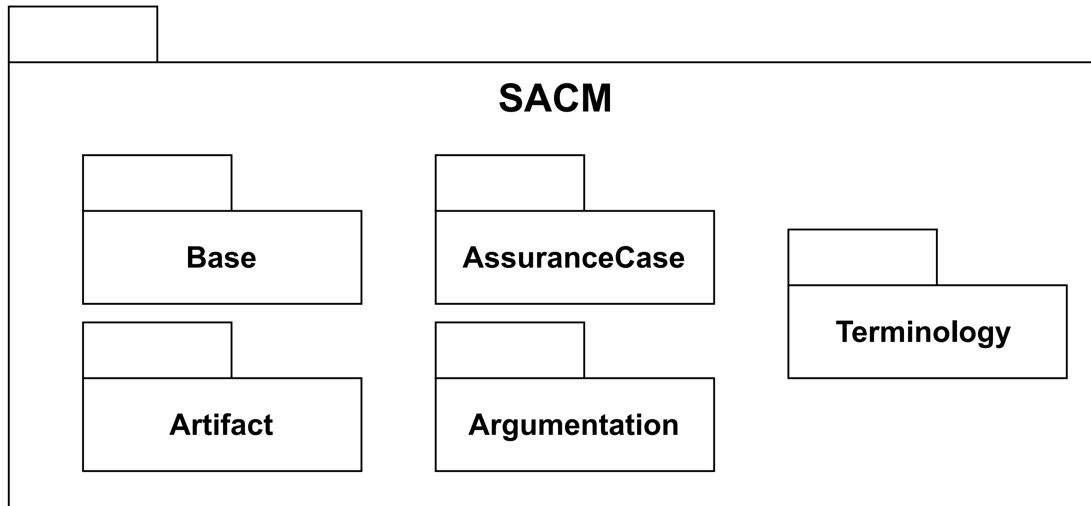


Figure 1.3: Components of SACM, from [124]

- *Artifact component* captures the concepts used in providing evidence for the arguments made for system properties.
- *Argumentation component* captures the concepts used in arguing system properties (such as safety and/or security)
- *Terminology component* captures the concepts used in expressing the arguments regarding system properties, such as Expressions.

1.5.2 Software security engineering

In system engineering, security may be compromised on several system layers. Usually, security is considered when design decisions are made leading to potentially conflicting implementations. The integration of security features requires the availability of a system architect, application domain-specific knowledge, and security expertise to manage the potential consequences of design decisions on the security of a system and on the rest of the architecture [46]. For instance, at the architectural level, security means having a mechanism (it may be a component or integrated into a component) that mitigates security issues.

Over time, significant research has been dedicated to developing specialized methodologies and techniques for secure software engineering. Thus, the aim is to reconcile the system and software engineering process and security process to consider security in

the first phase of the development cycle to avoid conflicts in the more advanced stages. Accordingly, security is not an add-on to the system but is directly integrated into the software development life cycle.

1.5.2.1 Incorporating security in software engineering

Experts agree that security is not just code. The most harmful vulnerabilities are those of the architecture, which will then be propagated in the other phases of the development cycle. Approaches considering earlier security aspects in software systems engineering are often considered part of the "Security by design" movement. In this section, we will analyze and investigate two activities for developing secure software architectures, namely threat modeling and security requirements formal verification. Moreover, we will investigate different methods used to realize the aforementioned activities to identify the commonalities and discuss the unique aspects of each approach, such as the tools used and the key assumptions made.

1.5.2.2 Security threat Modeling

This is the first step where potential threats to the system are identified and analyzed. This helps in understanding what kind of attacks the system might face and how to mitigate them. There are a lot of threat modeling techniques. Each technique was developed with distinct perspectives and priorities. Some methods focus on assets, others concentrate on attackers, and still others prioritize risks.

Major techniques used for threat modeling are the following:

- **STRIDE** is the most mature threat modeling method. The STRIDE model divides security threats into the following six categories. $Spoofing$ (impersonating a user or device), $Tampering$ (unauthorized modification of data), *Repudiation* (denying an action without proof), $Informationdisclosure$ (unauthorized access to information), $DenialofService$ (disrupting service availability), and $Elevationofpriveleges$ (gaining unauthorized access to higher-level functions). Tools used for assisting threat modeling include Microsoft Threat Modeling Tool[1] (TMT) [77], OWASP threat dragon[2] [36].

- **P.A.S.T.A** The **P**rocess for **A**ttack **S**imulation and **T**hreat **A**nalysis is a risk-centric method. PASTA consists of seven unique stages, each contributing detailed

[1]https://www.microsoft.com/en-us/securityengineering/sdl/threatmodeling
[2]https://www.threatdragon.com

Method	Tools	Key assumptions
STRIDE	TMT, OWASP	A Data Flow Diagram (DFD) is used to visualize the system + STRIDE classification is mapped to the identified threats.
PASTA	ARISTIUN	There is sufficient time to follow all the steps of the method

Table 1.1: Threat modeling methods

insights about the target object, its environment, potential threats, and associated risks [89]. By providing specific instructions for each stage, this methodology simplifies the complexity of threat modeling. The principal stages of PASTA method are: *Define the objectives*,*Define the technical scope*, *Decompose the application*, *Analyze the threats*, *Vulnerability analysis*, *Attacj analysis*,*Risk and impact analysis*. PASTA, by definition, exists as a framework of instructions, questionnaires and checklists. Some threat modeling tools need to be parameterized to support the PASTA method, e.g., ARISTIUN [80][3]

Table 1.1 summarizes the classifications of the threat modeling activities in the literature, highlighting the different tools, techniques and key assumptions.

More details about the activity is presented in Appendix A.

1.5.2.3 Formal Verification of security requirements

Formal verification is the use of mathematical techniques to ensure that a design conforms to some precisely expressed notion of functional correctness. Concretely, assume that you have (1) a model of a design, (2) some description of the environment that the design is supposed to operate in, and (3) some properties that the design is intended to fulfill.

We focus on formal methods (techniques and tools) that allow the creation of a system software architecture model and support the specification and verification of concerns and solutions properties of the system model based upon the architecture. Three major techniques are used for formal verifying security requirements:

- **Model checking** involves creating a finite model of the system and then systematically exploring all possible states to verify if certain properties hold. Tools used for assisting model checking include UPPAAL [42][4], NuSMV [127][5], Alloy [96][6].

[3]https://www.aristiun.com/
[4]https://uppaal.org/
[5]https://nusmv.fbk.eu/
[6]https://alloytools.org/

Method	Tools	Key assumptions
Model checking	Alloy	• System architecture model is well defined • The model checker capabilities are not exceeded by system size
Detective methods	Coq, Rodin (EventB), Isabelle	• The system can be executed symbolically with symbolic inputs, allowing the exploration of multiple paths simultaneously. • The resources are sufficient

Table 1.2: Formal verification methods

- **Deductive methods** use mathematical arguments to establish each property of a formal model. Proofs are normally constructed using a theorem-proving tool, e.g., Coq [21][7], Isabelle [65][8], Rodin [28][9], either automatically or in an interactive way
- **Abstract Interpretation** a kind of static analysis that automatically computes information about the program behavior without executing it. Tools that can be used include Splint [32][10], Astree [76][11].

Table 1.2 summarizes the classifications of the formal verification activities in the literature, highlighting the different tools, techniques and key assumptions.

More details about the activity is presented in Appendix A.

1.5.3 Model-Based Engineering (MBE)

Models are used to denote abstract representation of computing systems. In particular, we need models to represent software architecture and software platforms to test, to simulate and to validate the proposed solutions. Model-Based Engineering (MBE) based solutions seem very promising to meet the needs of secure and dependable applications development. The idea promoted by MBE is to use models at different levels of abstraction for developing systems. In other words, models provide input and output at all stages of system development until the final system itself is generated.

[7] https://coq.inria.fr/
[8] https://isabelle.in.tum.de/overview.html
[9] http://rodin.cs.ncl.ac.uk/
[10] https://splint.org/
[11] https://www.absint.com/astree/index.htm

Models

Models can be found in a variety of forms in a number of contexts in the area of software engineering. Their primary purpose is to adjust the representation of a complicated system by presenting essential information that may be of interest to the user. Models accomplish this by abstracting those details that are irrelevant to the purpose for which we want to use them. Because humans have limited observation and processing capabilities, we use models to describe and interact with our environment, even if we aren't aware of it. Sentences spoken or written in a given language might represent a variety of mental models. Mathematical equations are employed as models in physics, finance, and a variety of other areas to do calculations and reasoning. While the use of models in software engineering is not new, there has been a lot of attention in this topic in the previous decade since modeling can help create trustworthy software systems [81].

A metamodel is a model that defines an abstract representation of models. As shown in Figure 1.4, the Object Management Group (OMG) identified three degrees of abstraction on top of a reality layer. The Meta Object Facility (MOF) [87], an OMG standard that defines a collection of key concepts needed to construct models (and sufficient to define itself, i.e., MOF is a metamodel of itself), is at the summit of this pyramid. As a result, MOF (M3) can be used to develop modeling languages (M2), which can then be used to define models (M1), which are representations of real-world systems (M0).

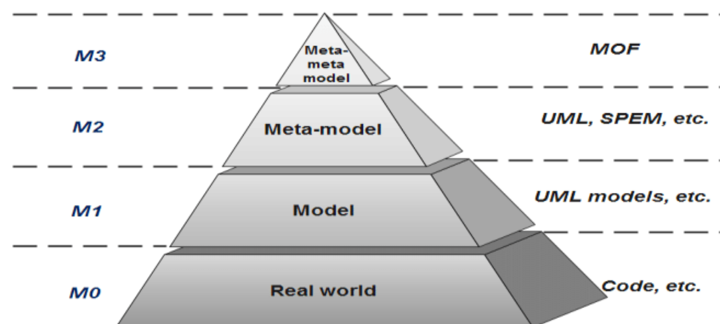


Figure 1.4: Modeling pyramid of the OMG

1.5.3.1 Model-Driven Engineering (MDE)

The concept of a model is becoming a major paradigm in software engineering. Its use represents a significant advance in terms of level of abstraction, continuity, generality, scalability, etc. Model-Driven Engineering (MDE) is a form of generative engineering [101], in which all or a part of an application is generated from models. MDE is a

promising approach since it offers tools to deal with the development of complex systems improving their quality and reducing their development life cycles. The development is based on model approaches, metamodeling, Model-To-Model transformations, development processes and execution platforms. The advantage of having an MDE process is that it clearly defines each step to be taken, forcing the developers to follow a defined methodology. MDE allows to increase software quality and to reduce the software systems development life cycle. Moreover, from a model it is possible to automatize steps by model refinements and generate code for all or parts of the application.

MDE provides a useful contribution for the design of trusted systems, since it bridges the gap between design issues and implementation concerns. It helps the designer to specify in a separate way non-functional requirements such as security and/or dependability needs at a higher level of abstraction. This allows implementation independent validation of models, generally considered as an important assurance step.

The development process cycles are mainly iterative, resulting in different levels of model refinement from analysis to design. There are implementation platforms that address these issues in a specific context (e.g., the MDA standard [13]), but in many other contexts, the links between models refined or processed to solve references (to non-existent elements, elements not referenced, created elements, etc.) are still solved in ad hoc manner, without adequate support from generic technologies.

1.5.3.2 Domain Specific Modeling Language (DSML)

In software engineering, Domain Specific Modeling (DSM) [37, 44] is a methodology that uses models to specify applications within a particular domain. Domain Specific Modeling Languages (DSMLs) are languages that are specifically tailored to the needs of a particular problem or application domain. Domain experts can understand, validate, modify, test, and sometimes even develop such languages. Domain Specific Modeling Languages (DSMLs) are frequently used in MDE [104].

A language is defined by an abstract syntax, a concrete syntax and the description of semantics [50, 37, 69]. The abstract syntax defines the concepts and their relationships which are often expressed by a metamodel. On the one hand, the concrete syntax defines the appearance of the language. A grammar or set of regular expressions is often used to design the concrete syntax. On the other hand, semantics defines the sense and meaning of the structure by defining sets of rules. Domain Specific Modeling (DSM) in software engineering is used as a methodology using models as first-class citizens to specify applications within a particular domain. The purpose of DSM is to raise the level of abstraction

by only using the concepts of the domain and hiding low level implementation details [44]. A Domain Specific Language (DSL) typically defines concepts and rules of the domain using a metamodel for the abstract syntax, and a concrete syntax (textual, tree-structured, tabular, diagrammatic, etc.). Domain Specific Language (DSL) allow specifying systems in a domain-friendly manner. Most metamodels and/or abstract syntaxes offer one or more concrete syntaxes to instantiate their concepts. The UML standard, for example, provides concrete syntaxes with diagrams for different viewpoints, in a graphical manner with icons and links. Other metamodels, and especially domain-specific modeling languages, often come with a textual syntax. As we shall see, processes in DSM reuse a lot of practices from MDE, for instance, metamodeling and transformation techniques.

1.5.4 Development environment

In this section, we will briefly introduce the technologies that we use in our development environment to support the approach presented in this thesis through tool support.

Eclipse Modeling Framework

There are several Domain Specific Modeling (DSM) environments, one of them being the open-source Eclipse Modeling Framework (EMF) [108]. Eclipse Modeling Framework (EMF) provides an implementation of Essential MOF (EMOF), a subset of Meta-Object Facility (MOF), called Ecore2. EMF offers a set of tools to specify metamodels in Ecore and to generate other representations of them, for instance Java. In our context, we use the Eclipse Modeling Framework. Note, however, that our vision is not limited to the EMF platform. Here, we outline the different Eclipse tools used in the development of the DSLs to support the modeling of the Security and Dependability (S&D) artifacts, the repository and its Application Programming Interfaces (APIs). Among the tools used here are cited:

- Eclipse is an open-source software project providing a highly integrated tool platform. The applications in Eclipse are implemented in Java and target many operating systems including Windows, Mac OSX, and Linux [108].

- EMF is a modeling framework and code generation facility for building applications based on a structured data model. In addition, EMF provides the foundation for interoperability with other EMF-based tools and applications [108].

- RCP plug-in allows developers to use the Eclipse platform to create flexible and extensible desktop applications upon a plug-in architecture [117].

- Xtext [12] is a programming language and domain-specific language development framework. With Xtext, you may use a powerful grammar language to define your language. Xtext provides a set of APIs and DSLs to develop such DSLs easily. It not only gives you a generated parser but supports the full stack of infrastructure that is needed, which includes a parser, linker, type checker, compiler, and editing support for Eclipse, any editor that supports the Language Server Protocol.

Object Constraint Language

The Unified Modeling Language (UML) is a graphical modeling language that is regarded as the de-facto standard for object-oriented modeling. OCL [95][12] is a textual language complementing UML that can be used for specification tasks that are difficult or impossible to accomplish with UML diagrams alone. OCL has a variety of applications at different modeling levels. The expressions of OCL constitute the core of the language. The evaluation of OCL expressions is free of side effects. OCL provides convenient means for navigating across associations and retrieving objects. For an OCL expression e, the expression $e.a$ denotes the value of the attribute a for the object e. In addition to user-defined classes and the primitive types Boolean, Integer, Real and String, OCL offers the collection types Set, Bag (i.e., multiset), Sequence and OrderedSet. Collections appear usually by evaluating attributes with a corresponding multiplicity, but OCL also features dedicated collection constructors. The OCL standard library provides numerous collection operations. These operations are accessed by means of the $->$ operator, e.g., $e->size()$ denotes the size of the collection-valued expression e. Special operations called iterators allow variables to refer to collection elements. A type of such iterators are comprehensions, which are named collect in OCL. The mathematical expression $x+1|x \in A$ would be written as $A->collect(x|x+1)$ in OCL. Variables in iterators can also be left implicit, e.g., $e->collect(a)$ denotes a collection of all values the attribute a can assume for any object in the collection e. The precise type of the resulting collection depends on the type of the source e.

1.6 Thesis outline

The outline of the dissertation is as follows. Chapter 2 presents the state of the art relevant to our research. Chapter 3 is dedicated to present a method for the creation of a security assurance case framework to assist software architects in the assurance process.

[12] https://www.omg.org/spec/OCL/

Chapter 4 illustrates a security assurance framework as a building block to create security assurance models. Chapter 5 presents methodologies used within the framework to create security cases from scratch or by reusing security argument models. Chapter 6 presents some reusable argument patterns and templates. An illustration of the use of the aforementioned framework within a case study and a discussion on the feasibility and the potential applications of the proposed approach, as well as the potential for its generalization and extension are presented in Chapter 7. Finally, Chapter 8 concludes the dissertation and proposes some future works and perspectives.

Part II

Related Work

Chapter 2

Related works: Approaches to creating Security Assurance case

Contents

2.1 Introduction **25**
2.2 Materials and Method **26**
2.3 Literature exploration **28**
2.4 Assessment and Key Takeaways **35**
2.4.1 Key finding 35
2.4.2 Lacks in research 37
2.5 Conclusion **38**

2.1 Introduction

Considering the context and problems stated in Chapter 1, we establish a systematic state-of-the-art in literature and industry practices. In particular, we are interested in understanding how current approaches integrate security assurance practices, specifically security cases, into the software development process. By “approach”, we mean a way or strategy covering entangled or correlated aspects like frameworks, methods, or implementation tools to address the mentioned problems. In this study, we relied upon the basic principles from [68] for carrying out a Systematic Literature Review SLR to explore, analyze, and interpret various contributions from the literature relevant to the security assurance domain. In addition to individually reporting the works, we characterize them

based on multi-attribute taxonomies relying upon the system, assurance, and security domain concepts. In the beginning, we position our context and requirements to the state-of-the-art. Then, an attempt is made to identify and assess concepts and methods as key takeaways to be used in the context of model-based approaches for the generation of security assurance cases

Overall, the chapter is organized as follows. In Section 2.2 we detail the methodology employed, including the search queries and selection criteria used to identify and evaluate the relevant research. In Section 2.3, we present the results of this systematic review. Then, Section 2.4 summarizes and synthesizes the key findings from the selected studies. Moreover, we highlight the identified gaps in the current body of knowledge, offering insights into areas where further research is needed to advance understanding in the field. Finally, we conclude the chapter in Section 2.5.

2.2 Materials and Method

To be precise with the scope of our investigation, we mainly focused on the approaches concerning the creation of security assurance cases. As mentioned, this work follows the guidelines proposed in [68] for conducting an SLR. In this case, the aim is to rely on a commonly accepted methodology for identifying and interpreting the existing literature relevant to security assurance while maintaining the transparency of the results. The methodology that this work endorsed broadly includes the following four phases: 1) research context-oriented formulation of the problems defining the scope of this study, which is already covered in previous Chapter 1, 2) exploration of the existing literature in the direction of the formulated research problems, 3) multi-attribute analysis of the identified literature in the context of the problems, and 4) discussion on the key findings based on the literature analysis, comprising existing lacks to be addressed. We sought to iteratively go through these different phases for this study to enable its thorough evaluation. The individual steps are documented in detail in the following paragraphs.

Literature exploration

We sought to identify the building blocks and associated literature to address the creation of security assurance cases. We adopted both automated and manual search processes for exploring the literature concerning conceptual and empirical contributions made by the research community relevant to this survey. We began with conducting an automated

search on existing scientific databases like ACM Digital Library [1], DBLP[2], and IEEE Xplore[3], using a set of search keywords constituting the following Boolean string:

"Security Assurance Case" **OR**
"Security" **AND** ("Assurance" **OR** "Assuring") **AND** ("Argument" **OR**
"Argumentation" **OR** "Evidence") **OR**
"Security Assurance" **AND** ("Argument Pattern" **OR** "Argument Template") .

We limited the search to the domains of computer science and engineering. Also, because of the high number of returned results, we decided to limit the included studies to the papers written in English, papers published between (2010 / 2023), and those published in a conference or a journal.

Furthermore, we developed and elucidated a multi-attribute taxonomy to analyze the obtained research contributions, comprising some novel and already used attributes in the existing literature. To accomplish this, we used the concepts across system, assurance, and security engineering domains to characterize the selected research contributions, as detailed in sub-section 2.3.

Positioning attributes

To address the problems mentioned in the previous chapter, we used the following list of attributes to characterize the selected approaches for the creation of security cases for prospecting the capabilities offered by them in the domain:

Approach type outlines the approach followed to create security cases. This classification is applied not only to security cases but also to other types of assurance cases. **Model-based approaches** are based upon the usage of software models to create security cases. **Standard-based approaches** allow the creation of security cases that comply with standards and regulations. **Asset-based approaches** or asset-driven approaches create security cases that have assets as drivers of the structure of the security arguments [82]. **Formal-based approaches** use formal methods to produce evidence elements and or to ensure that the confidence is met [99].

Life cycle Stage represents the target phase of the system engineering process to which the security case is applicable. The values that we considered include: *Requirement*

[1]https://dl.acm.org/
[2]https://dblp.org/
[3]https://ieeexplore.ieee.org/

(R), *Design* (D1), *Development* (D2), *Risk Analysis* (RA), *Operation* (O) or *Generic* (G) for all the phases. It is inspired by [93, 27].

Argumentation outlines which argumentation strategies are used to divide the goals and create the arguments.

Evidence highlights the collected evidence (if any) to support the security case.

Tool outlines the automated tool (if any) used to generate the security case. Tools may be accompanied by modeling language.

Use Case highlights the use case or case study (if any) for demonstrating the proposed solution and feasibility analysis.

Reusability outlines whether the approach fosters reuse and allows the definition of argument patterns and or templates.

Evaluation Outlines the means to interpret and assess the security assurance cases.

2.3 Literature exploration

Herein, we summarize the results of our study on the approaches categorized into the following categories: model-based, standard-based, asset-based, and formal-based.

Model-based security assurance case

The approach presented in [116] integrates security assurance creation with system development. The key idea of the methodology is to reflect the structure of the system in the assurance case throughout the development life-cycle phases: analysis, design, development, and test. At each phase, the approach relates security assurance case elements to the software development artifacts. The argumentation strategies go through security requirements, system goals, system view,s and models.
The work in [85] presents a layered assurance approach to creating security cases. It includes security assurance in three phases of the development life-cycle: (1) the safety and security requirements engineering phase, (2) the architecture phase, and (3) the implementation phase. The arguing strategies involve claiming safety requirements, security requirements, the interaction between them, and the trade-off satisfactory. For the supporting evidence, we find artifacts resulting from the safety and security analysis, documents about interactions between safety and security, and also documents capturing

trade-off analyses and decisions made. The approach is applied to a case study from the avionic domain where both safety and security are relevant.
The work [112] proposes a semi-automated approach for the creation of model-based safety and security assurance cases. The approach leverages the design decisions made early in the development, particularly, how the high-level safety and security goals are refined into more concrete countermeasures. The countermeasures are reported on patterns. For each countermeasure pattern, an argumentation fragment is defined to capture the design rationale behind the countermeasures pattern. Assurance cases are created by merging the argumentation fragments.
Another model-based approach for creating assurance cases is proposed in [53]. The assurance case meta-model is based on an SACM meta-model extended with GSN elements and References Roles. The Reference Roles are abstract entities that need to be instantiated on concrete systems. The approach allows co-evolution of assurance and development by using information extracted directly from design, analysis, and development models to create assurance cases. The approach treats safety and security cases as the same. More recent efforts about creating assurance case models using SACM meta-model were conducted in [124]. SACM meta-model is strengthened with features like modularity, counter-arguments, and traceability from Evidence to Artifact. The authors provide SACM-compliant meta-models for existing system assurance approaches (the Goal Structuring Notation and Claims-Arguments-Evidence) and the mapping from these models to SACM. SACM supports the definition of argument patterns. However, authors use argument patterns and templates interchangeably. There is a need for a clear distinction between the two reusable model patterns and templates

We provide a summary of the selected approaches in Table 2.1, based on the attributes mentioned in the previous sub-section 2.2.

Ref.	Stage	Argumentation	Evidence	Tool	Case study	Reuse.	Eval.
[85]	G	• Safety and security and their interaction. • Lifecycle phases • Correctness of the building process. • Protection from malicious changes.	• Safety and security analysis • Trade-off security and safety	Any tool that uses CAE notation	Gateway from avionic domain	The development of argument templates is mentioned as future work	No
[124]	D1 + O	• Security properties	• Design artifacts	ACME editor that uses SACM notation	the European Train Control Systems (ETCS)	Argument patterns	No
[53]	D1	• Stages of security engineering methodology • Software components • Failure modes	• Design models • Software models	GSN tree-based editor	Crypto controller system	Argument patterns	No
[116]	G	• Compliance with legislation • Software components • Threat analysis	No	No	European project PICOS	No	No

[112]	D1	• Security goals • Security countermeasures	No	GSN graphical editor running on the Eclipse platform.	No	No	No

Table 2.1: Summary of Approaches for Model-based Security cases Development

Standard-based security assurance cases

The proposed approach in [94], early, integrates the security assurance cases in the engineering processes by mapping the security case creation activities to a security engineering methodology. The security cases are generated based on the attack surfaces, the vulnerabilities, and the attack scenarios. The approach implies that the security cases comply with regulations and the internal needs of medical cyber-physical systems manufacturers. Systems cannot be built to eliminate safety risks but can recognize, resist and recover from attacks. System should be prepared for implementation and maintenance in the initial acquisition and design. To ensure successful organizational protection over time, assurance must be scheduled over the life cycle. Consequently, the work in [35] presents a framework for the creation of security cases for medical devices. Their framework incorporates multiple security documents, e.g., standards and best practices [59], to develop a security argumentation pattern.
Common Criteria (CC: equivalent to ISO/IEC15408) [118] specifies a framework for evaluating the reliability of the security assurance level defined by a system developer.
From another perspective, the use of a cybersecurity case to document compliance with cybersecurity goals is suggested in ISO/SAE 21434, but no further information is provided. As the safety case is already widely used for functional safety in the automotive industry [88], this provides a natural basis for developing an assurance case for cybersecurity. Therefore, the work in [24] proposes a novel cybersecurity case framework that adapts existing approaches from safety engineering, emphasizes the limitations of the analysis through eliminative argumentation, and merges in the attack-defense tree techniques used in cybersecurity engineering, to provide a better reflection of the some of the

uncertainties in the cybersecurity risk analysis.
We summarize the selected approaches in Table 2.2, based on the attributes mentioned in the previous sub-section 2.2.

Ref.	Stage	Argumentation	Evidence	Tool	Case study	Reuse.	Eval.
[94]	G	• Stages of security engineering methodology • Development lifecycle	No	No	Medical Cyber-Physical system	No	No
[35]	D2 + O	Risk management as described in ANSI/AAMI/ISO14971 for medical devices	• Results of risk control verification	GSN graphical-based editor	Medical devices	Argument pattern	No
[24]	R	• Risk-based argumentation outlined in ISO/SAE 21434	• Threat analysis results • Cybersecurity risk analysis	GSN graphical-based editor	Automated-driving systems	No	No

Table 2.2: Summary of Approaches for Standard-based Security cases Development

Asset-based security assurance case

The asset-based security case presented in [126] is a part of a layered security assurance case. The asset-driven security case argues through the enumeration of the threats to mitigate to protect critical assets. An Asset-driven Approach to Build Security Assurance Cases for Automotive Systems.
The work in [72] presents a complete approach to creating security cases. The first step entails the identification of assets through an asset inventory. Next, data flow diagrams are used to map the flow of data assets across all system components. After, security controls are identified, each tied to a component and a security goal. The last step constructs security cases. The assurance arguments argue about protecting the assets throughout

their life cycles by protecting the components that store, process, and transmit those assets.
The work in [83] proposes an approach called CASCADE that combines asset-based and standard-based security cases. Particularly, they investigated how an asset-based approach for the creation of security assurance cases can assist automotive companies to fulfill their needs to conform with the upcoming ISO/SAE-21434 standard [106]. First, they created a high-level structure of an asset-driven security assurance case, which included the identification of the assets, the tracing of such assets to system elements (e.g., processing, communication, and storage operations), and the identification of the relevant security goals for each asset. Second, we improved the structure of CASCADE to better align with the development activities at automotive companies and better cover their needs for security assurance cases, including the need to confront with ISO/SAE-21434.
We provide a summary of the selected approaches in Table 2.3, based on the attributes mentioned in the previous sub-section 2.2.

Ref.	Stage	Argumentation	Evidence	Tool	Case study	Reuse.	Eval.
[72]	R	• Sensitive assets • Security goals • Security activities	• Protective mechanisms • Auditing reports	No	No	No	No
[83]	R	• Sensitive assets • Security goals • Threat scenarios • Risk assessment • Security requirements	• Verification results • Test coverage reports	No	Road vehicle's headlamp from ISO/SAE-21434	Argument pattern	No
[126]	R	• Sensitive assets • Security threats • Security controls	No	No	Instant messaging software	Argument patterns	No

Table 2.3: Summary of Approaches for Asset-based Security cases Development

Formal-based security assurance case

The paper [22] addresses the problem of constructing an assurance case by presenting an approach to extracting information from a large set of documents. In the proposed approach, document retrieval and formal concept analysis techniques are systematically combined to assist users in exploring relevant information from huge data sets and understanding several concepts in such data sets and the relation among them. The approach proposes some criterias to evaluate the quality of the security case. The claim coverage indicates how much an AC is mentioning about a reference-issues-list. The argument coverage indicates the coverage of the argument and the evidence.
The work in [39] proposes to use Architecture Analysis and Design Language (AADL) and annex it with Resolute[4] language to specify the system architecture, safety rules, and security claims. The analysis leads to the generation of fragments of assurance cases. This approach avoids inconsistencies between a design and its assurance cases, thanks to automated model transformation. The tool also incorporates external analyses. Each time the Resolute is invoked, some properties, such as schedulability or resource allocation, are analyzed.
We provide a summary of the selected approaches in Table 2.4, based on the attributes mentioned in the previous sub-section 2.2.

Ref.	Stage	Argumentation	Evidence	Tool	Case study	Reuse.	Eval.
[22]	G	• Design analyses • Risk assessment • Tests and verification • Requirements from stakeholders.	• Design documents • Formal verification results	No	Bug tracking system: Moodle	No	Claim and argument coverage

[4]https://github.com/loonwerks/Resolute

[39]	D1	• Hazards enumeration • Hazards mitigation	• Testing results • Formal verification results	Resolute: a language and a tool	Unmanned Air Vehicles (UAV)	No	Logic based analysis

Table 2.4: Summary of Approaches for Formal-based Security cases Development

2.4 Assessment and Key Takeaways

In this section, we discuss the results obtained from the previous section for the key findings upon which we build our research and find the gaps in existing contributions that are yet to be addressed.

2.4.1 Key finding

Based on the characterization summary of the approaches presented in the previous section 2.3, we offer a concrete view of our findings regarding the selected attributes:

- *Approach type* The approaches based on the usage of models are dominant in the security context. Fewer contributions are based on standards, they ensure compliance with industry regulations and frameworks, particularly in sectors where adherence to standardized risk management is critical. Likewise, asset-based approaches focus on protecting critical assets by implementing specific security measures and controls. These approaches are integral to organizations prioritizing asset protection as a key aspect of their security strategy. Lastly, formal-based assurance is revolutionizing the domain of security assurance by leveraging the strengths of formal methods to create more rigorous and convincing arguments. Despite their potential, the adoption of formal-based approaches remains limited to fewer contributions, primarily due to the complexity and specialized knowledge required for their implementation.

- *Life-cycle stage:* Existing approaches span various stages of the software development lifecycle, ensuring comprehensive coverage across different phases. However, most of these approaches are concentrated in the early stages, such as requirements

analysis and design. This focus underscores the importance of addressing key aspects early in the process. Crucially, it is essential to consider security assurance before the development and deployment of the system, as establishing a strong security foundation during these initial stages can significantly reduce vulnerabilities and enhance the overall resilience of the software.

- *Argumentation:* A range of contributions argue through two views: (1) the security positive view where the concerns are specified as security goals, objectives, requirements, controls and countermeasures [72, 83, 112]; and, (2) the security negative view where the concerns are specified as security threats, failure modes, attacks and risks [116, 126]. Only the standard-based approaches indicate a strong emphasis on adhering to regulatory guidelines and structured risk-based argumentation. Moreover, asset-based approaches argues through strategies that focus on safeguarding critical assets and managing risks through well-defined security measures. Overall, the argumentation strategies vary across approaches, with each set tailored to address specific concerns, whether technical, regulatory, or asset-focused, reflecting the diverse needs of different domains and systems.

- *Evidence:* There is a diverse range of evidence types across approaches, highlighting the varying priorities and levels of rigor in ensuring system security. They predominantly rely on design-related artifacts, with evidence including safety and security analysis, design artifacts, and models. Moreover, formal model-based approaches employ robust evidence types, notably the formal verification results, demonstrating a strong commitment to rigorous validation and verification processes. However, some approaches lack specified evidence, which may suggest gaps in their adoption.

- *Tool:* Regarding the tool support and languages used to model the security cases, the studied approaches show a strong preference for tools supporting GSN notation. Fewer contributions discuss using specific notations (CAE, SACM, Resolute) for specific purposes. This distribution suggests a varied landscape where certain model-based approaches are well-supported by specific tools. In contrast, others either lack dedicated tools or do not specify them, potentially limiting their practical implementation.

- *Case Study:* Regarding the case study, most approaches are applied to a range of complex systems from different security-critical domains. Notably, standard-based approaches are prominently applied in medical cyber-physical systems where

security standardization is more mature compared to other security-critical domains [109]. Overall, the table highlights the broad applicability of these approaches across various industries, though some lack direct case study validation.

- *Reusability:* The studied approaches display a limited use of reusable argument models. Several approaches recognize the value of reusable security argument patterns. In contrast, the argument templates introduced within GSN have not been explored yet.

- *Evaluation:* It can be observed that most approaches (model-based, standard-based, and asset-based) lack evaluation criteria for the quality of the security case. This suggests a potential weakness in validating the effectiveness or reliability of these approaches. In contrast, formal model-based approaches show a more rigorous evaluation, with specific methods like claim coverage and argument coverage for one application and logic-based rules for another. This indicates that formal model-based approaches are more likely to construct a security case with accurate information

2.4.2 Lacks in research

Security assurance is a relatively new domain and requires a critical understanding of the needs, gaps, state-of-the-art, and practices [61]. Our study of the existing literature brings forward the following associated gaps structured across the modeling, coverage, and analysis, along with the tool support for their realization that must be fulfilled to address the idea of security assurance:

- *Modeling:* The use of Domain-Specific Modeling Languages (DSMLs) implemented as Unified Modeling Language (UML) or Systems Modeling Language (SysML) meta-models and profiles is widespread for modeling assurance cases. Some approaches, e.g., [124, 53], incorporate security properties. However, they adopt a fixed modeling view, not covering reusable assurance models, like patterns. In addition, existing approaches often provide a unified viewpoint regarding the modeling language to capture the assurance of domain-specific concerns. However, comprehending assurance, system engineering, software architecture, and security interplay requires maintaining individual concerns from all domains and analyzing them for potential conflicts.

- *Coverage:* Several contributions indicate a tendency towards covering the argumentation part more than the evidence part. Nonetheless, studies focusing only on one

part of the security cases have limited their applications. For example, if we take an approach that focuses only on the argumentation part, we will not be able to understand which is the appropriate evidence part to support it. Moreover, it is difficult to assess the approaches focusing only non evidence elements. The efficiency of evidence depends on the goals they claim to support.

- *Analysis* The evaluation or analysis process will typically identify structural errors in security cases. Also, they identify errors in their content make the arguments more comprehensible to the stakeholders of the system. There are recent efforts to define a standardized or formal approach to represent an argument in an assurance case. As a result, arguments are expressed in different styles and exhibit different characteristics. The first contribution that is interested to the evaluation problem is the work of Chowdhury et al [23]. They proposed rules that semi-formally define the structure and content of assurance cases. These rules guide the work of assurance case developers and reviewers. Assurance cases developers are instructed to use a more rigorous approach to their arguments. External reviewers have a basic checklist that guides them in assessing the rigor of arguments.
- *Tool support:* The main concern demanding tool support is the passage from argument pattern and template models to the generated security case. Several works demonstrate the use of the tooled methods to apply argument patterns to generate safety assurance case [30]. However, these works are constrained by concepts chosen for modeling safety, e.g., hazard, safety architecture, and risk analysis. Further guidance is still required to interpret the concepts from the security engineering structured models to their assurance models.

2.5 Conclusion

In this chapter, we surveyed several existing approaches for creating security cases. We outlined and analyzed their features and capabilities regarding our research's key objective: provisioning methodological support for non-savvy engineers to conduct integrated assurance of security in the software engineering process. First, we provided a basis to analyze the existing approaches for the creation of security cases in Section 2.3. To facilitate this survey, we categorized the identified approaches into 1) model-based, 2) standard-based, 3) asset-based, and 4) model-based approaches. We also defined a multi-attribute taxonomy relying on previous surveys and knowledge in the domain to analyze the selected contributions. By doing that, we prepared a characterization summary of these

approaches.

According to the findings in Section 2.4.2, a new methodology is needed. It should consider the existing approaches and fulfill the identified methodological lacks for incorporating assurance concerns in the system security development in unison. The approach should be holistic, adopting the secure-by-design paradigm, targeting, particularly the architecture design phase. Despite existing approaches addressing different stages development stages, mainly in a segregated fashion, further work is needed to integrate these approaches during the software architecture design. Besides, the conceived methodology should be benefited from the reusable security assurance models, including security argument patterns and templates. Accordingly, the approach must improve the creation of security cases regarding the complexity associated with understanding the security engineering activities and artifacts.

Part III

Contribution

Chapter 3

Approach

Contents

3.1 Introduction **43**

3.2 Need for Security Assurance – The Stakeholder's Perspective **44**

3.3 Method to Build the Security Assurance Modeling Framework **45**

3.3.1 Modeling framework development process 46

3.3.2 Application development process 47

3.4 Supporting the approach within SDLC **47**

3.5 Use Case: Autonomous Avoidance Detection System **49**

3.6 Research methodology **51**

3.7 Conceptual model **51**

3.8 Conclusion **54**

3.1 Introduction

In this chapter, we present a general overview of our proposed approach to ease software security assurance concerning the overall research problem stated in Section 1.2, which is—*How to define and assess a framework for the generation of security cases using reusable models?* The aim is to articulate a methodological view of the overall development process that is used in the rest of this work to incorporate security assurance into the system's architectural design. The proposal starts by adopting stakeholders' viewpoints in the context of the assurance process and pursuing support and guidance to ensure software

security. Subsequently, we outline the key aspects upon which the approach is constructed and applied. This includes the development of a security case modeling framework to support the security assurance case creation process (addressing the *Problem* **P1**) and the definition of a modeling language for security cases (addressing the *Problem* **P2**).

Overall, the chapter is organized as follows: In Section 3.2, we present an instance of the stakeholders' interaction in a typical software engineering process to derive the need for security assurance. This is followed by a method to build the security case modeling framework in Section 3.3. In Section 3.4, we highlight the potential incorporation of the method into a system development lifecycle. This is followed by an introduction to the use case in Section 3.5 that will be useful to illustrate and assess the approach and specific contributions as part of it. The research methodology is detailed in Section 3.6. After that, Section 3.7 focuses on constructing a conceptual model that serves as the foundation of the security assurance framework. Finally, we conclude by summing up the contribution of the chapter in Section 3.8.

3.2 Need for Security Assurance – The Stakeholder's Perspective

Developing security-critical systems requires a dedicated engineering process incorporating security assurance concerns. Considering the security measures from early stages in system development can significantly impact the ability to provide security assurance of such systems. The goal is to enable developers to include appropriate security assurance and evaluation aspects early in the development of a system to avoid relying on retrospective security audits. These aspects, in turn, call for 1)the definition of security engineering processes that allow gathering early evidence to mitigate security concerns; 2) the generation of arguments for the security of the system under assurance; and 3) the coordination and harmonization of collaborative work among dispersed stakeholders and their tasks in the engineering process.

However, the aspects above lead to a complex enrichment process that often lacks methodological guidance, modeling language including semantics, and automated tool support.

For example, we consider the scenario depicted in Figure 3.1 as an instance to capture the interaction among different stakeholders in a development process driven by models. This interaction typically occurs in the following order, the numbers in parentheses corresponding to the numbers in the figure: 1) the *Security expert*, and *Process expert*

elaborate a variety of models encapsulating their respective knowledge or interests. 2) The *Assurance expert* defines assurance models that ensure the security engineering processes. The assurance models are defined to be reused later. 3) The Architect considers the security and process models and harmonizes the domain expertise and outcomes to build the secure system architecture model. 4) The Architect constructs security cases for the software architecture model based on the assurance models. The security cases are supported by the architecture related security evaluation artifacts. This, in turn, forms the basis to facilitate the System architect's tasks. Notably, the construction of a security case does not require direct collaboration with an assurance expert. Nevertheless, such collaboration makes the definition of assurance models complex, particularly when perceiving security models at different system process activities and jointly designing and analyzing security-related feared events to produce artifacts or solutions with the expected security features at the architecture level.

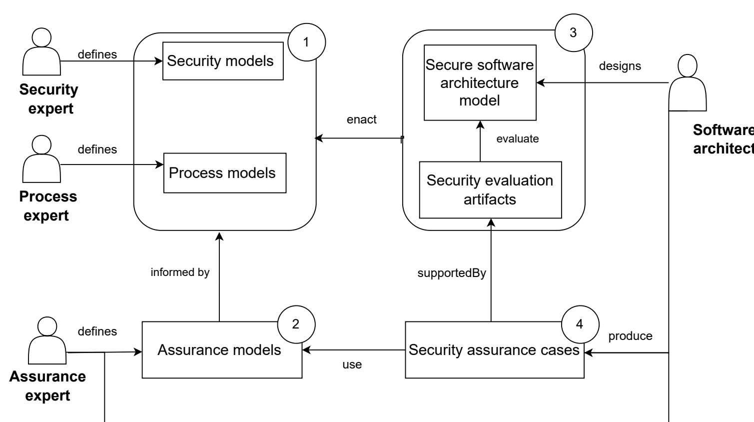


Figure 3.1: System Engineering (SE) Process for Security Assurance: An Instance of the involved Stakeholders.

3.3 Method to Build the Security Assurance Modeling Framework

The proposed methodology supports the security assurance of system software architectures. The goal is to capture various security concerns of the software under assurance and provide argumentation to reason about its security. The security assurance argumentation is delivered as verified reusable models. The approach enables the generation and management of evidence elements using security evaluation and measurement methods.

The resulting security case provides guidelines to design secure software architecture by following the ensured design activities. This is accomplished via the following main steps that are depicted in Figure 3.2:

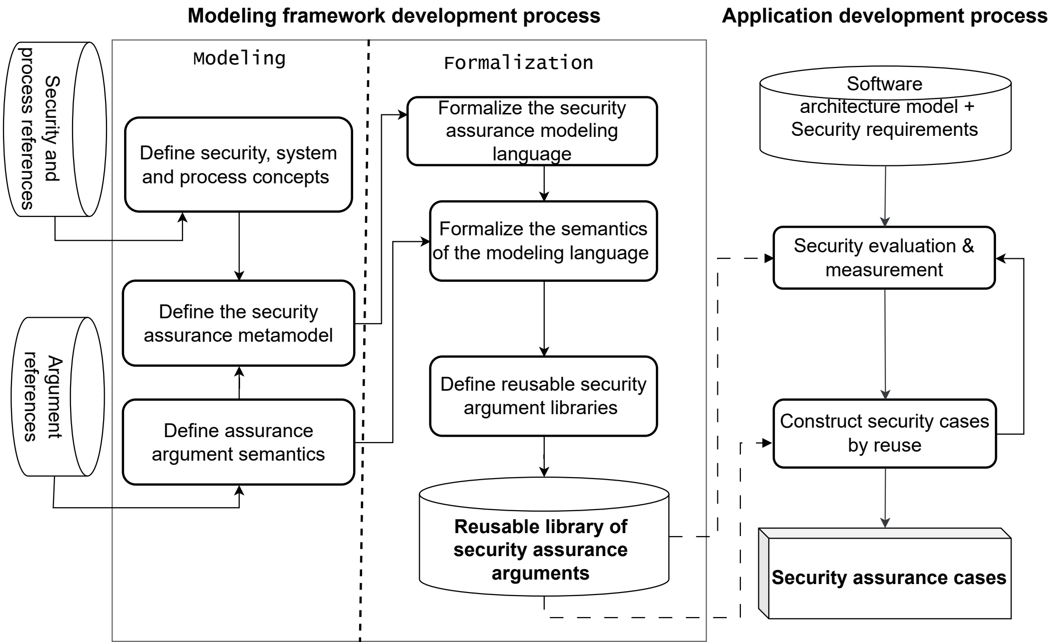


Figure 3.2: Method to Build a Framework to Support the Security Assurance process

3.3.1 Modeling framework development process

To support the specification and verification of reusable security argument models, we consider both the modeling and formalization activities.

Modeling We begin by defining high-level concepts related to the process, the security, and the system. The concepts describe the security engineering activities associated with the architecture design phase. These concepts are used to personalize a metamodel and make it more dedicated to describing security assurance cases. In addition, we reinforce the metamodel with some semantics rules. We provide static semantic rules to ensure that the security assurance models are well-formed. Moreover, we provide dynamic semantic rules to make the argument less refutable. The entry points of the approach are expertise, best practices, and development processes from the security and process experts.

Formalization After constructing the security assurance metamodel, we formalize the developed metamodel using a suitable language with automated tool support for

specifying and verifying semantic rules related to security assurance models. Moreover, we formalize some semantics of the modeling language. By doing so, we obtain a formal security assurance modeling language to define reusable security argument libraries (patterns and templates). This library facilitates the creation of security cases for a concrete software architecture model.

3.3.2 Application development process

First, we start with a concrete secure software architecture model and security requirements. A software architecture is said to be secure when it encompasses various mechanisms and policies to fulfill security requirements throughout the software development lifecycle. Then, we use the security argument libraries to identify adequate arguments that fit with the claims about the security of the software architecture. The selected argument pattern(s) or template indicates the security evaluation and measurement activities that should be handled to gather the necessary evidence to support the security case. The evidence elements are the artifacts resulting from the different security engineering activities. After that, we construct security cases by mapping evidence elements to the argument pattern(s) or template. Subsequently, we verify the satisfaction of the security requirements in the security case. If we determine that the claims are not sufficiently supported, due to a lack of evidence elements or other argument patterns, we go back to the security evaluation or the pattern library, respectively. In this way, we iterate over the process development and the evidence generation to revise and improve the security assurance case. As a result, we converge on a complete security case.

3.4 Supporting the approach within SDLC

In security-critical domains, it is widely accepted that security measures must be integrated throughout the entire lifecycle. As a result, methodologies for building secure systems should include security assurance in all stages. Security cases can be defined for each stage to ensure compliance. A more effective approach involves extending the development process to include security engineering and assurance activities at every stage, making it more practical for practitioners. A key concept in our proposed method is applying security principles consistently and demonstrating adherence through security cases.

We use the waterfall [11] development lifecycle as the basis for our approach. Moreover, we focus our approach in this work on the architecture design stage. Figure 3.3 sketches a

software development cycle that we consider effective for defining security cases of software architecture. It shows how the activities defined in Section 3.3 come as a supplement to the architecture design phase.

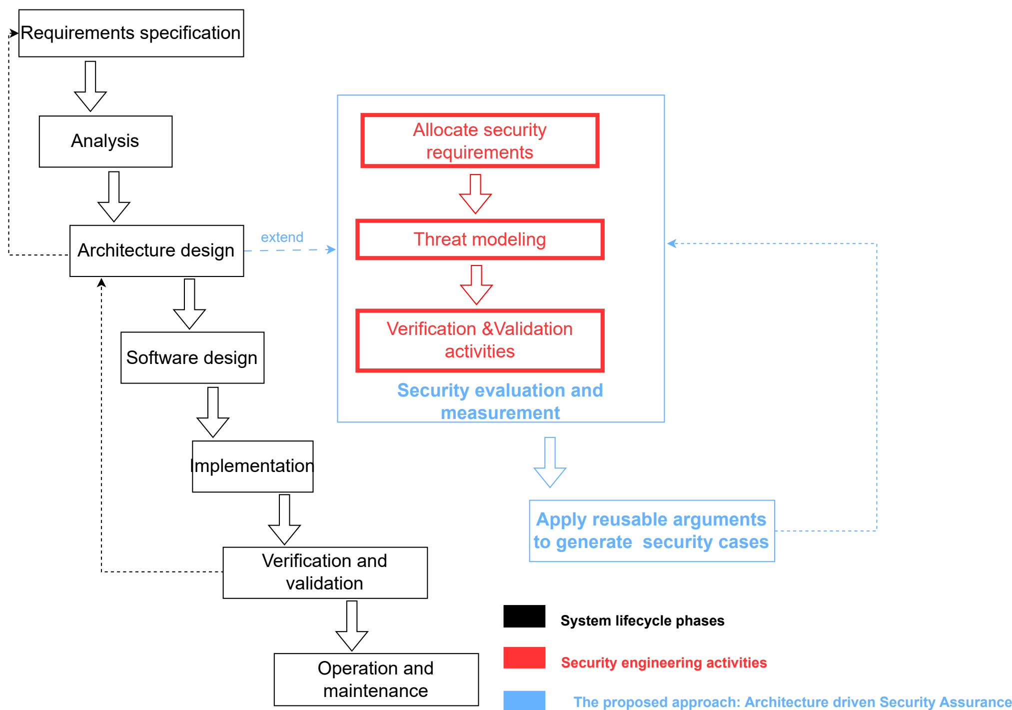


Figure 3.3: Integration of the approach in the development model

On the one hand, the architectural design is extended to include security engineering activities. First, security requirements are allocated to different assets of the software architecture. This is followed by threat modeling, which involves analyzing the software architecture while considering the previously allocated security requirements. Security threats can hinder the fulfillment of these requirements. Finally, security requirements are verified and validated to ensure that the software architecture model meets the desired security standards. All these extended security engineering activities constitute the first step in security assurance, known as security evaluation and measurement, to generate evidence to support the security assurance case.

On the other hand, we reuse the argument models and create the security cases for the system under assurance.

Finally, security cases are constructed and supported by evidence from the previous activity. The security cases can be used as a plan for the verification and validation of the target under assurance.

For simplicity, we show the proposed approach within a waterfall model. However, it can similarly be integrated with other system development life-cycle. As a result, the proposed approach enables the generation of security cases early enough in the life cycle to apply useful assurance analysis within the design loop, thereby supporting the principles of security by design.

3.5 Use Case: Autonomous Avoidance Detection System

In this section, we introduce the use case of Avoidance Collision Airborne Systems (ACASs) as an automotive domain-specific application to illustrate the proposed approach. We demonstrate the applicability of our approach to assuring this example in Chapter 7.

System description

We illustrate our contributions through an example of a use case scenario from ACAS Xu [74, 5] used for collision avoidance in Unmanned aircraft systems (drones). We consider encounters between two drones. We take the perspective of one of them and call it ownship. The ownship is equipped with ACAS Xu and has a functional space in which it must operate. This space is conceptually partitioned into two areas: collision avoidance threshold and collision volume with a high risk of collision for the ownship with intruders. When no risk of collision is detected, the ownship follows the current heading to the destination area. Otherwise, if another drone is detected in the collision volume, the ownship will turn right or left to avoid the collision and prevent the intruder from reaching the collision avoidance threshold. The system is visualized in Figure 3.4. The ownship sensors collect data about an eventual intruder (velocity, angle, distance, etc.). This data is then sent to the processor that computes a suitable decision to avoid collision (turn left, turn right, do nothing). According to the decision, the planner plans a trajectory to keep navigating while avoiding a collision. Finally, the actuator executes actions to follow the planned trajectory.

Relevance of the ACAS case study regarding security assurance

The planning component in ACAS Xu is based on Machine Learning techniques. Moreover, it is a Deep Neural Network (DNN). DNNs are known to be vulnerable to a wide range of security threats, including adversarial examples, overfitting, data poisoning,

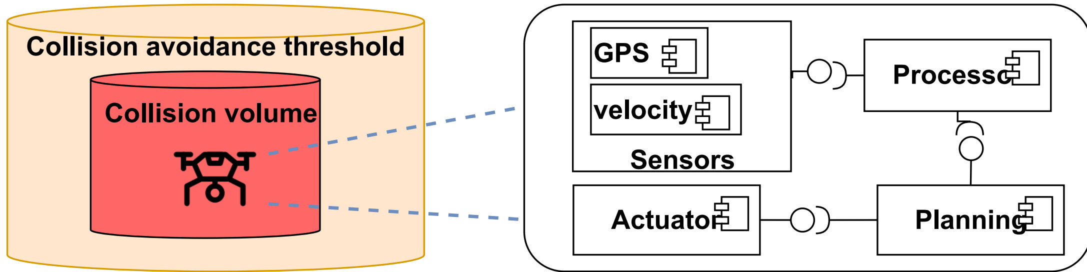


Figure 3.4: Architecture of ACAS Xu

model extraction, and Trojan attacks [49, 19]. Therefore, incorporating DNN in security-critical systems introduces new potential risks. ACAS Xu is vulnerable to adversarial examples threats. Adversarial examples are well-crafted inputs fed to the DNN to deceive it, i.e., assign a given input to a class it does not belong [129]. They are obtained by slightly modifying a sane (i.e., initially well-classified) input. Formally, given a DNN that assures the function f , the attackers find a minimal perturbation to add to a sane input x to obtain a new input x' so that $f(x) \neq f(x')$.
Furthermore, we consider a realistic scenario of airborne collision caused by the presence of adversarial examples. As shown in Figure 3.5, usually, when the distance (one feature of the DNN) between the victim ship (ownship) and the intruder is large, the victim ship advisory system will advise the left to avoid the collision and then advise right to get back to the original track. However, if the DNN is not verified, there may be a specific situation where the advisory system, for certain approaching angles of the attacker ship, advises the ship incorrectly to take a right turn instead of a left, leading to a fatal collision. If an attacker knows about such an adversarial case, he can specifically approach the ship at the adversarial angle to cause a collision.

For such critical domains, we hav to question the trustworthiness of the systems. Also, we must establish assurance cases that show, with sufficient confidence, that the system under assurance will perform the tasks securely. The assurance of DNN-based systems differs from classical systems because DNN are data-driven and also have characteristics quite different from classical systems. The key open challenge is the need for security cases for data-driven applications and the development of learning algorithms with provable security guarantees.

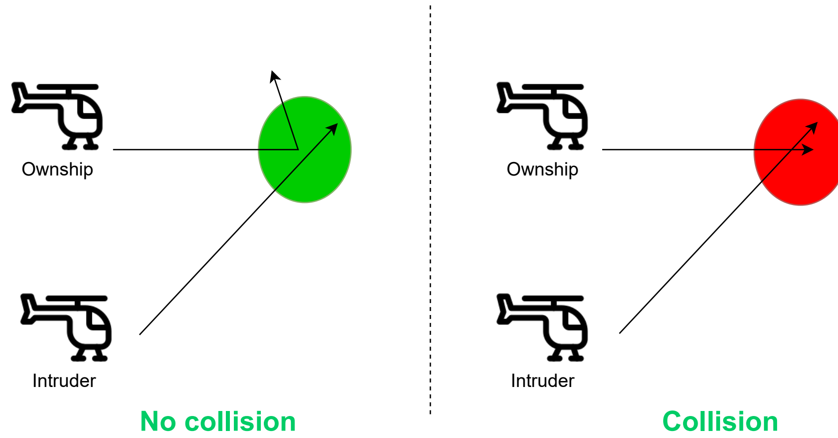


Figure 3.5: Use Case Scenario: Collision caused by adversarial examples. Revised from [120]

3.6 Research methodology

This section accounts for the methodology we followed to construct the proposed security assurance method and address the problems in this Ph.D. Figure 3.6 depicts the research methodology steps as following.

The first step involved the identification of the domain concepts to create a conceptual model of security assurance cases. Furthermore, we worked on defining the semantics specifying the meaning and interpretation of these concepts, including the rules and logic that govern their interactions.

The second step involved the development of a dedicated modeling language to facilitate the security assurance case generation and interpreting the modeling language to a formal-based tooled language.

The third step involved the implementation of the modeling language and formalisms to facilitate the system assurance for a target application and analyze the generated security cases.

3.7 Conceptual model

In Section 3.4, we indicated the need for an early incorporation of security in developing any system to simplify the generation of the required evidence, thus constructing security cases. We achieve this task in the first step of our approach by creating a conceptual model

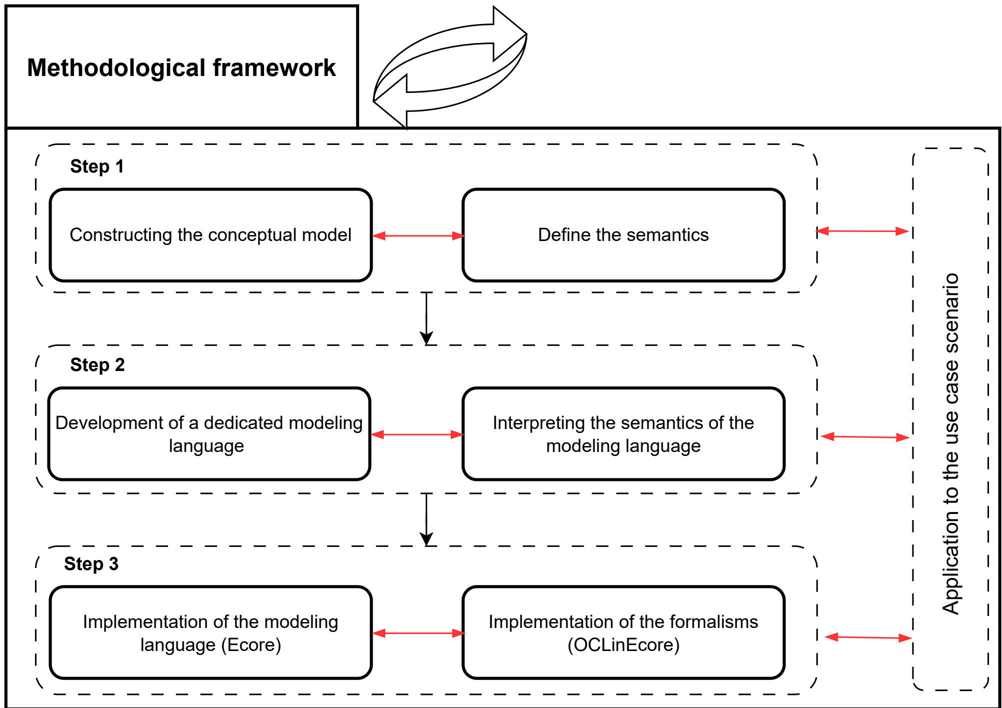


Figure 3.6: Methodological framework

of security assurance cases. A conceptual model of security cases provides a common understanding of all concepts employed in this thesis to ensure a precise description of the addressed problems and solution approaches.
A conceptual model of a security assurance case should capture the main concepts and relationships to describe the security case models in the context of different standards and domain-specific practices. We employ UML class diagrams to describe the conceptual model, where concepts are represented by classes, concept attributes are represented by class attributes and relationships between concepts are represented by associations. When an attribute's value belongs to a predefined set of possible values, we use enumerations. Package notation is used to create groupings of concepts.
A graphical representation of the concepts and relationships from the excerpt is given in Figure 3.7. To improve understanding and readability, the attributes of the different concepts and the links between the concepts are not described. We note that the model in Figure 3.7 is a partial representation of the concepts and relationships relevant to the security case definition and usage. .

Assurance Package contains the elements that allow the construction of a complete

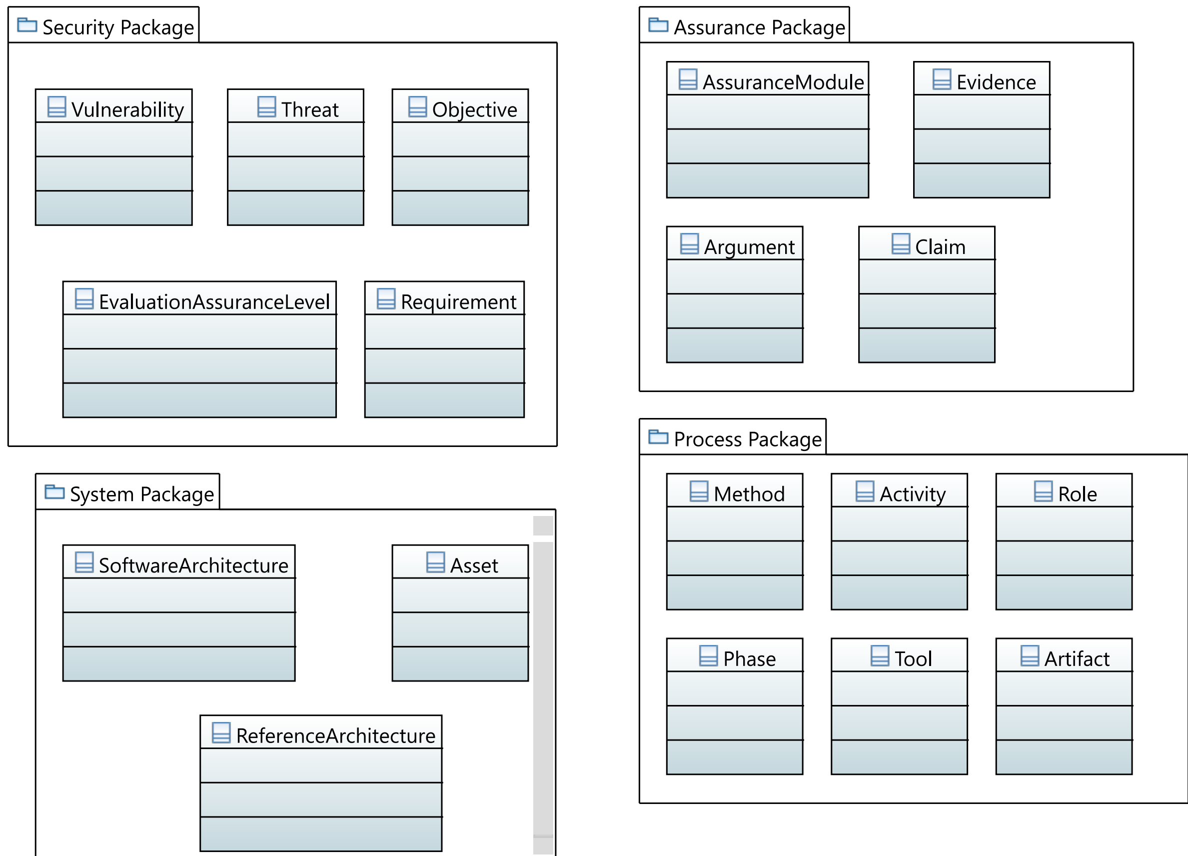


Figure 3.7: Conceptual model for the security assurance modeling language

assurance case including *claims* about the security of the system under assurance, *arguments* to demonstrate why the claim holds, and finally, *evidence* elements that support the argument.

Security Package focuses on security-related concepts and mechanisms essential for protecting the system against malicious behaviors, including *requirements*, *objectives*, *threats*, *vulnerabilities*, and *evaluation assurance levels*.

Process Package contains classes related to processes employed within the software development life-cycle, including *activities*, *roles*, *techniques*, *tools*, and engineering *artifacts*.

System Package contains classes that allows the description of the system at the architectural design phase, including *software architecture*, *assets*, *reference architecture*.

3.8 Conclusion

In this chapter, we presented a global view of the proposed approach for security case creation. The approach relies upon MDE and formal-based techniques as the primary means for addressing security assurance. The proposed method allows the creation of the security assurance framework to assist the security engineering process. Moreover, the approach is applied and evaluated in the context of the ACAS use case. This will provide methodological support to the system engineer via capturing the security and assurance expertise in the early design stages of the system development.

In the forthcoming chapters, we apply our research methodology to address the problem stated in the chapter 1 via a detailed description of the following:

- Security assurance framework for modeling and analysing security assurance models in Chapter 4.
- Methodologies supported by the security assurance framework, specifically, the definition of libraries of patterns and templates for security assurance in Chapter 5.
- A catalogue of reusable argument patterns and templates that will ease the creation of security cases in Chapter 6.
- Application of the approach and proposed tool-chain support prototype in the context of the ACAS use case in Chapter 7.

Chapter 4

Security Cases Modeling Language (SCML)

Contents

4.1 Introduction **55**

4.2 Desired features of the language **56**

4.3 Abstract Syntax **57**

4.3.1 Assurance Package 58

4.3.2 Process Package 61

4.3.3 Security Package 63

4.3.4 System Package 64

4.3.5 Relationships between the different packages 66

4.4 Semantics **67**

4.4.1 Static semantics 67

4.4.2 Dynamic semantics 68

4.5 Concrete Syntax **69**

4.6 Conclusion **71**

4.1 Introduction

This chapter is dedicated to describe one of the main constituents of our proposed approach. This part comprises the security case modeling language for addressing the *Prob-*

lem **P2**, which is defining a modeling language for security assurance cases integrating software architecture and security engineering activities, as mentioned in Section 1.2. To this end, we propose a Domain-Specific Modeling Languages DSML, relying upon UML [38] to define concepts and semantics corresponding to the assurance, the software architecture, the security, and the process development. Furthermore, the DSML is implemented using metamodels for the abstract syntax and xtext grammars for the concrete syntax to provide a standardized modeling environment.

Therefore, we define a domain-specific modeling language that we call **S**ecurity **C**ase **M**odeling **L**anguage SCML. The definition of SCML involves three key components: abstract syntax, semantics, and concrete syntax.

The remainder of this chapter is organized as follows. Section 4.2 presents the features that seem fundamental for SCML. Section 4.3 presents the abstract syntax of the SCML language which is more detailed and precise than the conceptual model presented 3.7. Section 4.4 describes some semantic rules added to the abstract syntax. Section 4.5 transits from the abstract syntax to the concrete syntax of the security assurance modeling textual language. Finally, Section 4.6 concludes the chapter.

4.2 Desired features of the language

Globally, the aim is to allow harmonizing knowledge derived from the stakeholders involved in the software architecture engineering process, namely architects, security analysts, and assurance case developers, and to capture basic concepts in a generic way, facilitating security assurance. To this end, the desired features of the modeling languages are described as follows:

- *Req.1* Allow capturing all the concepts and relationships represented in the conceptual model.
- *Req.2* Provide a concrete syntax for the creation of security cases.
- *Req.3* Allow the creation of reusable security argument patterns.
- *Req.4* Allow the creation of reusable security argument templates
- *Req.5* Allow the creation of security cases by applying patterns and/or templates.

- *Req.6* Provide precise semantics, enabling rigorous analysis, verification, and reasoning about security argument models. It should support formal methods to verify the correctness of security assurances.

- *Req.7* Have commonalities with well-known assurance languages previously presented in Section 1.5.1.2. In other words, we can easily identify the argumentation and evidence parts in the assurance models.

In this work, Eclipse Modeling Framework Technology EMFT is used. All metamodels are specified using the EMF. However, our vision is not limited to the EMF platform. Other modeling tools conforming to the previous requirements can also be used.

4.3 Abstract Syntax

The starting point in the definition of SCML is the abstract syntax. We propose an abstract syntax (a metamodel) using an OMG-style metamodel to construct the security case modeling language. The abstract syntax is based on previous desired features and describes various concerns from the conceptual model presented in Section 3.7. SCML is used to model security assurance cases while fostering the reuse of argument models (patterns and templates). However, since it is very costly to define a new language from scratch, SCML is defined by adopting and adapting meta-models from previous works that describe concepts presented in the conceptual model. The Goal Structuring Notation (GSN) presented in Section 1.5.1.2 corresponds mainly to an assurance metamodel. Thus, it should be extended with (or linked to) other modeling formalisms to enable a more detailed definition of the system and analysis of its security. Generally, GSN must be extended to deal with the linkage between its assurance framework and system architecture models.

The principal classes of SCML are described in the form of a class diagram with Ecore notations. In the class diagrams, concepts are represented as classes and concept attributes as class attributes. Relationships are represented by associations. Generalization associations are used to derive more specific concepts from abstract ones. When an attribute assumes a value from a predefined set of possible values, we use enumerations. Finally, the package notation is used to make groupings of concepts and thus better manage complexity. The meanings of the principal concepts are explained in the following paragraphs.

4.3.1 Assurance Package

The assurance package metamodel was based on GSN standard metamodel as given in [45]. In addition, we extend it with concepts that allow reusing assurance models. The reusable models are patterns and templates. The principal concepts of this package are illustrated in Figure 4.1 and detailed as follows. Concepts are represented in **Bold** and their attributes are represented in *Italics.*:

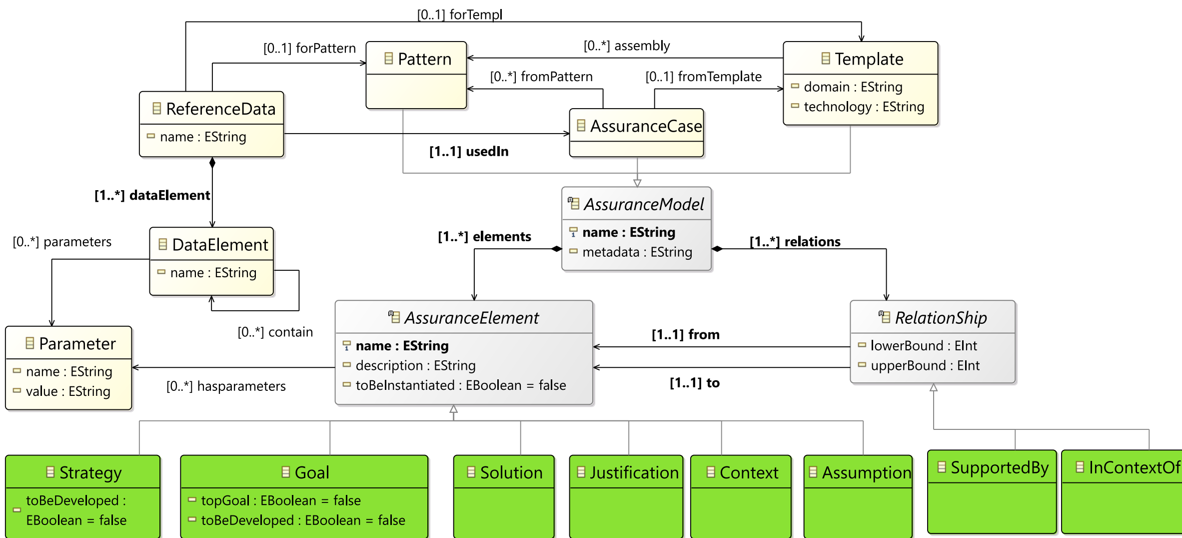


Figure 4.1: SCML: Assurance Package

Abstract concepts

- **AssurancePackage** the abstract containing element for a structured assurance model.
  - *metadata* provides information about the assurance model itself, but not necessarily about the content of the package. For example, we can document details such as version information, author, license, dependencies, and more.
- **AssuranceElement** the abstract container that holds all elements of the assurance case.
  - *description* refers to the content of the assurance element. It describes the meaning or purpose of that element. *toBeInstantiated* indicates whether, at

some later stage, the 'abstract' assurance element needs to be replaced (instantiated) with a more concrete instance (*toBeInstantiated* = True) or not (*toBeInstantiated* = False).

- **RelationShip** the abstract association class that enables the argument elements of any structured assurance model to be linked together.
  - *lowerBound* indicates the minimum number of occurrences allowed for the relationship.
  - *upperBound* indicates the maximum number of occurrences allowed for the relationship.

GSN concepts GSN concepts presented in Section 1.5.1.2 are shown in green in Figure 4.1. We recall their definition and present their attributes.

- **Goal** It is a claim that should hold in the argument. It can be intentionally left undeveloped in the argument.
  - *toBeDeveloped* indicates whether the development of an argument is required to support this goal
  - *topGoal* indicates whether this goal is the highest goal in the tree based assurance model hierarchy
- **Strategy** It describes the nature of the inference between a goal and its supporting goal(s). It can be intentionally left undeveloped in the argument.
  - *toBeDeveloped* indicates whether the development of an argument is required to support this strategy (*toBeDeveloped* = True) or not (*toBeDeveloped* = False).
- **Assumption** It presents an intentionally unsubstantiated statement.
- **Justification** It presents a statement of rationale.
- **Solution** It presents a reference to an evidence item or items.
- **Context** It presents a contextual artefact. This can be a reference to contextual information, or a statement.
- **InContextOf** It is a particular relationship that connects *Goal* or **Strategy** assurance elements to **Context** or **Justification** or **Assumption** elements.

- **SupportedBy** It is a particular relationship that connects *Goal or Strategy* assurance elements to *Solution or Strategy* elements.

Attributes *lowerBound* and *upperBound* are used when the exact number or form of relationships can vary depending on the specific use of the argument pattern. For example, an argument structure might repeat, but the number of repetitions depends on the context in which the pattern is used.

- *lowerBound* = 0 and *upperBound* = 1 indicates that the relationship is optional. It declares possible alternatives for satisfying a relationship.
- *lowerBound* = 1 and *upperBound* = ? indicates that the relationship will repeat an undetermined number of times.
- *lowerBound* = m and *upperBound* = n indicates that their exit m of n selection of the relationship.

Reusablity concepts We added the following concepts that allow us to define reusable patterns and templates

- **Pattern** an assurance model usually derived from recurrent activities of an engineering methodology and/or from standards.
- **Template** an assurance model that is semi-complete and quickly applicable compared to the patterns. We propose to define templates per domain and by grouping partially developed patterns.
  - *domain* indicates the domain application of the system under assurance where the template is applicable
  - *technology* indicates the technology used to develop the system under assurance where the template is applicable
- **AssuranceCase** an argumentation structure that is ready to convince stakeholders about the dependability (security, safety) of the system. It can be produced by applying patterns or a given template.
- **Reference data** a tree based data structure. It is necessary to automatically apply the patterns and/or templates.
- **Data element** the basic element of the reference data. Each data element is composed itself by other data elements.

- **Parameter** a couple of names and values. It is used to define non-instantiated assurance elements in argument patterns, templates, and reference data.

4.3.2 Process Package

The process package regroups the concepts related to the engineering process. It is based on the metamodel proposed in [48]. By including the process concepts, we ensure having an argumentation that is informed by the engineering process.
The principal concepts of this package are illustrated in Figure 4.2 and detailed as follows

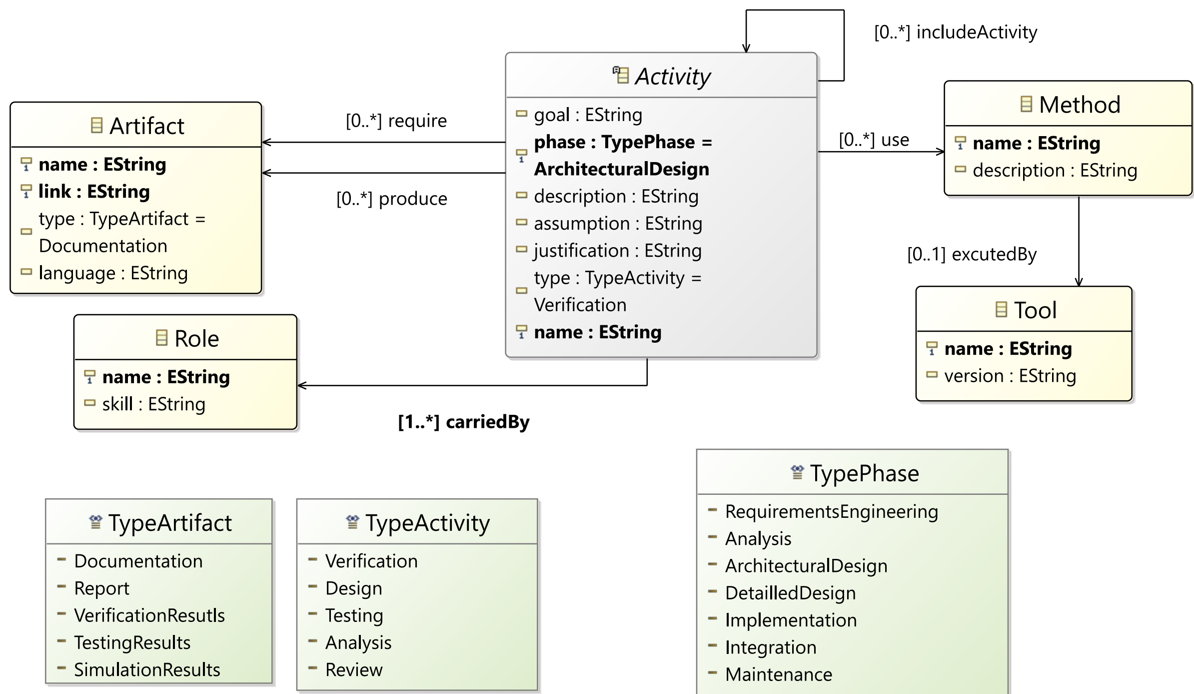


Figure 4.2: SCML: Process Package

- **Activity** represents a distinct, executable piece of work within the software development process.
  - *Phase* represents a significant period in a project. An enumeration of the possible phases is called **TypePhase**
  - *goal* represents objectives or desired outcomes for the activity, such as specific performance metrics or quality benchmarks. For an activity related to software

testing, the goal might be to ”ensure that the system operates without critical bugs before release”.

  - *assumption* represents the conditions that are meant to be true or must hold for the activity to succeed. For example, for a testing activity, an assumption could be that ”all necessary testing environments will be available and functional before testing begins”.
  - *justification* explains the rationale behind selecting the activity, detailing why it is necessary or valuable in the overall process. For example, a justification behind the code review activity is ”code review is essential to improve code quality, catch errors early, and ensure adherence to coding standards”.
  - *type* selects among an enumeration named **TypeActivity**. It indicates the type of the activity.

- **Role** represents the performer of the activity.
  - *Skill* represents the specific knowledge, expertise, or abilities that a person (or entity) in a particular

- **Tool** refers to any software used to facilitate or automate activities within the software development process.
  - *version* tracks updates and changes of the tool.

- **Method** refers to a method, approach, or procedure used to accomplish a specific activity.

- **Artifact** represents an object produced or used during the software development process.
  - *link* contains a URL or file path that points to the digital location where the artifact is stored.
  - *type* categorizes the artifact based on its nature. A predefined set of possible types is indicated in the enumeration **TypeArtifact**
  - *language* indicates in which language the artifact is written

4.3.3 Security Package

The concepts of this package are extracted from the Information System Risk Management metamodel presented in [31]. The principal concepts of this package are illustrated in Figure 4.3 and detailed as follows

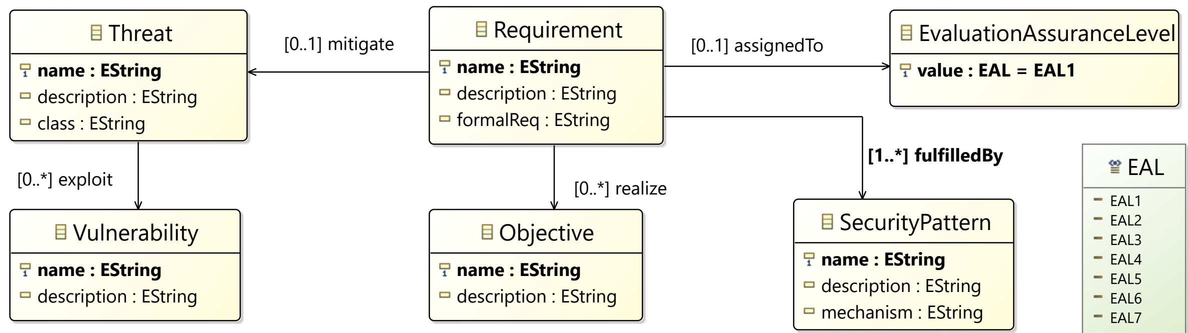


Figure 4.3: SCML: Security Package

- **Requirement** specifies a condition or constraint that a system must satisfy to mitigate potential threats or to realize security objectives.
  - *formalReq* contains the formal specification of the requirement.
- **Threat** refers to a potential danger, risk, or malicious entity that could exploit vulnerabilities in a system or environment to compromise its security.
  - *class* categorizes the type or nature of the threat. There are other ways to classify cyber threats, such as STRIDE.
- **Vulnerability** is a weakness, flaw, or gap in a system's defenses or design that could be exploited by threats to compromise its security.
- **Objective** represents a specific goal or aim that a system aims to achieve regarding its security posture.
- **Assurance Evaluation Level** represents the degree of confidence or assurance that a system, product, or service meets its security requirements and objectives.
  - *value* represents the security evaluation level assigned to a system or component. The value ranges from EAL1 to EAL7, where each level reflects the

depth of security analysis, testing, and assurance provided. The values of the evaluation assurance level are defined in the enumeration **EAL**, where higher levels indicate more rigorous evaluation and greater confidence in the system's security.

- **Security pattern** describes mechanisms that can stop specific security threats, and they also embody principles of good security design
  - *mechanism* refers to the specific technical or procedural methods that are employed to implement the security pattern

4.3.4 System Package

The principal concepts of this package are illustrated in Figure 4.4 and detailed as follows. It is adopted from the component-based metamodel presented in [97]

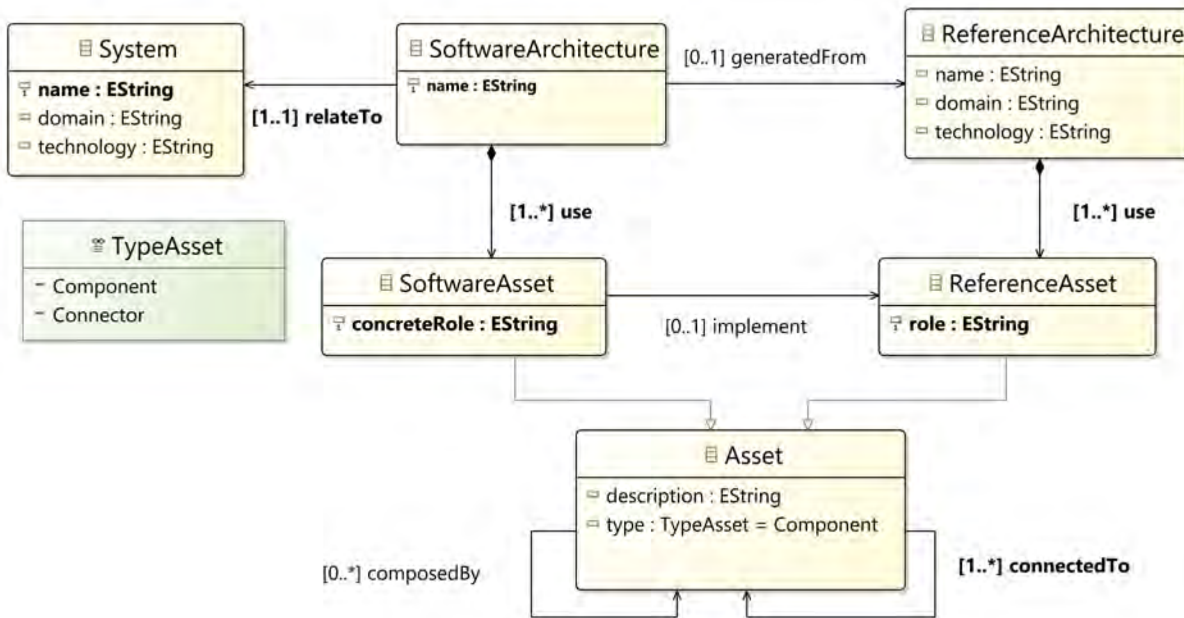


Figure 4.4: SCML: System Package

- **System** describes the system under assurance.
  - *domain* indicates the specific area of application or industry to which the reference architecture applies

  - *technology* specifies the key technologies, platforms, and tools that are recommended or required within the reference architecture.

- **SoftwareArchitecture** represents the architecture for a specific software system. It is derived from the reference architecture but is customized to meet the system's specific requirements.

- **ReferenceArchitecture** presents a description of a reference architecture related to a domain and/or a particular technology. The reference architecture typically covers high-level system components, their interactions, and their alignment with business goals or technical constraints.

  - *domain* indicates the specific area of application or industry to which the reference architecture applies

  - *technology* specifies the key technologies, platforms, and tools that are recommended or required within the reference architecture.

- **Asset** refers to an entity of hardware or software, or both, capable of accomplishing a specified purpose. A software architecture is described using connected assets where some can be composed of other assets.

  - *type* indicates the type of the asset. The types are described in an enumeration called **TypeAsset**.

- **ReferenceAsset** a sub-type of asset. It is used to describe reference architectures.

  - *role* indicates an abstract function or responsibility that the asset fulfills in the context of the reference architecture. This attribute defines the general role that a component plays within the architecture, such as managing payments or handling user information, without specifying any concrete implementation.

- **SoftwareAsset** represents the assets used to compose the software architecture. Each software asset is an implementation of a reference asset.

  - *concreteRole* represents the specific, implemented version of the abstract role in a real software system.

4.3.5 Relationships between the different packages

We additionally define structural and semantic linking via associations between concepts from different packages to ensure consistency during the construction of security cases. Accordingly, Figure 4.5 depicts the resulting interactions between the packages. This is a choice that simplifies the structuring of security cases. These links are described as follows:

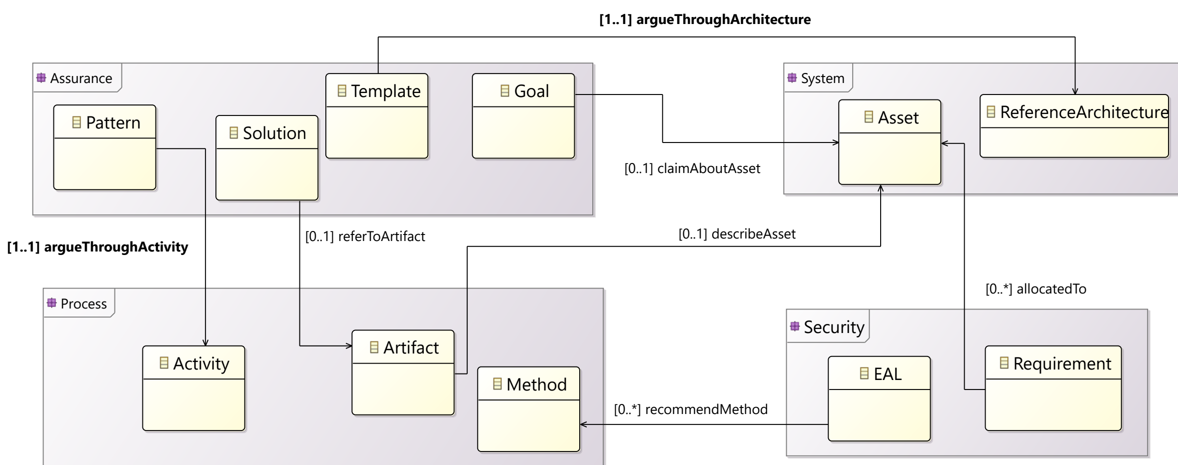


Figure 4.5: SCML: relationships between the packages

- *recommendMethod* An **EvaluationAssuranceLevel** recommends specific methods that aim to improve the security of the system under assurance.
- *allocatedToAsset* Each security **Requirement** is allocated to some assets that need to be protected from security issues.
- *referToArtifact* An **Artifact** produced while developing the system is used as evidence to support the assurance case.
- *protectedBy* An **Asset** is protected by **SecurityPattern**s. The security mechanisms that are used to fulfill **security requirements** at the architectural level are **security patterns**.
- *argueThroughActivity* A **Pattern** argues through an activity of the system engineering process. This will justify some argumentation strategies.

- *argueThroughArchitecture* A **Template** argues through a reference architecture related to a domain application, a technology, or both. This will justify some argumentation strategies and indicate the security requirements needed to secure the systems built upon this architecture.
- *claimAboutAsset* A **Goal** claims about an **Asset**.
- *describeAsset* An **Artifact** might describe the documentation, and the procedures revolving around the management, protection, and utilization of a particular *Asset*.

4.4 Semantics

The second step of defining SCML is the definition of language semantics. It includes the static semantics that define the structural properties of valid models. The dynamic semantics are concerned with the application of the assurance models.

4.4.1 Static semantics

These rules will help find inconsistencies and avoid error-prone buildings. They only allow security cases without structure errors.

- `Rule 1.` There shall be one and only one relationship between any two assurance elements
- `Rule 2.` Each assurance model has only one top goal.
- `Rule 3.` Only valid relations shall exist between any two assurance elements, for example, InContextOf is accepted between context and coal elements. Unlike, it is not accepted between two goal elements.
- `Rule 4.` Each developed goal must be directly supported with solution or supported with sub-goals that are directly supported with solution elements.
- `Rule 5.` The relations in the assurance case have upper bound and lower bound equal to one.
- `Rule 6.` The assurance elements to be instantiated are only accepted in patterns and/or templates.
- `Rule 7.` The to-be-developed strategies and goals are only accepted in patterns.

4.4.2 Dynamic semantics

These properties reinforce the argument's logical structure and provide stakeholders with a more comprehensive understanding of the assurance case, ultimately contributing to a more compelling and defensible presentation of software security and reliability. Moreover, they specify how the patterns and the templates are applied. They are evaluated whenever the user attempts to instantiate an assurance model within the system under assurance.

- `Rule 8.` The application of a pattern with a relation that has upperBound = n depends on the security concept on the argumentation strategy, for example, if the argumentation strategy implies arguing through all the security requirements we should have as many requirements as subgoals.
- `Rule 9.` The composition of the patterns implies renaming assurance elements with the same name.
- `Rule 10.` When an activity includes other activities, the assurance model should assume the correctness of the outputs of that activity or develop sub-goals to argue about the correctness of the included activities.
- `Rule 11.` The methods that appear in an assurance model must be recommended by the Evaluation Assurance Level required for the security assurance of the system.
- `Rule 12.` When a template is selected to create the assurance case of a system under assurance, the domain application and development technologies of the system conform to those of the template.
- `Rule 13.` An asset is protected by security patterns that fulfill the security requirements allocated to the target asset.
- `Rule 14.` The top goal is composed by a subject and a predicate
- `Rule 15.` The subject in the top goal of an assurance case should be the system
- `Rule 16.` The predicate in the assurance case top goal should be the EAL required to assure the system.
- `Rule 17.` In pattern arguments, each tool used to do an activity must appear in assumption about its correctness.
- `Rule 18.` Each Role that has done an activity should have competence assumed in an Assumption Node.

- `Rule 19.` The composition of patterns implies using a central pattern that coordinates between other patterns.

- `Rule 20.` All the uninstantiated attributes should appear in the reference data.

While some of these semantic rules may seem obvious to an assurance expert, they are relevant to many development teams and practitioners who may not be experts in assurance but want or need improved security assurance in their SDLC. Furthermore, these recommendations outline follow-on research directions that can stimulate the development of more effective security assurance solutions.

Note that this is not an exhaustive list of semantic rules. Moreover, these rules are not displayed. Instead, they are followed when the modeling language is used. Thus, OCL invariants are used to describe these rules. As an example, consider `Rule 17.` which relates the necessity to have an assumption about the correctness of the tool that supports an activity. The rule is formalized into the OCL constraint shown in Listing 4.1.

```
1  class pattern extends AssuranceModel{
2      invariant Rule6("The correctness of all the tools must be assumed"):
3  //Retrieve all the tools used for different security activities
4      let tools: Set(Tool) =
5      self.argueThroughActivity.use.excutedBy->select(t|t.oclIsKindOf(Tool)).oclAsType(Tool
       )->asSet(),
6  //Retrieve all the assumptions in the pattern
7      assumptions: Set(Assumption) = self.elements->select
8      (t|t.oclIsKindOf(Assumption)).oclAsType(Assumption)->asSet(),
9  //Check if the pattern assumes the correctness of the tools
10     correct : String = "is correct"
11     in tools->forAll (t | assumptions->exists(a:Assumption | a.description = t.toString()
        + correct )); }
```

Listing 4.1: OCL constraint for Rule 17

4.5 Concrete Syntax

Most metamodels and/or abstract syntaxes offer one or more concrete syntaxes to instantiate their concepts. The standards UML and SACM, for example, provide concrete syntaxes with diagrams for different viewpoints, in a graphical manner with icons and links. Other metamodels and especially domain-specific modeling languages often come with a textual syntax. To create model instances using SCML, we need to provide a concrete syntax. We choose to use an EBNF grammar to define a concrete syntax for the SCML language. Listing 4.2 gives an extract of the grammar. The textual grammar used to describe security, system, and process concepts are presented in Appendix B

```
1  AssurModel :=  Pattern | AssuranceCase | Template |FromPattern | FromTempl
2  AssuranceCase := "Assurance case" Id "{" metadata?  (AssurElem)+ "}"
3  Pattern := "Pattern" Id  "{" metadata? "secActivity" [Activity|Id]  params+ (AssurElem|
      Relationship)+ "}"
4  Template :=  "Template" Id "{" metadata?  "refArchitecture" [RefArch|Id] ("patterns" ([
      Pattern|Id])+ )?
5  params+ (AssurElem|Relationship)+ "}"
6  FromPattern := "Assurance case" Id  "application—of" ([Pattern|Id])+ ("using—data" [
      DataRef|Id])?  metadata? "{" (AssurElem)+ "}"
7  FromTempl :=  "Assurance case" Id "application—of"  [Template|Id] ("using—data" [DataRef|
      Id])? metadata? "{" (AssurElem)+ "}"
8  AssurElem := Goal | Assump | Justif | Strat | Solut | Cntxt
9  Relationship := ICO | ISB
10 Goal := top? "goal" Id toBeIns? toBeDev?  descrip  ("links {"  [AssurElem|Id]+ "}" )?
11 Cntxt := "context" Id  toBeIns? descrip
12 Assump := "assumption" Id  toBeIns? descrip
13 Justif := "justification" Id  toBeIns? descrip
14 Strat "strategy" Id  toBeIns? toBeDev? descrip ("links {"  [AssurElem|Id]+ "}" )?
15 Solut := "solution" Id  toBeIns?  descrip
16 ICO := "InContextOf" Id "lowerBound" bound  "upperBound" bound  "from" [AssurElem|Id] "to
      " [AssurElem|Id] comment
17 ISB := "IsSupportedBy" Id "lowerBound" bound "upperBound" bound  "from" [AssurElem|Id] "
      to" [AssurElem|Id]
18 bound := Int
19 top := "top"
20 metadata := "metadata" String
21 comment := STRING
22 toBeIns := "toBeInstantiated"
23 toBeDev := "toBeDeveloped"
24 params := "parameters {" Parameter+  "}"
25 descrip := TextDescription (TextDescription|ParameterUse)*
26 ParameterUse :=   " + {" Parameter "}"
27 Parameter := [Concept|ID]"."Attribute
28 Concept := SecurityConcept | ProcessConcept | SystemConcept
29 Attribute := enumeration
30 TextDescription := String
```

Listing 4.2: Excerpt of the textual grammar of SCML

The axiom rule in Line (1) of Listing 4.2 allows the creation of different assurance models. It involves an assurance case, pattern, template, an assurance case from the application of patterns, and an assurance case from the application of a template. Note that the grammar recalls the fact that patterns and templates are arguing through activities and reference architectures, respectively. The patterns and templates contain assurance elements with parameters and relationships to be instantiated. In contrast, the assurance cases contain only instantiated assurance elements. In Line (26), the rule ParameterUse indicates that the names of parameters are defined by combining the name of a concept with the name of an attribute. The concepts are defined in the presented packages: Security, Process, and System. For example, a parameter that refers to the name of an

activity is called: *Activity.name*.

Finally, to sum up, according to [25], the structure of SCML modeling language is represented in Figure 4.6 as follows.

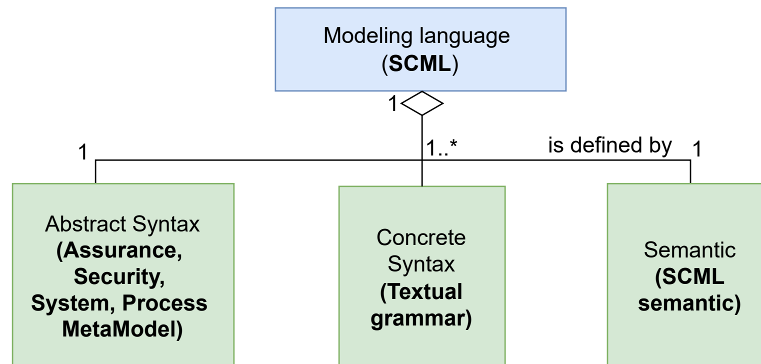


Figure 4.6: SCML structure adapted from [25]

4.6 Conclusion

In this chapter, we developed the Security Cases Modeling Framework in-depthh, covering its key components and features. The discussion began with the abstract syntax, which serves as the backbone of the modeling framework. The chapter continued with an examination of the **semantic aspects** of the language, divided into **static** and **dynamic semantics**. Finally, the **concrete syntax** was discussed, detailing how the abstract concepts are represented in a tangible and usable form using textual grammar.

In the following chapters, we will use the framework to create security cases and reusable security argument models.

Chapter 5

Methodologies and Tools for the creation of Security Cases

Contents

5.1 Introduction **74**

5.2 Inputs for the methodologies **75**

5.3 Creating Security Cases from Scratch **76**

5.3.1 Security case definition 77

5.3.2 Tool support 80

5.4 Creating Security Cases using Argument patterns **81**

5.4.1 Argument Pattern definition 82

5.4.2 Pattern application 85

5.4.3 Tool support 88

5.5 Creating Security Cases using Argument templates **90**

5.5.1 Argument Template definition 90

5.5.2 Template application 91

5.5.3 Tool support 92

5.6 Conclusion **93**

5.1 Introduction

In the context of the modeling framework, we propose a set of methodologies to generate security assurance cases. Firstly, the creation of security cases from scratch is advocated, offering a tailored approach to address unique security challenges and system architectures. Secondly, we delve into the methodology of generating security cases by reusing predefined patterns. Lastly, we explore the methodology of creating security cases through the reuse of domain-specific templates, allowing for customization to industry or organizational requirements and the targeted mitigation of domain-specific security issues.

Each of the methodologies is a sequence of steps that requires some inputs and outputs either an argument pattern, an argument template or a security case. The inputs and outputs of the methodologies are modeled using the SCML framework described previously. While defining the methodologies, we wanted to show the step where each element of our security case was defined. Thus, the words written in bold refers to the SCML concepts taken from GSN. This will facilitate both the definition of future security cases and the understanding and analysis of security cases already defined to make a possible modification.

Moreover, to automatically support these methodologies, we used EMF (Ecore) and Xtext to develop the DSL metamodel. We also used OCL to encode the corresponding formal metamodel. We developed a textual editor to model the architecture, the security, and the engineering activities of the system using EMF and Xtext. In addition, the textual editor models the security cases, the patterns and the templates. Furthermore, we aim to develop two transformation engines to generate the graphical and tabular reports.

The remainder of this chapter is organized as follows. Section 5.2 presents the inputs of required by the methodologies. Section 5.3 presents the primary use of the framework, involving the creation of security assurance cases through the instantiation of the metamodel. Section 5.4 transitions to the second methodology, enabling the creation of security cases by reusing existing argument patterns. Section 5.5 the final alternative for generating security cases is presented. This section elucidates how the framework facilitates the definition of domain-dependent argument templates, subsequently allowing the reuse of these templates for creating assurance cases. Finally, Section 5.6 concludes the chapter, summarizing key findings and highlighting the significance of the presented methodologies in the broader context of security analysis.

5.2 Inputs for the methodologies

In this first section, we outline the key inputs that inform and guide the methodologies utilized in this study. These inputs include the software architecture models, the security models, reusable patterns and templates...etc. Understanding these inputs is crucial for appreciating the rationale behind the chosen methods and their relevance to the research objectives. The inputs are described in the order in which they appear in the dissertation.

Secure software architecture model It represents the model of the software architecture extended with the security mechanisms that mitigate the identified security threats. It is described using the system and the security packages. Listing 5.1 describes the secure software architecture model of a toy example system that will be used as input for the methodologies described later. It is composed of two components connected by a connector. The software assets are protected using security patterns that prescribe security mechanisms.

```
SoftwareArchitecture ToyExmpl "This is a description of a toy example secure
    architecture"
    relateTo ToyExample
    use {SoftwareAsset Asset1 type "component" protectedBy SM1
        SoftwareAsset Asset2  type "component"  protectedBy SM2, SM3
        SoftwareAsset connect1   type "connector" connects Asset1 , Asset2
            protectedBy SM3, SM2
    }
```

Listing 5.1: Secure Architecture of a toy example

Secure architecture development activities These are the activities that aim to analyze and evaluate the security of the software architecture. Each activity is described using the process package. Recalling the toy example, the threat mitigation activity aims to find security patterns that protect the software. The activity is presented in Listing 5.2.

```
Activity ThreatMitigation
    goal "Mitigate the threats to protect assets from potential vulnerabilities"
    justification: "The security vulnerabilities targeted by the attackers exploit
        the security threats"
    phase ArchitecturalDesign
    type Analysis
    require {SoftwareArchitecture , ThreatLists}
    produce {SecureSoftwareArchitecture , SecurityPatterns}
    carriedBy "Architect"
Artifact SoftwareArchitecture
    link "Path/To/Artifact/SoftwareArchitecture.scml"
    type Documentation
    describeAsset SoftwareArchitecture
```

```
    language SCML
Artifact SecurityThreats
    link "Path/To/Artifact/SecurityThreats.scml"
    type Documentation
    language SCML
```

Listing 5.2: Threat mitigation activity

Secure software architecture artifacts These models refer to the concrete artifacts used and/or produced by the activities aiming to secure the software architecture. Among the artifacts used for the security engineering of the toy example, we find the artifact describing the security requirements. An excerpt of the artifact is presented in Listing 5.3.

```
    Requirement SR1
    mitigate  spoofing
    fulfilledBy SM1
    allocatedToAsset Asset1
    formalreq "formalSR1"
    Requirement SR2
    mitigate tampering
    fulfilledBy SM2, SM3
    allocatedToAsset  Asset2, connect1
    formalreq "formalSR2"
```

Listing 5.3: List of the security requirements of the toy example system

Security argument patterns These are the reusable patterns already defined and ready to be reused.

Secure reference architecture It represents the model of a reference architecture extended with the security solutions that mitigate the identified security threats. It is described using the system and the security packages.

Security argument templates These are the reusable templates already defined and ready to be reused.

5.3 Creating Security Cases from Scratch

The idea of the first methodology consists of creating security cases from scratch, without relying on pre-existing patterns. The novelty of this methodology is that it has been adapted to our modeling framework. First, the modeling framework allows the description of the methodology inputs and outputs, e.g., the architecture model and the activities. Then, the methodology steps are supported within the SCML tool support.

The rest of the section is organized as follows. Section 5.3.1 presents the methodology. Then, Section 5.3.2 presents the tool support prototype architecture used to create security cases.

5.3.1 Security case definition

In the following, we detail the different steps of this methodology, illustrated through a toy example described in the previous section. It typically occurs in the following order, the numbers in parentheses corresponding to the numbers in the Figure 5.1.

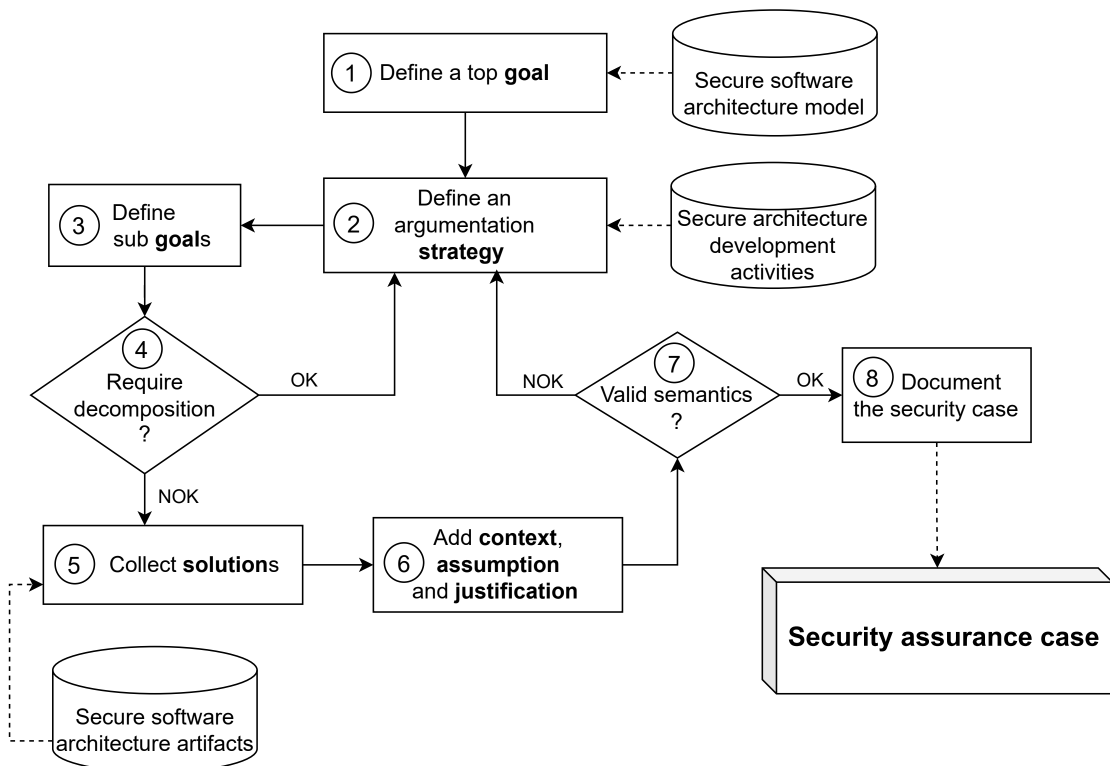


Figure 5.1: From Scratch methodology overview

- **Step(1):** we define the top goal to achieve. It refers to the security assurance target. The top goal is composed of a subject and a predicate. The subject refers to the assurance target which is either the whole software architecture or only some assets of it. The predicate describes the security concern, e.g. being protected against the attacks, fulfilling the security requirements. Recalling the toy example, a top

goal that claims the security of the entire software architecture model is *the system software architecture is secure according to EAL4*, where the subject is *the system software architecture* and the predicate is *being secure according to EAL4*. The top goal is the most distinguishing element of the security case. The security case body will be defined to attend the top goal.

- **Step(2):** we define an argumentation strategy. It is the most important step as it explains the rationale justification to support the top goal. It is affected by the activities that have been done to design a secure architecture. Multiple strategies can be followed to achieve the same top goal. In our methodology, we propose to argue using the security engineering activity that is supposed to attend to the top goal predefined earlier. Recalling the toy example, *the threat mitigation activity leads to an argumentation strategy based on mitigated threats.*

- **Step(3):** we define the sub-goals that are implied by the argumentation strategy. Following the toy example, the activity chosen as an argumentation strategy is *threat mitigation*. It leads to mitigating the tampering and the spoofing threats. Consequently, we will have two sub-goals. *The sub-goal(1): The tampering threat was mitigated. The sub-goal(2): The spoofing threat is mitigated.*

- **Step(4):** we decide if the sub-goals need decomposition into more manageable and verifiable sub-goals, thereby facilitating a clear and comprehensive argument for the system's assurance. If there is a need for a decomposition of the sub-goals, we go back and define a new argumentation strategy (**Step(2)**. Recalling the toy example, *the identified sub-goals are verifiable and do not require more decompositions.*

- **Step(5):** when there is no need for more decomposition, we collect the evidence elements that are required to fulfill the last decomposed sub-goals. The evidence elements are collected among the artifacts produced while designing the system software architecture. Recalling the toy example, *the threat mitigating activity proposes using security patterns to mitigate each threat. We collect the artifacts that describe the security patterns and we use them as evidence elements to support the security case.* Security patterns that mitigate the tampering threat support the sub-goal(1). However, security patterns that mitigate the spoofing threat support the sub-goal(2).

- **Step(6):** we add the necessary information that reports on the context in which the system under assurance is developed. We also add some assumptions that we made

about the system development, without which the argumentation in the security case does not hold. Sometimes, following the reasoning presented in the security case takes a lot of work. Thus, we must add some justifications to make the argument more convincing. Recalling the toy example, we add a context element that refers to *the software architecture model*.

- **Step(7):** we check the semantic rules of the created security case (as defined in Section 4.4.1). The validation starts after the goals have been identified and the evidence is assigned to the corresponding goals. The result of this step might require going back to the point where we define the argumentation strategy. From **Step(2)**, we go through each step and correct the error. For example, *if the security case lacks a context element to list the identified threats, we go back to Step(2). We add a context element that refers to the list of the threats outputted as artifact by the threat mitigating activity.*

- **Step(8)** we document the created security case. For example, *we give some additional information: the system under assurance, the notation and the tool used to describe the security case..etc.*

The toy example security case is depicted in Listing 5.4.

```
Assurance case   AC_example {
metadata "This is a security case created from scratch for the toy example system"
top goal G0 "The toy example is secure according to EAL4" links{S0, C0, C1}
context C0 "Description of the toy example software architecture in Listing 2.1"
context C1 "List of security requirements in Listing 2.3"
strategy S0 "Argue through mitigating the threats" links {G1, G2, J0, C2, C3}
context C2 "Description of the threat mitigation activity in Listing 2.2"
context C3 " List of the threats"
justification J0 "The security vulnerabilities targeted by the attackers exploit the
    security threats"
goal G1 "The spoofing threat is mitigated" links{Sn0, C4}
solution Sn0 "Security mechanism SM1 mitigates the spoofing threat"
context C4 " Description of the security pattern within the mechanism SM1"
goal G2 "The tampering threat is mitigated"  links{Sn1, C5}
solution Sn1 "Security mechanisms SM2 and SM3 mitigate the tampering threat"
context C5 " Description of the security pattern within the mechanisms SM2, SM3"
}
```

Listing 5.4: A security case for the toy example system

The security case documentation tells that the security case is written within SCML framework, and this is the version of September 2024.

5.3.2 Tool support

We have implemented a prototype to support the approach as an Eclipse plug-in. Our starting point is the security assurance metamodel presented in Section 4.3.

Figure 5.2 depicts the prototype tool-chain support architecture used for creating security cases from scratch, comprising the following two primary blocks: 1) Modeling framework block and 2) Application development block. These are detailed in the following sub-sections, with their related tasks numbered in parentheses.

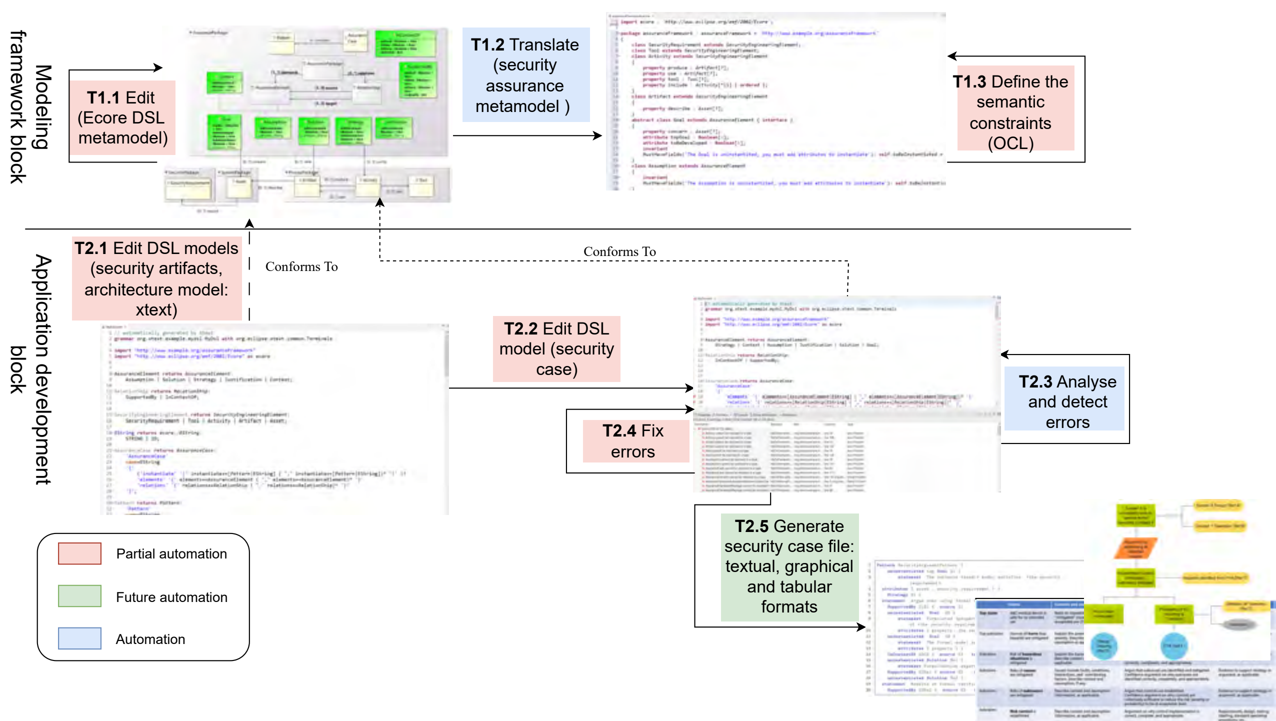


Figure 5.2: Tool support architecture and artifacts used for the scratch methodology

5.3.2.1 Modeling Framework Block

The first block is dedicated to supporting the following three tasks.

- *(T1.1)* Edition of the DSL metamodel in Ecore.
- *(T1.2)* Translation of the Ecore metamodel into xtext grammar referencing the Ecore metamodel.
- *(T1.3)* Definition of the OCL constraints that express the semantic rules.

5.3.2.2 Application Development Block for Reuse

The second block is dedicated to support five tasks.

- *(T2.1)* Edition of the DSL models that describes the inputs of the methodology presented in section 5.2, namely, the security software architecture model, the security engineering activities, and the architecture artifacts. The textual models must conform to the DSL metamodel.

- *(T2.2)* Edition of the security case following the steps presented in the previous section 5.3. The security case must conform to the metamodel.

- *(T2.3)* Analyse of the security case and detection of the errors that result from the violation of the OCL constraints.

- *(T2.4)* Correction of the detected errors.

- *(T2.5)* Generation of the final security case after correcting all the errors. Currently, the tool allows a textual-based presentation of the security case. In future works, we aim to implement a tabular and graphical representation of the security case.

5.4 Creating Security Cases using Argument patterns

The central idea of the second methodology is to enable the creation of security cases using argument patterns. Given that Gamma et al. [41] define design patterns as solutions to a recurrent problem in software system design that helps in improving reusability, maintainability, comprehensibility, evolvability, and robustness of legacy applications. In our work, we define argumentation patterns as solutions to the recurrent problem of the need for a compelling argument that argues about the efficiency of security engineering activities, particularly those conducted at the architectural design level. Therefore, we propose a methodology to define security argument patterns for each security engineering activity. Moreover, we define a methodology for the creation of security cases using argument patterns. Both of the methodologies are supported by a tool.

The rest of the section is organized as follows. We first need to define a catalogue of argument patterns following the methodology presented in Section 5.4.1. Then, we select and compose a set of patterns and apply them to create security cases as presented in Section 5.4.2. Finally, we present the tool support prototype architecture used to define argument patterns and to reuse them to create security cases as presented in Section 5.4.3.

5.4.1 Argument Pattern definition

.

In Figure 5.3, we detail the steps to create security argument patterns. We will run the methodology on a security engineering activity used to describe the vulnerability testing activity. More examples will be presented in the patterns catalogue presented in Section 6.2.

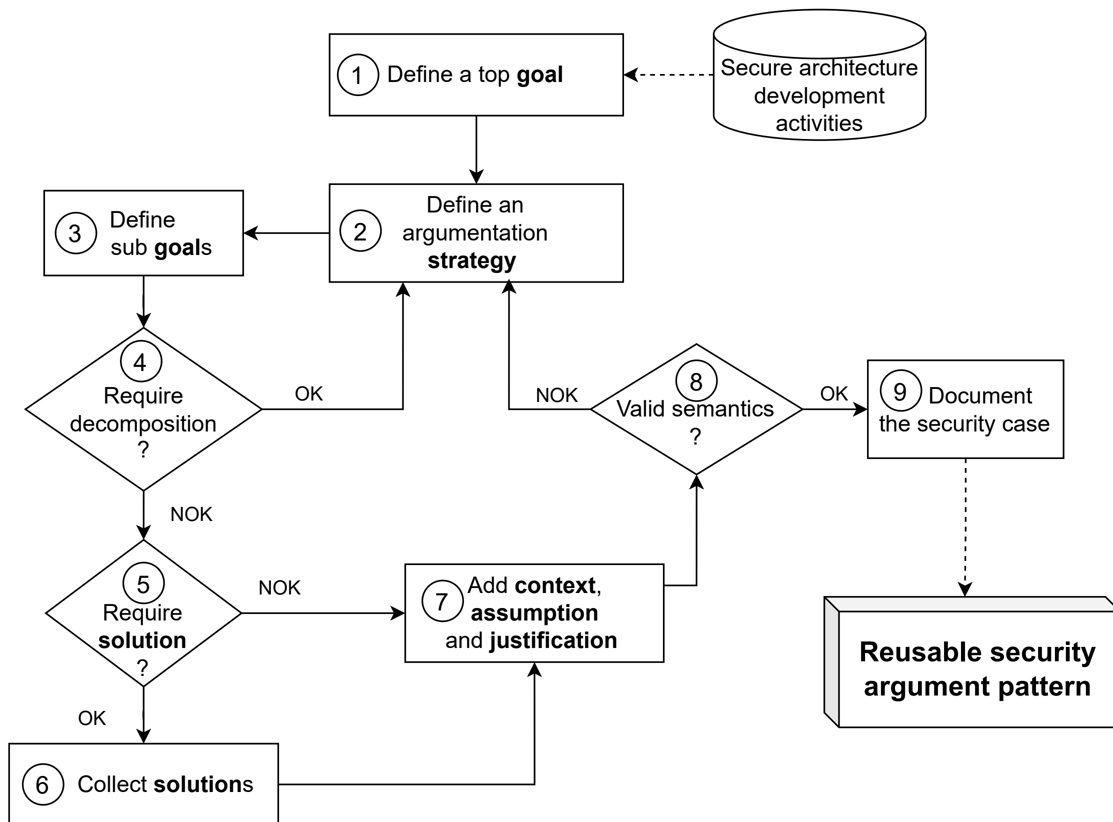


Figure 5.3: Pattern definition process

- **Step(1):** we define the top goal of the pattern. The goal of the activity inspires the top goal argued in the argument pattern. For example, *the goal of the vulnerability mitigation activity is to address the vulnerabilities in a software design artifact. According to our methodology, a top goal of the argument pattern that argues over this activity is that the software architecture is protected from vulnerabilities.*

- **Step(2):** we define an argumentation strategy. The first strategy in our argument patterns argues through conducting the security engineering activity. For example, *the first strategy that supports the top goal defined previously is to argue through conducting a vulnerability testing activity.* We can parametrize this strategy by adding the parameter {methodId} that refers to the name of the vulnerability testing method. Further strategies can be defined to demonstrate that the satisfaction of the sub-goals implies the satisfaction of the top goal.

- **Step(3):** we define the sub-goals that are implied by the argumentation strategy. Usually, the sub-goals refer to other activities included in the main activity indicated in the strategy. For example, *the vulnerability testing activity requires identifying all the assets, identifying all the vulnerabilities, and mitigating them. Therefore, we define three sub-goals:(1) All the assets are identified, (2) All the vulnerabilities are identified, and (3) All the vulnerabilities are mitigated.* For abstraction seeks and according to GSN pattern notation, the expressions written between braces will be replaced by concrete information depending on the system under assurance.

- **Step(4):** we decide if the sub-goals need decomposition into more manageable and verifiable sub-goals. If there is a need for a decomposition of the sub-goals, we go back and define a new argumentation strategy **Step(2)**. Recalling the example, *the first and second sub-goals are verifiable and do not require more decompositions. However, the third sub-goal needs more decomposition.*. We go back to **Step(5)** and define a new strategy to argue about mitigating each vulnerability. After, we move to **Step(3)** to define the sub-goal: each vulnerability is mitigated. Note that the GSN pattern Notation allows us to define an abstract relationship linking the strategy and the sub-goal. This relationship is parameterized with {n: the number of vulnerabilities}. Finally, we return to **Step(4)** to check if the last sub-goal requires more decomposition.

- **Step(5):** we decide if the pattern focuses only on the argumentation part or includes also the evidence elements. Following our example, *we need to include evidence elements to support the goals presented in the pattern.* An example of a pattern focused only on the argumentation part is presented in [63].

- **Step(6):** when the pattern is not only argument-centric, we define the evidence elements that are required to fulfill the sub-goals. The artifacts produced by the activities described in the argument pattern will be referenced in the solutions as

evidence elements to support the sub-goals. *Vulnerability Scanning Logs, Mitigation Plan* are examples of the artifacts to refer to as evidence.

- **Step(7):** we add the context, justification, and assumption necessary to make the argument more convincing, as described in **Step(6)** from the previous methodology in Section 5.3.1. Examples of the elements we can add are: *Context: the list of vulnerabilities, Assumption: the tools used are correct..*

- **Step(8):** we check the semantics properties of the pattern precisely according to GSN pattern notation. If the pattern is not validated, we return to **Step(2)** and redefine the pattern. For example, *The decorator "uninstantiated" is only used under the goals with parameters to instantiate (the expressions between braces).*

- **Step(9):** we document the created security pattern by defining the underlying intent and the constraints on its use. The template for describing design patterns used in [41] is also used to describe the security argument patterns in this report. The sections in the descripton are the following.

Name The pattern's name should succinctly convey the pattern's essence. A good name is vital because, with use, it will become part of your design vocabulary.

Intent A short statement that answers the following questions: What does the pattern do/represent? What is its rationale and intent? What particular security issue / requirement / process does it address?

Motivation A scenario that illustrates a security issue / process and how the elements of the goal structure solve the problem. The scenario will help us understand the more abstract description of the pattern that follows.

Applicability A general description of the situations in which the security argument pattern can be applied, the information required as context for the pattern to be successful (necessary inputs to the pattern).

Structure A textual or graphical representation of the pattern using the SCML notation. The structure indicates generality or optionality it should be clear how the pattern can be instantiated.

Consequences It describes the benefits provided by the pattern and any potential liabilities, the trade-offs and results of using the pattern.

Herein is an excerpt of the vulnerability mitigation argument pattern from the previous methodology presented in Listing 6.2.

```
Pattern VulnerabilityMitigation {
    metadata "See the following section for the description of the pattern"
    secActivity Vulnerability Mitigation
    parameters {Vulnerability.name, Threat.class, Vulnerability.description, Method.name
        , Tool.name, Activity.require, Activity.description, Artifact.link }
    top goal G1 toBeInstantiated "The software architecture"+ {SoftwareArchtiecture.name
        } + "is protected from the vulnerabilities" links{C1, C2, S1}
    strategy S1 "Argue through conducting vulnerability testing method" + {Method.name}
        links{J1, A1, G2, G3, G4}
    context C1 toBeInstantiated "Vulnerability testing description" + {Activity.
        description}
    context C2 toBeInstantiated "The software architecture model description" +{Activity
        .require}
    assumption A1 toBeInstantiated "The tool" + {Tool.name} + "is correct"
    justification J1 "Mitigating the vulnerabilities allow fulfilling the security
        requirements, thus securing the software architecture"
    goal G2 "All the assets are identified" links{Sn1}
    solution Sn1 toBeInstantiated "Asset Inventory results" + {Artifact.link}
    goal G3 "All the vulnerabilities are identified" links{Sn2}
    solution Sn2 toBeInstantiated " Vulnerability scanning log" + {Artifact.link}
    goal G4 "All the vulnerabilities are mitigated" links{S2}
    strategy S2 toBeInstantiated"Argue through each vulnerability" links{G5.X}
    goal G5.X toBeInstantiated toBeDeveloped "Vulnerability" + {Vulnerability.name} + "is
        mitigated"
    IsSupportedBy ISB1 lowerBound 1 from S2 to G5.X /*The upperBound is equal to the
        number of vulnerabilities */
}
```

Listing 5.5: Vulnerability mitigation argument pattern

More complete examples of patterns are presented in the catalogue in the next chapter 6.2.

5.4.2 Pattern application

In Figure 5.4, we describe the methodology of creating security cases by re-using argument patterns. It is composed of three parts. *Pattern selection* allows reusing one or more patterns to construct the security case through (**Step(2), Step(3), Step(4), Step(10)**). *Pattern editing* allows adding some assurance elements to link the selected patterns through (**Step(1), Steps 5–9**). *Pattern application* enables the choice of a manual application of the patterns or an automatic one using the reference data(**Steps 11–15**). For brevity, we will exemplify the methodology in the evaluation section (see Section 7.2).

- **Step(1):** we define a top goal.
- **Step(2):** we check if there is a pattern that fits with this top goal. A pattern that fits with the assurance's goal should have a top goal that asserts the same goal as

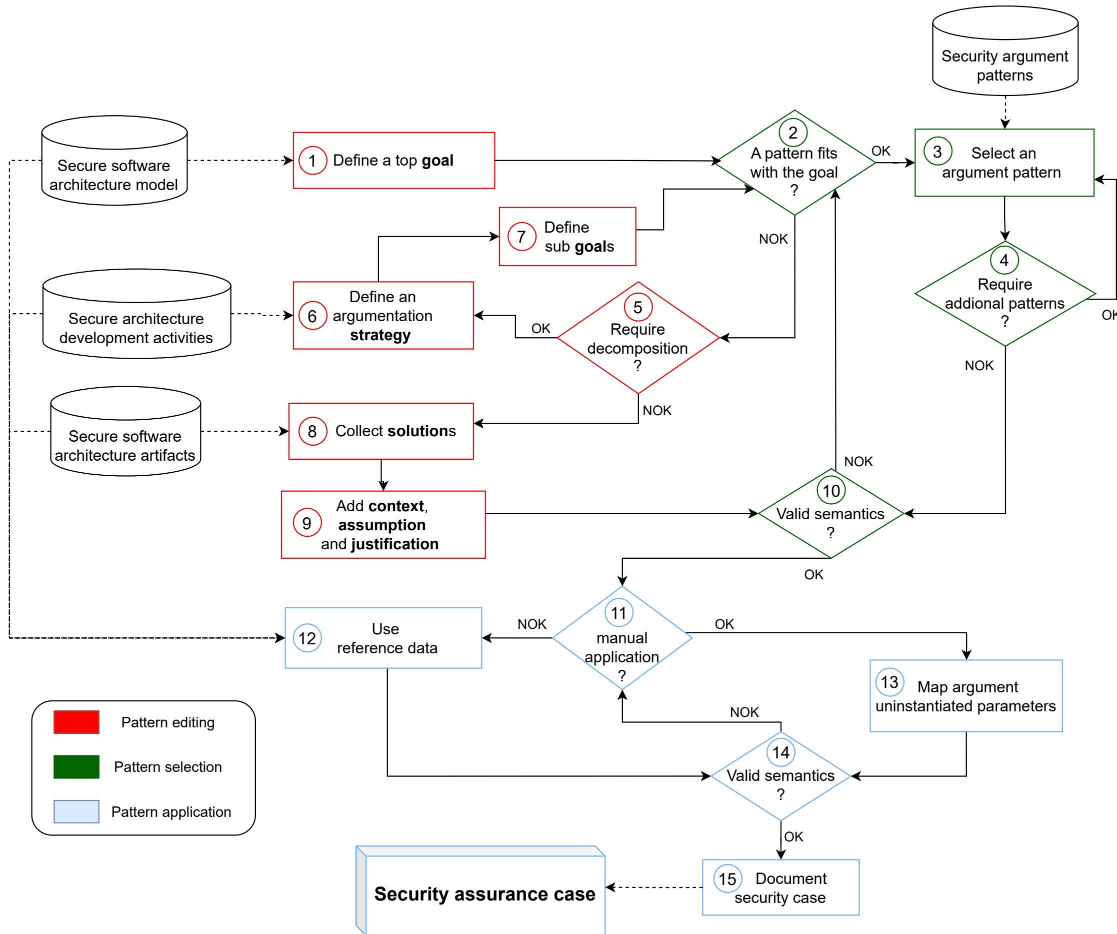


Figure 5.4: Pattern application methodology overview

the goal within the assurance. Otherwise, we move to **Step(5)**.

- **Step(3):** we select an argument pattern from the predefined catalogue of patterns. The selection is based on the understanding of the assurance developer.

- **Step(4):** we check if the selected pattern requires additional patterns to argue about some undeveloped nodes. Otherwise, we move to **Step(10)**. For example, *the pattern (P1) has an undeveloped goal that claims the identification of all the vulnerabilities. There is another pattern (P2) that argues about an activity that aims to identify the vulnerabilities. The selection of the pattern (P1) in **Step(3)** requires going back to the catalogue to also select the pattern (P2).*

- **Step(5):** when no pattern fits with a goal in the catalogue, we develop this goal

following the same steps indicated in the two previous methodologies. We start by checking if this goal requires a decomposition. Otherwise, we collect solution elements in **Step(8)**.

- **Step(6):** we define an argumentation strategy.
- **Step(7):** we define sub-goals. For each sub-goal, we go back to **Step(2)** to check if it fits with a predefined pattern.
- **Step(8):** we collect the solutions that support the goal that does not need more decompositions.
- **Step(9):** we add the context, assumption, and justification elements.
- **Step(10):** We check the semantics properties of the composed and edited argument patterns. The result of this step might require going back to pattern selection first step (**Step(2)**). Examples of the errors that can be detected at this level are *renaming conflicts (two repetitive elements from different patterns), mis-matching patterns (select a wrong required pattern)*
- **Step(11):** Once the structure of the composed patterns is validated, we decide if we will apply it manually or use reference data.
- **Step(12):** If we choose to use reference data, we automatically generate a reference data tree, as described in Algorithm 1 and fill it in by using the information from the artifacts.
- **Step(13):** If we select the manual application, we map uninstantiated parameters to concrete information from the concrete system architecture and the security architecture artifacts. More concretely, the uninstantiated Assurance elements will have their parameters mapped to concrete elements from the secure architecture artifacts. For example, the parameter *Method.name* will be mapped to a concrete method, e.g., model checking, theorem proving..etc.
- **Step(14):** We recheck for the security case. The second validation is based on application concerns. For example, *all the parameters are mapped to the correct type of data*. If the validation fails, we return to **Step(11)**.
- **Step(15):** we document the security case. We indicate, for example, *the patterns used to construct the security case, the reference data, if any, used to generate the security case..etc*

5.4.3 Tool support

We have implemented a prototype to support the approach as an Eclipse plug-in. Our starting point is the security assurance metamodel presented in Section 4.3.

Figure 5.5 depicts the prototype tool-chain support architecture used for creating security cases from scratch, comprising the following two primary blocks: 1) Modeling framework block and 2) Application development block. These are detailed in the following sub-sections, with their related tasks numbered in parentheses.

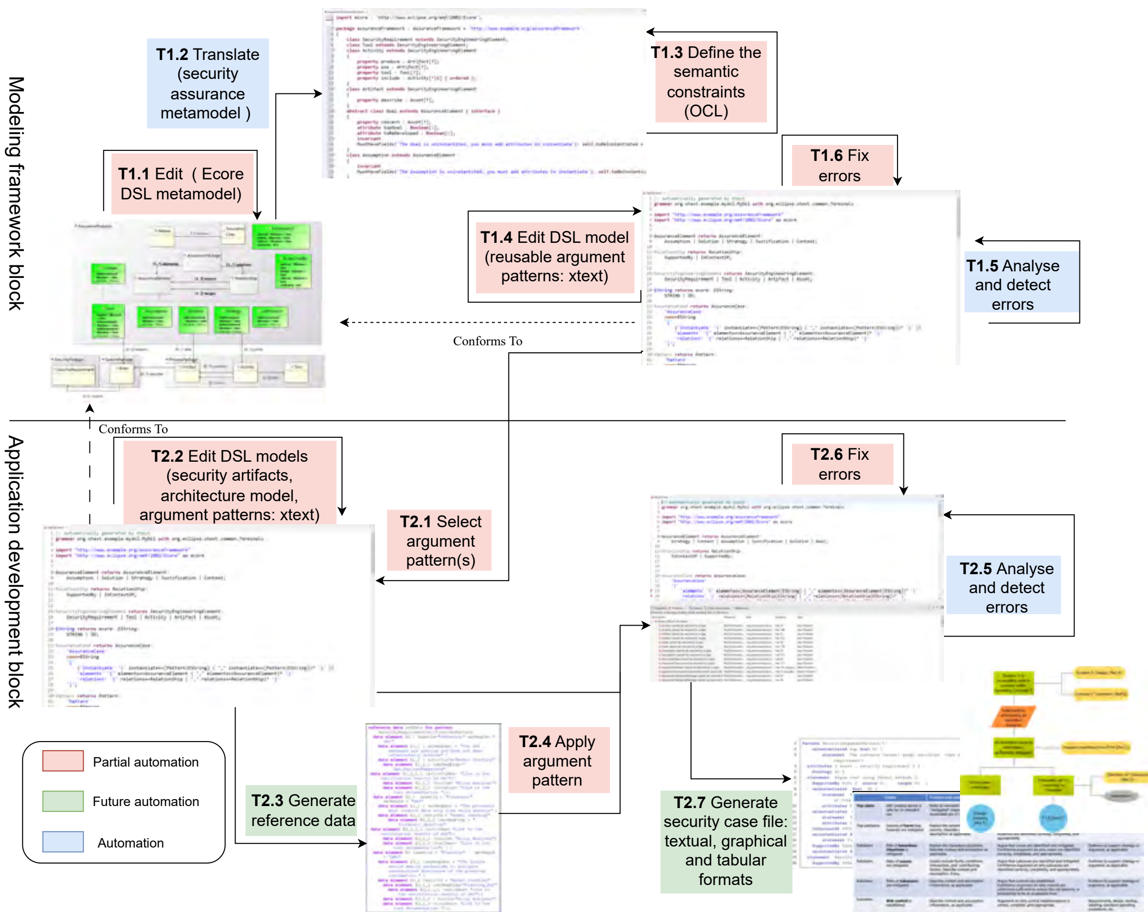


Figure 5.5: Tool support architecture and artifacts used for creating and applying patterns

Modeling Framework Block for Reuse

The first block is dedicated to support the following six tasks. The tasks *(T1.1),(T1.2),(T1.3)* are common to the modeling block in Section 5.3.2.1. We focus here on the tasks that

concern the modeling argument patterns.

- *(T1.4)* Edition of the DSL models of reusable argument patterns that are defined according to the methodology discussed in Section 5.4.1. The patterns are edited using the xtext-based editor and stored as textual files.
- *(T1.5)* Analyse and detection of the errors that result from the violation of the semantic rules of the modeling language. The detected errors are mainly concerned with the structure of the pattern.
- *(T1.6)* Correction of the detected errors.

Application Development Block for Reuse

The second block is dedicated to supporting six tasks.

- *(T2.1)* Selection and importation of the argument patterns that will be used to generate the security case.
- *(T2.2)* Edition of DSL models that describe the inputs of the methodology presented in section 5.2. As seen in the pattern application methodology, we can also edit the selected patterns to compose a big assurance case. The textual models must conform to the DSL metamodel.
- *(T2.3)* Generation of reference data that will be used to automatically apply the patterns. Currently, the tool automatically creates the tree-based structure of the reference data. However, it does not automatically extract the data from the DSL models, e.g. the architecture models, the security artifacts...
- *(T2.4)* Application of argument patterns manually or automatically using the generated reference data.
- *(T2.5)* Analyse of the security case and detection of the errors. The errors are concerned with the structure and the application of the patterns.
- *(T2.6)* Correction of the detected errors.
- *(T2.7)* Generation of the final security case after correcting all the errors. Currently, the tool allows a textual-based presentation of the security case. In future works, we aim to implement a tabular and graphical representation of the security case.

5.5 Creating Security Cases using Argument templates

The idea of the third methodology is to enable the creation of security cases using argument templates. The argument template will be defined for a specific domain application or technology development. In our work, we describe the domain application using a secure reference architecture. To achieve this, we first need to define a catalogue of argument templates as presented in Section 5.5.1. Then, we select a template and apply it to create security cases as presented in Section 5.5.2.

5.5.1 Argument Template definition

We describe the methodology of creating argument templates which consists of three parts. Most of the steps have already been covered in the pattern application methodology. This final methodology inherently involves the creation of a template, as it defines the template as a system of patterns. In Figure 5.6, we detail the steps to create security argument templates.

- **(Steps 2–4), Step(10):** These steps are responsible for the *Pattern selection*. They allow the selection of the patterns reused to compose the template. The template in this methodology is seen as a system of pattern or a composite pattern. The crucial input of the methodology includes a secure reference architecture model that will be the assurance target.

- **(Step(1), Steps 5–9):** These steps are responsible for the *Template edition*. They allow the definition of assurance elements that link the patterns. Noting that, some patterns can be edited and adapted to the domain application inputted to the tool via the reference architecture model. For example, we want to define an argument template for medical devices. The threat modeling is declared as a secure architecture development activity. We select an argument pattern that argues about the threat modeling activity. However, the patterns are domain-independent and need some adaptations to the domain application, namely medical devices.

- **Step(11)** It concerns the *Template documentation*. It specifies, for example, the patterns involved in the template construction and the emplacement of the inputs used by the methodology.

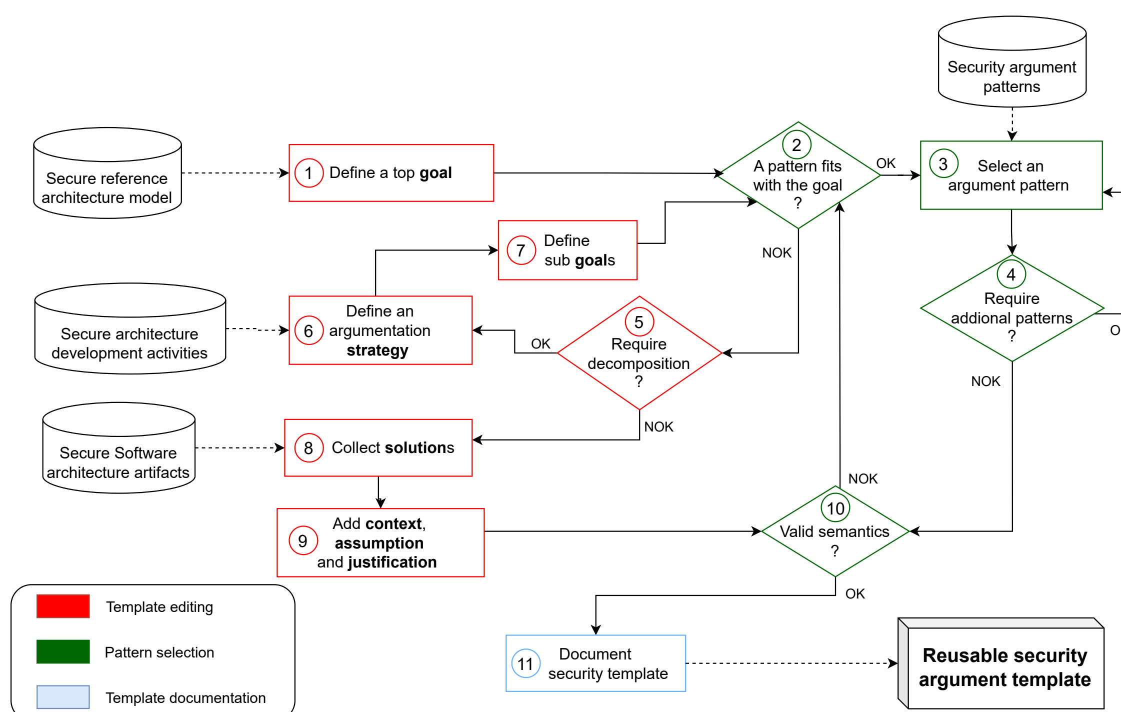


Figure 5.6: Template definition process

5.5.2 Template application

In Figure 5.7, we describe the methodology of creating security cases by re-using a domain-dependent template.

- **Step(1):** we select an argument template from the predefined catalogue of templates. The choice is determined based on the system under assurance. We can select the template that argues about a specific technology used in developing the target system, e.g., cloud and machine learning technologies ... Another possibility is to select the template that argues about a specific domain, e.g., medical and railway systems...
- **Step(2):** we decide if we want a manual application of the template so that we move to **Step(4)**. Otherwise, we move to **Step(3)** for an automatic application of the template.
- **Step(3):** we generate the reference data related to the template.
- **Step(4):** we map manually each uninstantiated parameter in the template.

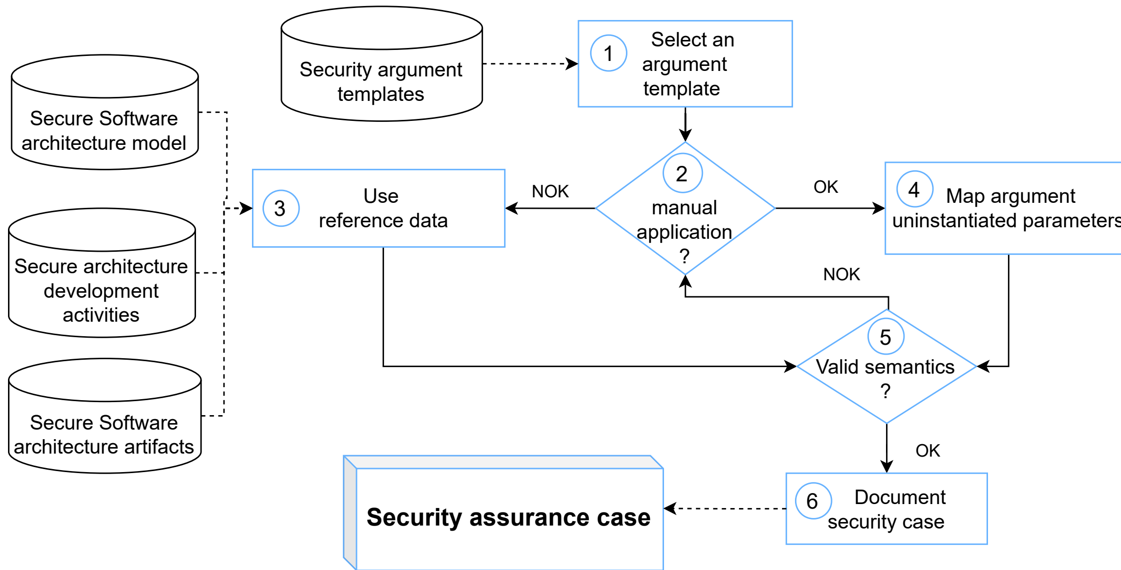


Figure 5.7: Template application methodology overview

- **Step(5):** the validation in the process of creating assurance cases based on templates concerns the structure of the template and the content of the security case.

- **Step(6):** we document the resulted security case.

5.5.3 Tool support

Figure 5.8 depicts the prototype tool-chain support architecture used for creating security cases from scratch, comprising the following two primary blocks: 1) Modeling framework block and 2) Application development block. These are detailed in the following subsections, with their related tasks numbered in parentheses.

Modeling Framework Block for Reuse

The first block is dedicated to supporting six tasks. Most tasks are common to the modeling blocks in Section 5.3.2.1 and Section 5.4.3. The task *(T1.4)* edits the argument templates as DSL models that must conform to the DSL metamodel defined within the tasks *(T1.1)*,*(T1.2)* and the OCL constraints defined within the task *(T1.3)*. These OCL constraints help detect errors *(T1.5)*. A consistent template has no error *(T1.6)*.

Application Development Block for Reuse

The second block is dedicated to support seven tasks.

- *(T2.1)* Selection and importation of the argument template from the predefined templates, based on the system under assurance.
- *(T2.2)* Edition of DSL models that describe the inputs of the methodology presented in section 5.5. The edition of the selected template allows more adaptation of the template to the assurance target. The textual models must conform to the DSL metamodel.
- *(T2.3)* Generation of reference data that will be used to automatically apply the template.
- *(T2.4)* Application of argument template manually or automatically using the generated reference data.
- *(T2.5)* Analyse of the security case and detection of the errors. The errors are concerned with the structure and the application of the template.
- *(T2.6)* Correction of the detected errors.
- *(T2.7)* Generation of the final security case after correcting all the errors. Further implementation is required to support the graphical and tabular representations of the security cases.

5.6 Conclusion

In this chapter, we presented three methodologies for creating security cases using the modeling framework presented in the previous Chapter 4. The first methodology allows the creation of security cases from scratch. The second allows defining reusable argument pattern and their application to create security cases. The third methodology allows the definition of a reusable argument template specified for a particular domain application. Each of the methodologies is supported with some automated, partially automated tasks.

Furthermore, we walked through an MDE-based prototype to support the proposed approach. An example of this tool suite is constructed using EMFT, Xtext and is currently provided in the form of Eclipse plug-ins.

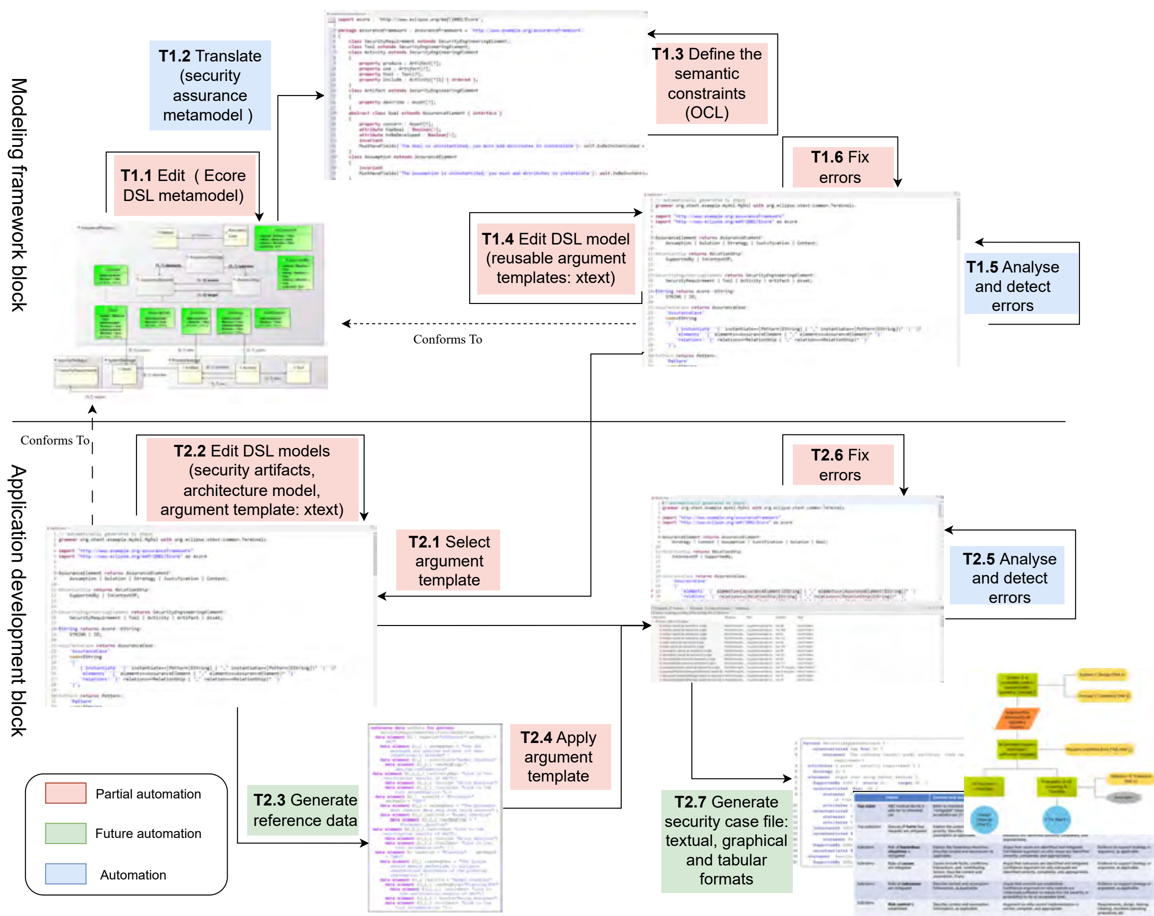


Figure 5.8: Tool support architecture and artifacts used for creating and applying templates

Chapter 6

Catalogue of reusable security argument patterns and templates

Contents

6.1 Introduction **95**

6.2 Security argument patterns **96**

6.2.1 Secure software architecture argument pattern 96

6.2.2 Argument pattern for threat modeling using STRIDE 98

6.2.3 Argument pattern for threat modeling using PASTA 101

6.2.4 Argument pattern for model checking 104

6.2.5 Argument pattern for well-formedness of the model 106

6.2.6 Summary 109

6.3 Reusable argument templates **110**

6.3.1 Template for SCADA 110

6.3.2 Template for MLBS 120

6.3.3 Summary 128

6.4 Conclusion **129**

6.1 Introduction

To implement the methodologies presented in the previous chapter 5, we have to define a comprehensive catalogue of patterns and templates: providing that is one of the main

objectives of this thesis. We use the pattern template provided by [66] for describing argument patterns. The template is adapted from the Gang of Four template [41]. A related aspect is that of how to compose these patterns to define reusable argument templates.

Overall, the chapter is organized and divided into two big catalogs. Section 6.2 presents a selection of security argument patterns, each arguing about one security engineering activity that is held in the architecture design. Section 6.3 delves on argument templates, each arguing about the security of a reference architecture. Finally, we conclude the chapter.

6.2 Security argument patterns

The first part of this chapter presents some security argument patterns. Recalling that the patterns are defined to document the argumentation about a security engineering activity. First, we present an argument about the design of a secure software architecture in Section 6.2.1. Then, two argument patterns argue about the threat modeling activity. They argue through the STRIDE and PASTA methods in Sections 6.2.2, 6.2.3. Lastly, two argument patterns arguing about the formal verification of security requirements are presented in Sections 6.2.4, 6.2.5.

6.2.1 Secure software architecture argument pattern

This root pattern provides claim decompositions to demonstrate that the software architecture adequately satisfies its required security requirements.

Pattern name SecureArchitecture Pattern

Definition of the pattern We follow the method presented in section 5.4.1 to define the pattern, and we resume the results as follows.

Step 1 The top goal is inspired by the activity's goal which is to provide a secure software architecture.

Step 2 The argumentation strategy implies that a secure software architecture must verify all the security requirements [64].

Step 3 The following sub-claims are defined in such a way that their conjunction implies the satisfaction of the top goal. Consequently, the first sub-goal ensures that all

security requirements are enumerated, and the second sub-goal ensures that all security requirements are verified.

Step 4 The pattern aims to show the essential activities for designing a secure software architecture. We keep the sub-goals undeveloped in this pattern: we don't need more decompositions.

Step 5 The undeveloped sub goals do not require solutions.

Step 7 We add some contextual elements like links to the artifacts describing the system and its software architecture. Indeed, we add a justification to demonstrate the implicit reasoning in the argument.

Step 8 Using the static semantic rules that must be verified, we check if the pattern structure is correct. Once all the semantic rules, we move to the last step.

Step 9 Document the argument pattern as shown in the following section. Document the intent, motivation, applicability, structure and consequences.

Intent The pattern provides a claim decomposition to argue that the design of a software architecture satisfies its security requirements if some security engineering activities are well done. The goal is to provide a convincing argument that the software architecture under assurance complies with the necessary processes and activities to support claims that the software architecture is adequately secure. The aim is to help architects construct such an argument while reducing the effort required to do so.

Motivation Multiple activities are undertaken while designing a software architecture. Each activity generates some artifacts whose security should be ensured. Consequently, we will have many security assurance fragments. The secure software architecture argument pattern is a preliminary brick that indicates how the architecture-related patterns should be assembled.

Applicability The pattern should be applied when compliance is required in the secure architecture design processes for a system under assurance. The pattern assumes that the relevant design methodology provides sufficient context for its instantiation.

Structure The argumentation strategy is captured in the argument pattern shown in Listing 6.1. It's represented using SCML language. A graphical presentation is added in the appendix C.1

```
Pattern SecureArchitecture {
    metadata "See section for the description of the pattern"
    secActivity "architectureDesign"
    parameters {System.name, Artifact.link, EAL.value, SoftwareArchitecture.name}
    top goal G1 "The software architecture" + {SoftwareArchitecture.name} + "of the
        system" + {System.name} + "is secure in level" + {EAL.value}
    links {C1, C2, C3, S1}
    strategy S1 "Argue through verifying the security requirements" links{J1, G2, G3}
    justification J1 "A software architecture design is secure if it satisfies the
        security requirements"
    goal G2 toBeDeveloped "All the security requirements are identified"
    goal G3 toBeDeveloped "The software architecture model satisfies the security
        requirements"
    context C1 toBeInstantiated "The system description" + {Artifact.link}
    context C2 toBeInstantiated "The software architecture" + {Artifact.link}
    context C3 toBeInstantiated "The software architecture is defined in the context of
        EAL" + {EAL.value}
}
```

Listing 6.1: SecureArchitecture argument pattern

Consequences The pattern enables a separation of concerns in the argumentation about the fulfillment of software architecture security requirements. The decomposition of the security engineering activities allows us to focus on each activity in the software architecture design stage, one at a time. However, after applying the pattern, some undeveloped goals will remain. Specifically, the goals of the exhaustivity of the identified security requirements and their satisfaction will require further decomposition and support. One of the key challenges in constructing assurance arguments is determining the granularity by which each goal is decomposed so that sufficient evidence can be generated to fully support the goal [60]. Because there are many different ways to decompose goals, it is not immediately evident when to stop the decomposition or what sufficient granularity would be to ensure that the required evidence could be provided. In some cases, these goals can be decomposed by adopting other argument patterns recast in the context of security.

6.2.2 Argument pattern for threat modeling using STRIDE

Derivation of the pattern

We follow the method presented in section 5.4.1 to define the pattern.

Pattern name STRIDEThreatModeling pattern

Step 1 The top goal is inspired by the activity's goal: All the security threats are identified and mitigated.

Step 2] The argumentation strategy is depicted by the method. STRIDE classifies the threats into six classes. Thus, the argument pattern argues through the threat classes.

Step 3 We define a subgoal for each class threat among STRIDE classification.

Step 4 The subgoals require inturn more decomposition

Step 2 We define a second strategy that argues for each security threat

Step 3 Each subgoal argues for one threat belonging to one class threat

Step 4 No more decomposition is required

Step 5 Yes, the pattern does need evidence elements

Step 6 The artifacts produced by the activity will be used as evidence and will be referred to in the solution elements of the pattern

Step 7 Extra information about the method, the artifacts, the tool, and the security threats are added to the pattern using context or assumption, or justification assurance elements.

Step 8 Validate the structure and the content of the pattern by checking the semantic rules.

Step 9 Document the argument pattern as shown in the following section. Document the name, intent, motivation, applicability, structure, and consequences.

Intent The goal of this pattern is to provides a structured argument that demonstrates how the STRIDE method has been used to comprehensively identify potential threats to a system software architecture. The primary goal is to ensure that all relevant threat categories have been considered and addressed, thereby enhancing the system's security posture

Motivation Identifying and mitigating security threats early in the system development lifecycle is crucial to building robust and secure systems. The STRIDE methodology offers a systematic approach to threat modeling by categorizing threats into six distinct areas, ensuring a thorough examination of potential vulnerabilities. This pattern is motivated by the need to provide evidence that a comprehensive threat analysis has been conducted, covering all aspects of STRIDE. It helps stakeholders understand that the system has been scrutinized for security weaknesses and that appropriate mitigation strategies have been planned or implemented.

Applicability The Threat Modeling STRIDE Argument Pattern should be applied in the security architecture design phase of system development, particularly when there is a requirement to demonstrate a thorough threat identification process. It is relevant for systems that handle sensitive data, provide critical services, or operate in environments with high-security risks. The pattern assumes that there is sufficient context and detail about the system's architecture and functionality to apply the STRIDE methodology effectively. It applies to new system designs, system upgrades, and regular security reviews to ensure ongoing threat awareness and mitigation.

Structure The argumentation strategy is captured in the argument pattern shown in Listing 6.2. It's represented using SCML language. A graphical presentation is added in the appendix C.2

```
Pattern STRIDEThreatModeling {
    metadata "See the following section for the description of the pattern"
    secActivity threatModeling
    parameters {Threat.name, Threat.class, Activity.description, Activity.require, Method
        .name, Method.description, Artifact.link, SecurityPattern.mechanism,
        SecurityPattern.description }
    top goal G1 "All the threats are identified and mitigated" links{C1, C2, C3, S1}
    strategy S1 "Argue through each class of STRIDE classification" links{J1, A1, C4, G2.
        X}
    context C1 toBeInstantiated "Threat modeling description" + {Activity.description}
    context C2 toBeInstantiated "The system architecture model description" +{Activity.
        require}
    context C3 "Data flow diagram" + {Activity.require}
    goal G2.X toBeInstantiated "Threats of the class" + {Threat.class} + "are mitigated"
         links{S2.X}
    strategy S2.X toBeInstantiated "Argue through the threat class" + {Threat.class}
        links{G3.X.Y, G4.X.Y}
    goal G3.X.Y toBeInstantiated "Threat" + {Threat.name} + "is mitigated using" + {
        SecurityPattern.mechanism} links {Sn1.X.Y, C5}
    goal G4.X.Y "The mechanism" + {SecurityPattern.mechanism} + "does not violate the
        functional requirements of the system" links{C6, Sn2.X.Y}
    justification J1 "STRIDE is the most mature threat modeling activity"
```

```
    solution Sn1.X.Y toBeInstantiated "Agreement over inspection conducted by security
        experts" + {Artifact.link}
    solution Sn2.X.Y toBeInstantiated "Results of the verification of functional
        requirement" + {Artifact.link}
    assumption A1 "STRIDE classification is mapped to the threats identified "
    context C4 toBeInstantiated "Description of STRIDE method" + {Method.description}
    context C5 toBeInstantiated "Security mechanism is provided by the security pattern"
        + {SecurityPattern.description}
    context C6 toBeInstantiated "List of the functional requirements" + {Artifact.Link}
    IsSupportedBy ISB1 lowerBound 1 from S1 to G2.X  /*The upperBound is equal to the
        number of threat classes*/
    IsSupportedBy ISB2 lowerBound 1  /*The upperBound is equal to the number of threats
        per class*/
    InContextOf ICO1 lowerBound 0 upperBound 1 from S1 to J1 /*This relationship is
        optional */
}
```

Listing 6.2: STRIDEThreatModeling argument pattern

Consequences Applying the STRIDE method ensures that the threat identification process is exhaustive and structured. This pattern enables a clear separation of threat categories, making it easier to develop targeted security measures for each type of threat. Consequently, the argument supports the claim that the system's security requirements are met by addressing all identified threats. Additionally, the explicit consideration of each STRIDE category fosters a more detailed and nuanced understanding of the system's security needs, leading to better-prepared defenses against potential attacks.

6.2.3 Argument pattern for threat modeling using PASTA

Pattern name PASTAThreatModeling pattern

Derivation of the pattern We follow the method presented in section 5.4.1 to define the pattern

Step 1 The top goal is inspired by the activity's goal: All the threats are identified and mitigated

Step 2 The argumentation strategy is inspired by the method: Argue through each stage of the PASTA method.

Step 3 The definition of sub-claims. Each stage of PASTA method is seen as an activity included in the main activity PASTA threat modeling. Thus, we will define seven sub-goals.

Step 4 The sub goals do not require more decompositions.

Step 5 Yes, the pattern does need evidence elements .

Step 6 The artifacts produced by the activity will be used as evidence and will be referred to in the solution elements of the pattern.

Step 7 We make the argument more convincing by adding extra contextual information (the software architecture model) and assumptions about the correctness of the tool.

Step 8 Semantic rules regarding the structure and the content of the pattern that must be verified.

Step 9 Document the argument pattern as shown in the following section. Document the name, intent, motivation, applicability, structure and consequences.

Intent The pattern aims to provide a structured argument demonstrating how the PASTA method has been employed to identify and analyze potential threats to a system comprehensively. The primary goal is to show that all relevant threats have been systematically identified and assessed, ensuring a thorough understanding of the system's threat landscape.

Motivation Threat modeling is essential for proactive security management, and PASTA offers a risk-centric approach that aligns security activities with business objectives. By simulating potential attacks and analyzing their impact, PASTA helps prioritize threats based on their risk to the organization. This pattern is motivated by the need to create a convincing argument that the system's threat identification process is robust, methodical, and aligned with real-world attack scenarios. It supports stakeholders in understanding the threats and making informed decisions about risk management and mitigation strategies.

Applicability The Threat Modeling PASTA Argument Pattern should be applied when there is a need to perform a detailed and business-aligned threat analysis of a system. It is particularly relevant for complex systems with high-security requirements, where understanding the potential impact of threats on business objectives is crucial. The pattern assumes that there is adequate information about the system's architecture, data flows, and business context to effectively apply the PASTA methodology. It is applicable during the design phase, for existing systems undergoing security reviews, and in scenarios where a comprehensive threat analysis is necessary to inform security strategy and investment.

Structure The argumentation strategy is captured in the argument pattern shown in Listing 6.3. It's represented using SCML language. A graphical presentation is added in the appendix C.3

```
Pattern PASTAThreatModeling {
    metadata "See the following section for the description of the pattern"
    secActivity threatModeling
    parameters {Activity.description, Method.description, Tool.name , Artifact.link}
    top goal G1 "All the threats are identified and mitigated" links {C1, C3, C4, C5, S1}
    strategy S1 "Argue through each stage of P.A.S.T.A method" links {A1, A2, J1, C6, G2,
        G3, G4, G5, G6, G7, G8}
    context C1 toBeInstantiated "Threat modeling description" + {Activity.description}
    context C3 toBeInstantiated "System requirements" + {Artifact.link}
    justification J1 "The outputs of each stage provide inputs for the next stage"
    assumption A1 "There is sufficient time to follow this method "
    assumption A2 toBeInstantiated "The tool" + {Tool.name} + "used for the threat
        modeling is correct"
    goal G2 "The objectives are well defined" links {Sn1}
    goal G3 "Adequate technologies are chosen" links {Sn2, Sn3}
    goal G4 "All the assets are identified" links {Sn4, Sn5, Sn10}
    goal G5 "All the threats and threat actors are identified" links {Sn6, Sn11}
    goal G6 "All the vulnerabilities are identified and analysed" links {Sn7}
    goal G7 "All the attack scenarios are modeled" links {Sn8, Sn9}
    goal G8 "Risks are assessed and mitigated " links {Sn12, Sn13}
    solution Sn1 toBeInstantiated "Objectives validation by the stakeholders" + {
        Artifact.link}
    solution Sn2 toBeInstantiated "Expert opinion and endorsement" + {Artifact.link}
    solution Sn3 toBeInstantiated "High level architecture" + {Artifact.link}
    solution Sn4 toBeInstantiated "Data flow diagram" + {Artifact.link}
    solution Sn5 toBeInstantiated "Components diagram" + {Artifact.link}
    solution Sn6 toBeInstantiated "List of identified threats" + {Artifact.link}
    solution Sn7 toBeInstantiated "Vulnerability Assessment Reports" + {Artifact.link}
    solution Sn8 toBeInstantiated "Attack surface analysis" + {Artifact.link}
    solution Sn9 toBeInstantiated "Attack tree" + {Artifact.link}
    solution Sn10 toBeInstantiated "Mapping of the use cases with the assets and actors"
        + {Artifact.link}
    solution Sn11 toBeInstantiated "Security design flaws analysis" + {Artifact.link}
    solution Sn12 toBeInstantiated "Risk assessment report" + {Artifact.link}
    solution Sn13 toBeInstantiated "Risk mitigation plan" + {Artifact.link}
    context C4 toBeInstantiated "Security policies" + {Artifact.link}
    context C5 toBeInstantiated "System description" + {Artifact.link}
    context C6 toBeInstantiated "Description of PASTA method" + {Method.description}
}
```

Listing 6.3: PASTAThreatModeling argument pattern

Consequences Applying the PASTA method ensures a thorough and business-aligned threat identification process. This pattern facilitates a deep understanding of the attack surface and potential attack vectors, prioritizing threats based on their impact on business objectives. Consequently, the argument supports the claim that the system's security re-

quirements are met by addressing all identified threats in a prioritized and risk-informed manner. Additionally, the comprehensive nature of PASTA fosters detailed documentation and communication of threats, enhancing overall security preparedness and response capabilities.

6.2.4 Argument pattern for model checking

Pattern Name ModelChecking pattern

Derivation of the pattern We follow the method presented in section 5.4.1 to define the pattern,

Step 1 The top goal is the inspired by the activity's goal. The pattern claims that the software architecture satisfies the security requirements.

Step 2 The argumentation strategy is a decomposition strategy. We have to argue through satisfying each security requirement.

Step 3 We define a subgoal for each security requirement.

Step 4 Yes, sub-goals need more decomposition

Step 2 Define a second argumentation strategy that argues through the activities included in the formal verification.

Step 3 The definition of sub-claims depicts the two activities included in the formal verification activity. The first subgoal concerns the formal specification of the security requirements and the software architecture model. The second subgoal claims that the software architecture asset satisfies a security requirement.

Step 4 No, the subgoals do not require more decompositions

Step 5 Yes, the pattern needs evidence elements

Step 6 The artifacts produced by the activity will be used as evidence and will be referred to in the solution elements of the pattern

Step 7 Extra information about the method, the artifacts, the tool, and the security threats are added to the pattern using context or assumption, or justification assurance elements.

Step 8 We check the semantic rules that validate the structure and the content of the pattern.

Step 9 Document the argument pattern as shown in the following section. Document the name, intent, motivation, applicability, structure, and consequences.

Intent The goal of this pattern is to provide a convincing argument about using formal methods to both formalize and verify the security requirements at the architectural level. The pattern indicates what are the evidence elements necessary to support the claims about the satisfaction of security requirements.

Motivation The motivation behind verifying the fulfillment of security requirements stems from the potential risks associated with the software architecture models. Designing secure software architectures eases the detection and treatment of security failures which reduces the costs of fixing them after deploying the system. Consequently, the assurance of the security requirements fulfillment is of utmost importance.

Applicability The pattern should be applied when formal methods are required to verify the fulfillment of security requirements. The pattern assumes that the relevant standard or development methodology provides sufficient context for its instantiation.

Structure The argumentation strategy is captured in the argument pattern shown in Listing 6.4. It's represented using SCML language. A graphical presentation is added in the appendix C.4

```
Pattern ModelChecking {
    metadata "See the following section for the description of the pattern"
    secActivity fromalVerification
    parameters {Asset.name, Requirement.name, Requirement.description , Activity.produce,
        Activity.require , Method.name, Method.description , Tool.name, Requirement.
        formalReq , Artifact.link}
    top goal G0  "The software architecture model satisfies the security requirements"
        links {CO, C1, S0, A0}
    strategy S0 "Argue through verifying each security requirement" links {G1.X, C2, A1,
        C3}
    context C0 toBeInstantiated "The list of security requirements" + {Activity.require}
    context C1 toBeInstantiated "The software architecture model" + {Activity.require}
    context C2 toBeInstantiated "Description of the formal verification activity" + {
        Activity.description}
    context C3 toBeInstantiated "Description of the model checking method" + {Method.
        description}
    goal G1.X toBeInstantiated "The architecture asset" + { Asset.name} + "satisfies the
    security requirement" + {Requirement.name} links {C4.X, C5 .X, S1.X}
    strategy S1.X  "Argue over using model checking method" links {G2.X, G3.X}
```

```
    goal  G2.X toBeInstantiated  "Formal requirement" +{Requirement.formalReq} +"is the
        appropriate formal specification of the security requirement" + {Requirement.name
        } links {Sn1.X}
    goal  G3.X  toBeInstantiated "The formal model satisfies the formal requirement" + {
        Requirement.formalReq} links {Sn2.X}
    context C4.X toBeInstantiated "Description of the security requirement"+ {Requirement
        .description}
    context C5.X toBeInstantiated "Description of the asset"+ {Asset.description}
    solution Sn1.X  toBeInstantiated  "Validation report by formalization experts" + {
        Artifact.link}
    solution Sn2.X toBeInstantiated "Results of formal verification" + {Activity.produce}
    assumption A0  "The software architecture model is correct"
    assumption A1  toBeInstantiated "The tool" + {Tool.name} "is correct"
    IsSupportedBy S0G1.X lowerBound 1 from S0 to G1.X /*n is the number of security
        requirements */
}
```

Listing 6.4: ModelChecking argument pattern

Consequences The pattern demonstrates that the software architecture model will meet its security requirements. It also shows the relationship between the verification evidence and the security requirements. However, after instantiating the pattern, some goals will remain undeveloped. Specifically, the assumption related to the well-definedness of the software architecture model which requires further development and support.

6.2.5 Argument pattern for well-formedness of the model

This argument is a development of the assumption node *A0* presented in the previous pattern 6.2.4.

Pattern name ModelWellformedNess pattern

Derivation of the pattern We follow the method presented in section 5.4.1 to define the pattern,

Step 1 The top goal is inspired by the activity's goal: the software architecture model is well defined

Step 2 The argumentation strategy: Argue through the specification of the software architecture model

Step 3 The definition of sub-claims indicate the need for three sub-goals to claim: the correctness, consistency, and conformity of the software formal model.

Step 4 No, the subgoals do not require more decompositions.

Step 5 Yes, the pattern does need evidence elements

Step 6 The artifacts produced by the activity will be used as evidence and will be referred to in the solution elements of the pattern

Step 7 To make the argumentation pattern more convincing, we add more information from the security artifacts. This include descriptions of the formal model, formal language, and the tool

Step 8 The pattern semantics are validated

Step 9 Document the argument pattern as shown in the following section. Document the name, intent, motivation, applicability, structure, consequences.

Intent The goal of this pattern is to provide a convincing argument that a software architecture model is well-formed according to predefined criteria, standards and a formal language. The primary goal is to demonstrate that the model adheres to all necessary syntactic and semantic rules, ensuring its correctness and consistency.

Motivation Ensuring the well-formedness of a software architecture model is crucial for the reliability and maintainability of the system. A well-formed model is free from structural errors and inconsistencies, which facilitates accurate analysis, design, and implementation. This pattern is motivated by the need to establish confidence in the architecture model by proving its adherence to formal modeling rules and guidelines. It supports the development process by preventing errors that could propagate through subsequent stages, thereby reducing development costs and improving system quality

Applicability The Model Well-formedness Argument Pattern should be applied when there is a need to validate the structure and integrity of a software architecture model. This is particularly relevant in environments where formal modeling techniques are used, and adherence to specific modeling standards or frameworks is required. The pattern assumes that there are well-defined criteria for what constitutes a well-formed model, such as UML standards, domain-specific modeling languages, or other formal methods. It is applicable during the design phase, before detailed implementation, and in any situation where the correctness of the architecture model is critical for the project's success.

Structure The argumentation strategy is captured in the argument pattern shown in Listing 6.5. It's represented using SCML language. A graphical presentation is added in the appendix C.5

```
Pattern ModelWellFormedNess {
    metadata "See section for the description of the pattern"
    secActivity formalSpecification
    parameters {Tool.name , Artifact.link , Method.description , System.name}
    top goal G1 "The formal model of the system " + {System.name} + "software
        architecture is well defined " links {C1, C2, C3, S1}
    context C1 "Well defined means that it conforms to the specification and it is
        without errors "
    context C2 toBeInstantiated "Formal specification of the system architecture" + {
        Artifact.link}
    strategy S1 "Argue over correctness, consistency, and conformity of the model" links
        {G2, G3, G4}
    goal G2 "The model is correct" links {Sn1}
    goal G3 "The model is consistent" links {Sn2}
    goal G4 "The model is conform to the intended behavior" links {Sn3}
    solution Sn1 toBeInstantiated "The model is automatically validated using the tool" +
        {Tool.name}
    solution Sn2 toBeInstantiated "Results of the verification of the justification in
        justification J1" + {Artifact.link}
    solution Sn3 toBeInstantiated "Results of the formal verification of the model" + {
        Artifact.link}
    justification J1 "The meaning of consistent model according to the formal method"
    context C3 toBeInstantiated "Description of the formal method" + {Method.
        description}
}
```

Listing 6.5: ModelWellFormedNess argument pattern

Consequences Ensuring well-formedness of the model reduces the risk of incorrect conclusions during verification due to ambiguities or inconsistencies in the model structure.

However, this pattern may introduce some challenges, such as increased complexity and additional time spent in ensuring that the model conforms to formal rules and standards. The process of validating the well-formedness of the model might require specialized tools or expertise, increasing the overhead in the design and verification phases. This pattern enables the identification and correction of issues in the formal model before it is applied to the verification process.

6.2.6 Summary

In this section, we presented five argument patterns described using SCML. Additionally, other argument patterns have been defined in previous work related to this thesis, which are presented in Appendix C. Table 6.1 provides a summary of the features of the argument patterns discussed here and those in the appendix.

Table 6.1: Comparative summary of the argument patterns

Argument pattern	Notation	Reference	Activity	Method	Tool
SecureArchitecture	SCML	Section 6.2.1	Architecture design	/	/
STRIDEThreatModeling	SCML	Section 6.2.2	Threat modeling	STRIDE	/
PASTAThreatModeling	SCML	Section 6.2.3	Threat modeling	PASTA	/
ModelChecking	SCML	Section 6.2.4	Formal Verification	Model checking	/
ModelWellFromedness	SCML	Section 6.2.5	Formal specification	/	/
FormalVerification	GSN	Appendix C.6	Formal Verification	/	/
ThreatIdentification	GSN	Paper [131]	Threat Modeling	/	/
ThreatMitigation	GSN	Paper [131]	Threat Modeling	/	/
AlloyModelWellFromedness	GSN	Paper [130]	Formal specification	/	Alloy Analyzer
DNNSecureDevelopment	GSN	Paper[132]	DNN development	/	/
DNNVerification	GSN	Paper [132]	Formal Verification	/	/

Note that the patterns we defined in our previous works were described using GSN, which is a goal-based notation applied for general purposes. Therefore, we worked on a goal-based modeling language that involves security, process, and system concepts. Moreover, the level of abstraction considered while defining the pattern is crucial. Considering, for example, the threat modeling activity, we defined three argument patterns to argue about it. The first one is generic. The second and the third are adapted to the STRIDE and PASTA methods, respectively. Another example concerns the model well formedness pattern, where we define a particular pattern, namely the Alloy Analyzer tool.

6.3 Reusable argument templates

This second part of the chapter is dedicated to the security argument templates. Recalling that templates are defined to document the security argumentation of a reference architecture. Moreover, the reference architectures are defined per application domain and/or development technologies, for example, medical systems, cloud-based systems, or medical systems based on machine learning. First, we defined an argument template for the domain of Supervisory Control and Data Acquisition systems (SCADA). It is presented in Section 6.3.1. Then, we define a technology-related argument template for the machine learning-based systems (MLBS) which is presented in Section 6.3.2

6.3.1 Template for SCADA

Ensuring safety and reliability is crucial for SCADA systems, which are widely recognized for their strong performance in these areas. However, the primary concern today revolves around SCADA system security. Governments globally are increasingly focused on safeguarding SCADA systems that manage critical infrastructure, due to the rising risks of cyberattacks [84].

6.3.1.1 Introduction to SCADA systems

SCADA systems are meant to control processes through local controllers, acquiring field data and returning them to a SCADA master computer system. Figure 6.1 shows a typical SCADA systems reference architecture. It consists of a SCADA master, an operator workstation, and a number of field devices connected by a communication infrastructure. Field devices can be Programmable Logic Controllers (PLC), Remote Terminal units (RTU), sensors and actuators. The SCADA master provides the operator with a Human-Machine Interface (HMI) through a work station to issue commands to PLCs and gather

field data from them. PLCs are digital computers programmed to continuously monitor sensors and control actuators (e.g., valves, pumps, etc.). RTUs are used for converting sensor data into digital data.

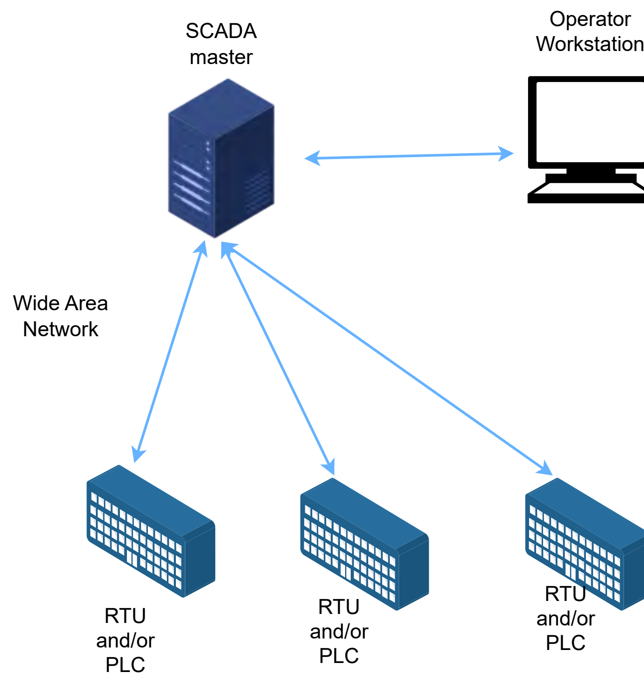


Figure 6.1: A typical architecture of SCADA systems adopted from [111]

We follow the methodology presented in 5.6 to define the argument template. To do so, we start by defining a secure reference architecture of SCADA systems.

6.3.1.2 Reference architecture for SCADA system

The reference architecture of SCADA systems is adopted from work in [84]. It is composed of two reference components and a communication channel. The description of the reference architecture is given in Listing 6.6.

```
ReferenceArchitecture RA_SCADA "This is a reference architecture for SCADA systems"
  domain "SCADA"
  use { ReferenceAsset ControlCenter  role "component"
    ReferenceAsset CommunicationChannel "The communication channel that connects control
        centers with remote terminal units (RTUs)" role "connector" connects RTU,
        ControlCenter
    ReferenceAsset RTU  role "component" }
```

Listing 6.6: Reference architecture of SCADA system

6.3.1.3 Securing a SCADA system

The main characteristic considered in selecting the SCADA system application was that these systems are targeted to many attacks (e.g., March 2016 event [103]). Securing SCADA systems was studied in detail in the work of Fernandez [33]. They propose a method to analyze, build, and evaluate secure SCADA systems using security patterns. The result introduced the SECURE SCADA pattern, which is used as a guideline for building secure reference architecture for SCADA systems.

Threat modeling The STRIDE threat modeling activity applied to SCADA reference architecture, has defined the following threats. Each of the threats is mitigated using a security pattern. We sum up the results of this activity in the following Table 6.2.

Consequently, the secure reference architecture is described in Listing 6.7.

```
ReferenceArchitecture RA_SCADA "This is a secure reference architecture for SCADA systems
  "
  domain "SCADA"
  use { ReferenceAsset ControlCenter role "component" protectedBy ROLE BASED ACCESS
      CONTROL, AUTHENTICATOR, SECURITY LOGGER, AUDITOR, AUTHORIZATION, Firewall
    ReferenceAsset CommunicationChannel "The communication channel that connects control
        centers with remote terminal units (RTUs)" role "connector" connects RTU,
        ControlCenter protectedBy ROLE BASED ACCESS CONTROL, AUTHENTICATOR, SECURITY
       LOGGER, AUDITOR, AUTHORIZATION, Firewall
    ReferenceAsset RTU role "component" protectedBy Securechannel , AUTHENTICATOR}
```

Listing 6.7: Secure reference architecture of SCADA system

Security Requirements Verification After enumerating the threats menacing the software architecture and proposing security patterns to mitigate them. We must verify whether the proposed security patterns ensure a secure software architecture model. Thus, we propose to use formal methods to verify the security of the reference architecture. Noting that the goal of threat modeling activity is twofold. On the one hand, it allows the enumeration of the threats and proposes security mechanisms to mitigate them. On the other hand, it elicits security requirements, where each security requirement prescribes security mechanisms that mitigate the threats. We sum up in Table 6.3 the correspondence between the threats, the security requirements, and the mechanisms that mitigate them.

6.3.1.4 Security argument template for SCADA system

Template Name SCADA security argument template

Table 6.2: SCADA System Security: Threats and Security Patterns

Asset	Threat	Threat class	Security pattern
Control	**th.1** Physical attacks	Tampering	ROLE-BASED ACCESS CONTROL, AUTHENTICATOR, SECURITY LOGGER, and AUDITOR
	th.2 Malicious settings	Tampering	AUTHORIZATION, AUTHENTICATOR, SECURITY LOGGER, and AUDITOR
	th.3 Wrong commands sent to field units	Repudiation	
	th.4 Malicious alteration of the parameters	Tampering	
	th.5 Denial of service attack	Denial of service	Firewall
Channel	**th.6** Sniffing commands	Information Disclosure	ROLE-BASED ACCESS CONTROL, AUTHENTICATOR, SECURITY LOGGER, and AUDITOR
	th.7 Spoofing	Spoofing	AUTHORIZATION, AUTHENTICATOR, SECURITY LOGGER, and AUDITOR
	th.8 Denial of service attack	Denial of Service	Firewall
RTU	**th.9** Physical attacks	Tampering	ROLE-BASED ACCESS CONTROL, AUTHENTICATOR, SECURITY LOGGER, and AUDITOR
	th.10 Malicious alteration of the run-time parameters	Tampering	Secure channel
	th.11 Incorrect commands sent to the central controller	Repudiation	AUTHENTICATOR

Derivation of the template We follow the method presented in section 5.5.1 to define the template, The starting point of the template definition is the definition of a top goal.

Table 6.3: SCADA System Security: Threats, Requirements, and Security Mechanisms

Asset	Threat	Security requirement	Security mechanisms
Control	**th.1**	**SR.1** The control must be protected from physical attacks	**SM.1** ROLE-BASED ACCESS CONTROL, AUTHENTICATOR, SECURITY LOGGER, and AUDITOR
	th.2	**SR.2** The control must be protected from malicious settings	**SM.2** AUTHORIZATION, AUTHENTICATOR, SECURITY LOGGER, and AUDITOR
	th.3	**SR.3** The control must be protected from wrong commands sent to field units	
	th.4	**SR.4** The control must be protected from malicious alteration of the parameters	
	th.5	**SR.5** The control must be protected from Denial of service attack	**SM.3** Firewall
Channel	**th.6**	**SR.6** The channel must be protected from sniffing commands	**SM.4** ROLE-BASED ACCESS CONTROL, AUTHENTICATOR, SECURITY LOGGER, and AUDITOR
	th.7	**SR.7** The channel must be protected from Spoofing attacks	**SM.5** AUTHORIZATION, AUTHENTICATOR, SECURITY LOGGER, and AUDITOR
	th.8	**SR.8** The channel must be protected from Denial of service attacks	**SM.6** Firewall
RTU	**th.9**	**SR.9** The RTU must be protected from physical attacks	**SM.7** ROLE-BASED ACCESS CONTROL, AUTHENTICATOR, SECURITY LOGGER, and AUDITOR
	th.10	**SR.10** The RTU must be protected from malicious alteration of the run-time parameters	**SM.8** Secure channel
	th.11	**SR.11** The RTU must be protected from incorrect commands sent to the central controller	**SM.9** AUTHENTICATOR

It claims that the reference architecture model of SCADA systems is secure. After, we select from the security argument patterns catalogue the patterns required to build the template.

Pattern selection Step 1 The definition of the template's top goal.

Step 2 Noting that the template top goal fits with the top goal of the pattern `SecureArchitecture` 6.1.

Step 3 Consequently, we select the `SecureArchitecture` to define the template.

Step 4 Noting that the pattern `SecureArchitecture` contains two undeveloped goals, namely G2 and G3, we go back to **Step 4** to select argument patterns that can fit with the selected pattern.

Step 3 We select the patterns `STRIDEThreatModeling` and `ModelChecking` that fit the undeveloped goals G2 and G3 of the pattern `SecureArchitecture`, respectively. The choices of these patterns need to be justified later in the template. They are influenced by the security engineering activities conducted to build the secure reference architecture.

Step 4 The pattern `ModelChecking` requires another pattern to argue about an assumption about the welldefinedness of the software architecture model. We add the pattern `ModelWellFormedNess` to the list of the selected patterns.

Step 10 We check if the selected patterns compose a correct template. We go back to **Step 2** to move to the editing template steps

Template editing Step 5 All the goals in the selected patterns are already developed and do not require more decomposition.

Step 8 The solution elements required to support the selected patterns will be solutions to support the template.

Step 9 We remove any repetitive assurance elements and add the necessary information about the context, the assumptions and justifications of the argumentation.

Template documentation This template is developed to argue about the security of the software architecture of SCADA systems. It requires conducting a threat modeling activity and a formal verification of the security requirements using model checking as a formal method.

Template Structure The argumentation strategy is captured in the argument template shown in Listing 6.8. It's represented using SCML language.

```
Template SCADA
metadata "This is an argument template for the security of SCADA systems software
    architectures"
refArchitecture "RA_SCADA"
patterns SecureArchitecture, STRIDEThreatModeling, ModelChecking
parameters {System.name, Artifact.link, EAL.value, SoftwareArchitecture.name}
/*  SecureArchitecture pattern */
top goal P1G1 "The software architecture" + {SoftwareArchitecture.name} + "of the
    system" + {System.name}  + "is secure in level" + {EAL.value}  links {P1C1, P1C2,
     P1C3, P1S1}
strategy P1S1  "Argue through verifying the security requirements" links{P1J1, P1G2,
    P3G0, P4G1}
justification P1J1 "A software architecture design is secure if it satisfies the
    security requirements"
goal P1G2  "All the security requirements are identified" links{S1}
strategy S1 "Argue through following a threat modeling to elicit the security
    requirements" links{J1, P2G1}
justification J1  "The security requirements prescribe security mechanisms that
    mitigate the identified threats. Thus, the identification of all the threats
    implies the identification of all the security requirements"
/* STRIDEThreatModeling pattern  */
goal P2G1 "All the threats are identified and mitigated" links{P2C1,  P2C2,  P2C3,
    P2S1}
strategy P2S1 "Argue through each class of STRIDE classification" links{P2J1, P2C4,
    P2A1, P2G2.1, P2G2.2, P2G2.3, P2G2.4, P2G2.5}
justification P2J1 "STRIDE is the most mature threat modeling activity"
/* Spoofing threats*/
goal P2G2.1   "Threats of the class Spoofing are mitigated" links{ P2S2.1}
strategy P2S2.1  "Argue through the threat class Spoofing "  links{P2G3.1.1, P2G4
    .1.1}
/* Threat th.7 */
goal P2G3.1.1 "Threat th.7 is mitigated using SM.5 "  links {P2Sn1.1.1, P2C5}
solution P2Sn1.1.1 toBeInstantiated "Agreement over inspection conducted by security
    experts" + {Artifact.link}
goal P2G4.1.2 "The mechanism SM.5 does not violate the functional requirements of the
     system" links{P2C6, P2Sn2.1.2}
solution P2Sn2.1.2 toBeInstantiated "Results of the verification of functional
    requirements" + {Artifact.link}
/* Tampering threats*/
goal P2G2.2  "Threats of the class Tampering
are mitigated" links{P2S2.X}
strategy P2S2.2  "Argue through the threat class   Tampering" links{P2G3.2.1, P2G4
    .2.1, P2G3.2.2, P2G4.2.2, P2G3.2.3, P2G4.2.3, P2G3.2.4, P2G4.2.4, P2G3.2.5, P2G4
    .2.5}
/*  Threat th.1 */
goal P2G3.2.1 "Threat th is mitigated using SM.1 "  links {P2Sn1.2.1, P2C5}
solution P2Sn1.2.1 toBeInstantiated "Agreement over inspection conducted by security
    experts" + {Artifact.link}
goal P2G4.2.1 "The mechanism SM.1 does not violate the functional requirements of the
     system" links{P2C6, P2Sn2.2.1}
solution P2Sn2.2.1 toBeInstantiated "Results of the verification of functional
```

```
    requirements" + {Artifact.link}
/*  Threat th.2 */
goal P2G3.2.2 "Threat th.2 is mitigated using SM.2" links {P2Sn1.2.2, P2C5}
solution P2Sn1.2.2 toBeInstantiated "Agreement over inspection conducted by security
    experts" + {Artifact.link}
goal P2G4.2.2 "The mechanism SM.2 does not violate the functional requirements of the
     system" links{P2C6, P2Sn2.2.2}
solution P2Sn2.2.2 toBeInstantiated "Results of the verification of functional
    requirement" + {Artifact.link}
/*  Threat  th.4 */
goal P2G3.2.3 "Threat th.4 is mitigated using SM.2 " links {P2Sn1.2.3, P2C5}
solution P2Sn1.2.3 toBeInstantiated "Agreement over inspection conducted by security
    experts" + {Artifact.link}
goal P2G4.2.3 "The mechanism SM.2 does not violate the functional requirements of the
     system" links{P2C6, P2Sn2.2.3}
solution P2Sn2.2.3 toBeInstantiated "Results of the verification of functional
    requirements" + {Artifact.link}
/*  Threat th.9 */
goal P2G3.2.4 "Threat th.9 is mitigated using SM.7" links {P2Sn1.2.4, P2C5}
solution P2Sn1.2.4 toBeInstantiated "Agreement over inspection conducted by security
    experts" + {Artifact.link}
goal P2G4.2.4 "The mechanism SM.7 does not violate the functional requirements of the
     system" links{P2C6, P2Sn2.2.4}
solution P2Sn2.2.4 toBeInstantiated "Results of the verification of functional
    requirements" + {Artifact.link}
/*  Threat th.10 */
goal P2G3.2.5 "Threat th.10 is mitigated using SM.8" links {P2Sn1.2.5, P2C5}
solution P2Sn1.2.5 toBeInstantiated "Agreement over inspection conducted by security
    experts" + {Artifact.link}
goal P2G4.2.5 "The mechanism SM.8 does not violate the functional requirements of the
     system" links{P2C6, P2Sn2.2.5}
solution P2Sn2.2.5 toBeInstantiated "Results of the verification of functional
    requirements" + {Artifact.link}
/* Denial Of Service */
 goal P2G2.3  "Threats of the class Denial Of Service are mitigated" links{P2S2.X}
strategy P2S2.3 "Argue through the threat class Denial Of Service  " links{P2G3.3.1,
     P2G4.3.1, P2G3.3.2, P2G4.3.2}
/* Threat th.5 */
goal P2G3.3. "Threat th.5 is mitigated using SM.3 " links {P2Sn1.3.1, P2C5}
solution P2Sn1.3.1 toBeInstantiated "Agreement over inspection conducted by security
    experts" + {Artifact.link}
goal P2G4.3.1 "The mechanism SM.3 does not violate the functional requirements of the
     system" links{P2C6, P2Sn2.3.1}
solution P2Sn2.3.1 toBeInstantiated "Results of the verification of functional
    requirements" + {Artifact.link}
    /* Threat th.8 */
goal P2G3.3.2 "Threat th.8 is mitigated using SM.6" links {P2Sn1.3.2, P2C5}
solution P2Sn1.3.2 toBeInstantiated "Agreement over inspection conducted by security
    experts" + {Artifact.link}
goal P2G4.3.2 "The mechanism SM.6 does not violate the functional requirements of the
     system" links{P2C6, P2Sn2.3.2}
solution P2Sn2.3.2 toBeInstantiated "Results of the verification of functional
    requirements" + {Artifact.link}
/*  Information Disclosure */
```

```
 goal P2G2.4 "Threats of the class Information Disclosure are mitigated" links{P2S2
     .4}
strategy P2S2.4 "Argue through the threat class Information Disclosure " links{P2G3
    .4.1, P2G4.4.1}
/* Threat th.6 */
goal P2G3.4.1 "Threat th.6 is mitigated using SM.4 " links {P2Sn1.4.1, P2C5}
solution P2Sn1.4.1 toBeInstantiated "Agreement over inspection conducted by security
    experts" + {Artifact.link}
goal P2G4.4.1 "The mechanism SM.4 does not violate the functional requirements of the
     system" links{P2C6, P2Sn2.4.1}
solution P2Sn2.4.1 toBeInstantiated "Results of the verification of functional
    requirements" + {Artifact.link}
/* Repudiation */
 goal P2G2.5 "Threats of the class Repudiation
 are mitigated" links{ P2S2.X}
strategy P2S2. "Argue through the threat class Repudiation" links{P2G3.5.1, P2G4
    .5.1, P2G3.5.2, P2G4.5.2}
/* Threat th.3 */
goal P2G3.5.1 "Threat th.3 is mitigated using SM.2" links {P2Sn1.5.1, P2C5}
solution P2Sn1.5.1 toBeInstantiated "Agreement over inspection conducted by security
    experts" + {Artifact.link}
goal P2G4.5.1 "The mechanism SM.2 does not violate the functional requirements of the
     system" links{P2C6, P2Sn2.5.1}
solution P2Sn2.5.1 toBeInstantiated "Results of the verification of functional
    requirements" + {Artifact.link}
    /* Threat th.11 */
goal P2G3.5.2 "Threat th.11 is mitigated using SM.9 " links {P2Sn1.5.2, P2C5}
solution P2Sn1.5.2 toBeInstantiated "Agreement over inspection conducted by security
    experts" + {Artifact.link}
goal P2G4.5.2 "The mechanism SM.9 does not violate the functional requirements of the
     system" links{P2C6, P2Sn2.5.2}
solution P2Sn2.5.2 toBeInstantiated "Results of the verification of functional
    requirements" + {Artifact.link}
/*  ModelChecking  pattern*/
goal P3G0 "The  software architecture model satisfies the security requirements"
links {P3CO, P3S0, P3C1, P3A0}
strategy P3S0  "Argue through verifying each security requirement" links {P3G1.X,
    P3C2, P3A1, P3C3}
/* Security Requirement SR.1 */
goal P3G1.X toBeInstantiated "The architecture asset control satisfies the security
    requirement" + {Requirement.name}  links {P3C4.X, P3C5.X, P3S1.X}
strategy P3S1.X "Argue over using model checking method" links {P3G2.1, P3G3.1}
goal P3G2.X toBeInstantiated "Formal requirement" + {Requirement.formalReq} + "is the
     appropriate formal specification of the security requirement" +{Requirement.name
    }  links {P3Sn1.1}
goal P3G3.X toBeInstantiated "The formal model satisfies the formal requirement" + {
    Requirement.formalReq} links {P3Sn2.1}
context P3C4.X toBeInstantiated "Description of the security requirement" + {
    Requirement.description}
context P3C5.X toBeInstantiated "Description of the asset" + {Asset.description}
solution P3Sn1.X toBeInstantiated "Validation report by formalization experts" + {
    Artifact.link}
solution P3Sn2.X toBeInstantiated "Results of formal verification" + {Activity.
    produce}
```

```
/* Welldefinedness model pattern */
goal P4G1 "The formal model of the system " + {System.name} + "software architecture
    is well defined " links {P4C1, P4C2, P4C3, P4S1}
context P4C1 "Well defined means that it conforms to the specification and it is
    without errors "
context P4C2 toBeInstantiated "Formal specification of the system architecture" + {
    Artifact.link}
strategy P4S1 "Argue over correctness, consistency, and conformity of the model"
    links {P4G2, P4G3, P4G4}
goal P4G2 "The model is correct" links {P4Sn1}
goal P4G3 "The model is consistent" links {P4Sn2}
goal P4G4 "The model is conform to the intended behavior" links {P4Sn3}
solution P4Sn1 toBeInstantiated "The model is automatically validated using the tool"
    + {Tool.name}
solution P4Sn2 toBeInstantiated "Results of the verification of the justification in
    justification J1" + {Artifact.link}
solution P4Sn3 toBeInstantiated "Results of the formal verification of the model" + {
    Artifact.link}
justification P4J1 "The meaning of consistent model according to the formal method"
/*Contextual assurance elements */
context P1C1 toBeInstantiated "The system description" + {Artifact.link}
context P1C2 toBeInstantiated "The software architecture model" + {Artifact.link}
context P1C3 toBeInstantiated "The software architecture is defined in the context
    of EAL" + {EAL.value}
context P2C1 toBeInstantiated "Threat modeling description" + {Activity.description
    }
context P2C3 "Data flow diagram" + {Activity.require}
assumption P2A1 "STRIDE classification is mapped to the threats identified "
context P2C4 toBeInstantiated "Description of STRIDE method" + {Method.description}
context P2C5 toBeInstantiated "Security mechanism is provided by the security pattern
    " + {SecurityPattern.description}
context P2C6 toBeInstantiated "List of the functional requirements" + {Artifact.Link}
context P3C0 toBeInstantiated "The list of security requirements" + {Activity.require
    }
context P3C2 toBeInstantiated "Description of the formal verification activity" + {
    Activity.description}
context P3C3 toBeInstantiated "Description of the model checking method" + {Method.
    description}
/* This assumption will be replaced with the modelwellformedness pattern
assumption P3A0 "The software architecture model is correct" */

assumption P3A1 toBeInstantiated "The tool" + {Tool.name} "is correct"
/* Abstract relationships */
IsSupportedBy ISB1 lowerBound 1 from P3S0 to P3G1.X
```

Listing 6.8: Security argument template for SCADA system

Template seen as a system of patterns The template is also seen as a system of patterns. We use a pattern diagram in which rounded rectangles represent patterns and arrows indicate the contribution of a pattern to another. Figure 6.2 presents the structure of the SCADA template.

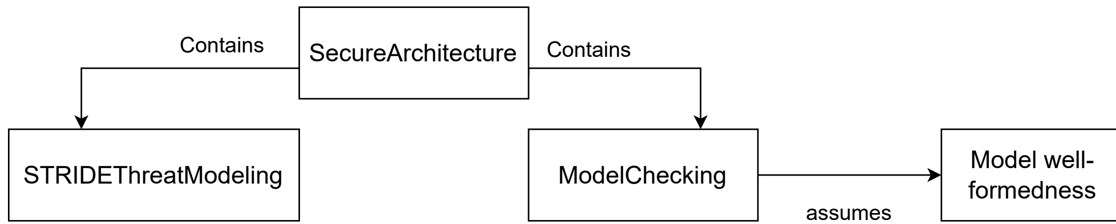


Figure 6.2: Patterns diagram for the SCADA Template

6.3.2 Template for MLBS

Despite the rapid growth of MLBS and their deployment in critical domains, interest in their security and trustworthiness has only arisen in the last few years, and it is not yet well-studied. The ML components that do not learn enough, are not competent to be trusted to make the intended decision [51]. For instance, ML techniques were used to build a pedestrian detector for a self-driving car. This ML component was trained to only recognize pedestrians on a crosswalk. The self-driving car integrating this ML component killed a pedestrian that was not near the crosswalk [51]. The developers of MLBS are asked to provide assurance cases for the system's security. Several works have recently been interested in assuring the ML components [18], [54].

6.3.2.1 Introduction to MLBS systems

MLBS are such systems that integrate some components based on ML techniques. ML techniques replace repetitive tasks done by human experts. Developing the ML components passes through two phases: the training phase and the inference phase. The specific task-related data supplied by domain experts is used to create a training data set. The ML component will learn how to carry out the same task using the data provided. The output of the training is the ML model handled by the ML component. During the inference phase, the ML components are used in large systems composed of different kinds of components, some of which may also be ML-based. The ML component should receive data of the same type as the training data. Figure 6.3 represents the different components that align with various steps to set up an MLBS (the numbers in parentheses correspond to the numbers in Figure 6.3):

- Raw data in the world is used in both the training and the inference ML component setting up phases (1).

- The Data collection process aims to collect the raw data and to prepare it for the training phase (2).
- Data Set is the component where the data that will be used to train the ML component is stored (3).
- The Model training process launches the ML algorithm on the training data to derive the ML model (4).
- Input Sender is the component used to query the ML component (5).
- ML component is the component that handles the ML model (6).
- Output Receiver is the component that receives the results returned by the ML component during the inference phase (7).

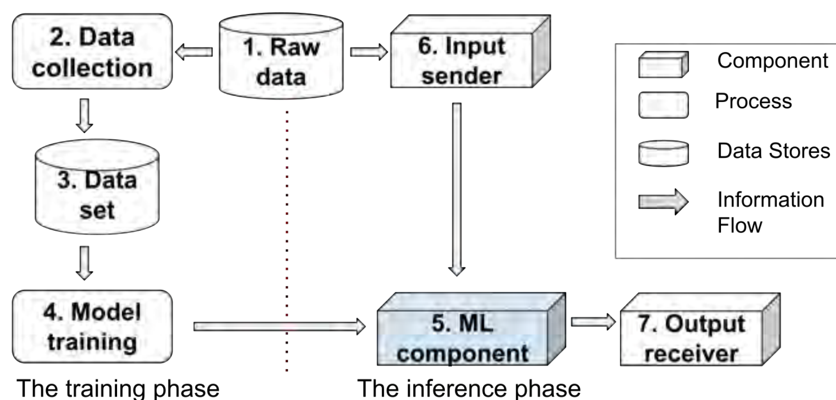


Figure 6.3: A typical architecture of MLBS systems

We follow the methodology presented 5.6 to define the argument template. To do so, we start by defining a secure reference architecture of MLBS systems.

6.3.2.2 Reference architecture for MLBS system

The reference architecture of MLBS systems is adopted from our work in [129]. It is composed of three types reference components and a communication channel. The description of the reference architecture is given in Listing 6.9.

```
ReferenceArchitecture RA_MLBS "This is a reference architecture for MLBS systems"
  technology "MLBS"
 use {  ReferenceAsset RawData type"Component"
   ReferenceAsset DataCollection type"Component"
   ReferenceAsset DataSet type"Component"
```

```
ReferenceAsset ModelTraining type"Component"
ReferenceAsset MLcomponent  type"Component"
ReferenceAsset InputSender  type"Component"
ReferenceAsset OutputReceiver  type"Component"
ReferenceAsset DataFlow1 type"Connector" connects RawData, DataCollection
ReferenceAsset DataFlow2 type"Connector" connects DataCollection , DataSet
ReferenceAsset DataFlow3 type"Connector" connects DataSet , ModelTraining
ReferenceAsset DataFlow4 type"Connector" connects ModelTraining ,  MLcomponent
ReferenceAsset DataFlow5 type"Connector" connects  MLcomponent, OutputReceiver
ReferenceAsset DataFlow6 type"Connector" connects RawData,  InputSender
ReferenceAsset DataFlow7 type"Connector" connects  InputSender , MLcomponent
 }
```

Listing 6.9: Reference architecture of MLBS system

6.3.2.3 Securing a MLBS system

ML-based systems (MLBS) development profoundly depends on software architecture. MLBS has been proven vulnerable to both traditional threats that plague systems without ML components, as well as new threats that are targeting ML components [71].

Threat modeling The STRIDE threat modeling activity applied to MLBS reference architecture, has defined some of the threats, they are called non ML specific threats. However, there are some ML specific threats that are not mapped to STRIDE classification. Details about the ML specific threats are found in the Appendix C.3.

The security of the MLBS is a challenging topic. To the best of our knowledge, there is no proposition of a secure reference architecture. Moreover, there are no security patterns that mitigate the ML-specific threats, unlike those for SCADA reference architecture. We sum up the results of this activity in Table 6.4.

Consequently, the secure reference architecture is described in Listing 6.10.

```
ReferenceArchitecture SRA_MLBS "This is a secure reference architecture for MLBS systems"
  technology "MLBS"
  use {  ReferenceAsset RawData type"Component"
    protectedBy SM1, SM2, SM3, SM4, SM5, SM6, SM7, SM8
    ReferenceAsset DataCollection type"Component" protectedBy SM1, SM2, SM3, SM4, SM5,
        SM6, SM7, SM8
    ReferenceAsset DataSet type"Component"
    protectedBy SM1, SM2, SM3, SM4, SM5, SM6, SM7, SM8
    ReferenceAsset ModelTraining type"Component" protectedBy SM1, SM2, SM3, SM4, SM5, SM6
        , SM7, SM8
    ReferenceAsset MLcomponent  type"Component"  protectedBy SM1, SM2, SM3, SM4, SM5, SM6
        , SM7, SM8
    ReferenceAsset InputSender  type"Component" protectedBy SM1, SM2, SM3, SM4, SM5, SM6,
        SM7, SM8
    ReferenceAsset OutputReceiver  type"Component" protectedBy SM1, SM2, SM3, SM4, SM5,
        SM6, SM7, SM8
```

Table 6.4: MLBS Security: Threats and Security Mechanisms

AssetRole	Threat	ML-specific	Security mechanism
Component	**th.1** Spoofing	Non ML-specific	**SM.1**
	th.2 Tampering	Non ML-specific	**SM.2**
	th.3 Repudiation	Non ML-specific	**SM.3**
	th.4 Information Disclosure	Non ML-specific	**SM.4**
	th.5 Denial of service	Non ML-specific	**SM.5**
	th.6 Model extraction	ML-specific	**SM.6**
	th.7 Data extraction	ML-specific	**SM.7**
	th.8 Adversarial examples	ML-specific	**SM.8**
Connector	**th.9** Information Disclosure	Non ML-specific	**SM.9**
	th.10 Tampering	Non ML-specific	**SM.10**
	th.11 Denial of service	Non ML-specific	**SM.11**

```
    ReferenceAsset DataFlow1 type"Connector" connects RawData, DataCollection
        protectedBy SM9, SM10, SM11
    ReferenceAsset DataFlow2 type"Connector" connects DataCollection, DataSet protectedBy
         SM9, SM10, SM11
    ReferenceAsset DataFlow3 type"Connector" connects DataSet, ModelTraining protectedBy
         SM9, SM10, SM11
    ReferenceAsset DataFlow4 type"Connector" connects ModelTraining, MLcomponent
        protectedBy SM9, SM10, SM11
    ReferenceAsset DataFlow5 type"Connector" connects MLcomponent, OutputReceiver
        protectedBy SM9, SM10, SM11
    ReferenceAsset DataFlow6 type"Connector" connects RawData, InputSender protectedBy
        SM9, SM10, SM11
    ReferenceAsset DataFlow7 type"Connector" connects InputSender, MLcomponent
        protectedBy SM9, SM10, SM11
  }
```

Listing 6.10: Secure reference architecture of MLBS system

After enumerating the threats menacing the software architecture, we propose abstract security mechanisms to mitigate them. For example, the security mechanism **{SM.1}** mitigates the threat **th.1**. Finally, we must verify whether the proposed security mechanisms ensure a secure software architecture model. Thus, we have multiple choices to verify the satisfaction of ML-specific security requirements: (1)*Formal verification*, (2)*Testing*, and (3)*Monitoring*. More details about the activities are found in Appendix A

6.3.2.4 Security argument template for MLBS system

Template Name MLBS security argument template

Derivation of the template We follow the method presented in section 5.5.1 to define the template, The starting point of the template definition is the definition of a top goal. It claims that the reference architecture model of MLBS systems is secure. After, we select from the security argument patterns catalogue the patterns required to build the template. We adapt the domain-independent patterns to the machine learning technology considering the particularities of the security of those systems.

Pattern selection **Step 1** We define the template top goal.

- **Step 2** Noting that the template top goal fits with the top goal of the pattern `SecureArchitecture` 6.1.
- **Step 3** Consequently, we select the `SecureArchitecture` to define the template.
- **Step 4** Noting that the pattern `SecureArchitecture` contains two undeveloped goals, namely G2 and G3, we go back to **Step 4** to select argument patterns that can fit with the selected pattern.
- **Step 3** We notice that the patterns `STRIDEThreatModeling` and `ModelChecking` are not adapted to machine learning-based systems. There are new emerging ML-specific threats that do not appear within the STRIDE classification. Moreover, the model checking as a formal method is not recommended for the security requirements allocated to ML components. Thus, we replace the undeveloped goal G2 claiming about the identification of all the security requirements with an assumption A4. However, the goal G3 remains undeveloped.

Template editing **Step 5** The goal G3 needs to be developed by adding assurance elements directly to the template.

- **Step 6** This goal requires decomposition; we suggest an argumentation focusing on the ML-specific security requirements.
- **Step 7** The two sub-goals **G5, G6** argue about the satisfaction of the non-ML-specific and ML-specific security requirements, respectively.

Pattern selection **Step 2 , Step3** The goal G5 argues about the satisfaction of non-ML-specific security requirements. It fits with the top goal of the pattern *FormalVerification*. This pattern is selected and added to the template. However,

there is no argument pattern that argues about the satisfaction of ML-specific security requirements.

Step 4 No more argument is required

Step 10 We check if the selected patterns compose a correct template. We go back to **Step 2** to move to the editing template steps.

Template editing **Step 5, Step 6** The goal G6 needs to be developed. It requires a decomposition based on the choice of the verification strategy.

Step 7 The sub-goals refer to the verification choices presented in the template. The sub-goal G12 claims the formal verification. Then, the sub-goal G13 claims to test the security requirements. Finally, the sub-goal G14 claims about monitoring the ML component. None of the argument patterns fits with the sub-goals, thus we continue developing the template

Step 8 The solution elements required to support the template includes the artifacts produced by the formal verification and testing activities, a list of the performance metrics, validation of the experts and standards that define the monitoring policies.

Step 9 We remove any repetitive assurance elements and add the necessary information about the context, the assumptions, and justifications of the argumentation. For example, we need to assume that the ML-specific security requirement can be formally verified to follow that strategy. Moreover, we assume that the test platform covers the system.

Template documentation This template is developed to argue about the security of the software architecture of MLBS systems. It requires conducting a threat modeling activity and a formal verification of the security requirements using model checking as a formal method.

Template Structure The argumentation strategy is captured in the argument template shown in Listing 6.11. It's represented using SCML language.

```
Template MLBS
metadata "This is an argument template for the security of MLBS systems software
    architectures"
refArchitecture "RA_MLBS"
patterns SecureArchitecture, FormalVerification
parameters {Artifact.link , EAL.value, System.name, SoftwareArchitecture.name ,
    Activity.assumption, Activity.justification, Activity.require, Requirement.
    formalReq, Activity.produce, Method.description, Requirement.name, Requirement.
    description, Asset.name, Tool.name, Asset.description}
```

```
/*   SecureArchitecture pattern */
top goal P1G1 "The software architecture" + {SoftwareArchitecture.name} + "of the
    system" + {System.name}  + "is secure in level" + {EAL.value}  links {P1C1, P1C2,
     P1C3, P1S1}
strategy P1S1  "Argue through verifying the security requirements" links{P1J1, P1G2,
    P1A4}
justification P1J1 "A software architecture design is secure if it satisfies the
    security requirements"
assumption P1A1  "All the security requirements are identified"
assumption P1A4  "The system model is well defined"
goal P1G2 "The software architecture model satisfies the security requirements" links
    {P1A1, S2}
strategy S2 "Argue through ML security requirements" links{J2, P2G5, P2G6}
justification J2 "The focus in this template is the satisfaction of ML security
    requirements "
/* Formal verification pattern */
goal P2G5 "The software architecture satisfies the non-ML-specific security
    requirements" links {P2C4, P2S4}
strategy P2S4 "Argue through each non ML security requirement" links{P2G1.X, P2J6,
    P2A1, P2A2, P2A3}
assumption P2A1 toBeInstantiated "The activity assumption" + {Activity.assumption}
assumption P2A2  "The formal model of the system is well-defined"
assumption P2A3 toBeInstantiated "The tool" + {Tool.name} "is correct"
justification P2J6 toBeInstantiated "The activity justification" + {Activity.
    justification}
goal P2G1.X toBeInstantiated  "The architecture asset" +{Asset.name} +"satisfies  the
     security requirement" +{Requirement.name} links{P2S1.X, P2C1.X, P2C5.X}
strategy P2S1.X toBeInstantiated  "Argue over using formal verification" + {Method.
    name} links{P2C2.X, P2G2.X, P2G3.X}
goal  P2G2.X toBeInstantiated "Formal requirement" + {Requirement.formalReq} + "is
    the appropriate formal specification of the security requirement" + {Requirement.
    name} links{P2Sn1.X}
solution Sn1.X toBeInstantiated "Validation report by formalization experts" + {
    Artifact.link}
goal  P2G3.X toBeInstantiated "The formal model satisfies the formal requirement" + {
    Requirement.formalReq} links{P2Sn2.X}
solution Sn2.X toBeInstantiated "Results of formal verification"+{Activity.produce}
/* The verification of ML-specific security requirements */
goal G6 "The software architecture satisfies ML-specific security requirements" links
    {C7, S9}
strategy S9 "Argue through the choice of the verification method" links{G12, G11, G14
    , }
/*  Formal verification of ML-specific security requirements  using
    FormalVerification pattern */
goal G12 "The  ML-specific security requirements are formally verified" links{S3}
strategy S3 "Argue through each ML security requirement" links {P3G1.X, P3A1, P3J6,
    P3A2}
assumption toBeInstantiated  P3A2 "The tool" + {Tool.name} + "is correct"
assumption toBeInstantiated  P3A1 {Activity.assumption}
justification toBeInstantiated P3J6 {Activity.justification}
goal P3G1.X toBeInstantiated  "The architecture asset" +{Asset.name} +"satisfies  the
     security requirement" +{Requirement.name} links{P3S1.X, P3C1.X, P3C5.X}
strategy P3S1.X toBeInstantiated  "Argue over using formal verification" + {Method.
    description} links{P3C2.X, P3G2.X, P3G3.X}
```

```
goal  P3G2.X toBeInstantiated "Formal requirement" + {Requirement.formalReq} + "is
    the appropriate formal specification of the security requirement" + {Requirement.
    name} links{P3Sn1.X}
solution P3Sn1.X toBeInstantiated "Validation report by formalization experts" + {
    Artifact.link}
goal  P3G3.X toBeInstantiated "The formal model satisfies the formal requirement" + {
    Requirement.formalReq} links{P3Sn2.X}
solution P3Sn2.X toBeInstantiated "Results of formal verification"+{Activity.produce}
/*  Testing the ML-specific security requirements*/
goal G11 "The ML-specific security requirements are tested" links{S8, A5, A7}
assumption A5 "Test platform is sufficiently representative of the system"
assumption A7 toBeInstantiated "The tool" + {Tool.name} "is correct"
strategy S8 "Argue through each ML security requirement" links{G12.Y, G14.Y, C6}
goal G12.Y toBeInstantiated  "Test cases related to the security requirement" + {
    Requirement.name} + "are passed" links{Sn3.Y}
solution Sn3.Y  toBeInstantiated "Test results" +  {Activity.produce}
goal G14.Y toBeInstantiated "Test cases are sufficient to demonstrate the
    satisfaction of " + {Requirement.name} + " within the defined operating context
    of the system " links{Sn4.Y }
solution Sn4_Y toBeInstantiated "Verification log" + {Artifact.link}
/* Monitoring the ML-specific security requirements */
goal G14  "Monitoring the ML component satisfies the ML-specified security
    requirements" links{S5}
strategy S5 "Argue through the monitoring policy" links{J7, G8, G9, G10, A6}
Assumption A6 toBeInstantiated "The tool" + {Tool.name} "is correct"
justification J7  "Monitoring policies help ensure that the model behaves safely
    under all circumstances, including during abnormal inputs or adversarial attacks"
goal G8 "The monitoring policy is well-defined" links{Sn2, Sn3}
solution Sn2 toBeInstantiated "List of the standards used to define the policies" + {
    Activity.require}
solution Sn3 toBeInstantiated "Qualification of the experts who defined the
    monitoring policy" + {Artifact.link}
goal G9 "The monitoring does not impact the system performance" links{Sn1}
solution Sn1 toBeInstantiated "Results of performance metrics" + {Artifact.link}
goal G10 "The monitoring detects data extraction queries" links{Sn4}
solution Sn4 toBeInstantiated "Test results of the monitoring policy " + {Activity.
    produce}
/* Contextual assurance elements  */
context P1C1 toBeInstantiated  "The system description" + {Artifact.link}
context P1C2 toBeInstantiated "The software architecture model" + {Artifact.link}
context P1C3  toBeInstantiated "The software architecture is defined in the context
    of EAL" + {EAL.value}
context P2C1.X toBeInstantiated "Description of the security requirement"+{
    Requirement.description}}
context P2C2.X toBeInstantiated   "Description of the method" + {Method.description}
context P2C4 toBeInstantiated "The list of non ML-specific security requirements" + {
    Artifact.link}
context C6 toBeInstantiated "Testing activity description" + {Artifact.link}
context C7 toBeInstantiated "The list of ML-specific security requirements" + {
    Artifact.link}
context P2C5.X.Y toBeInstantiated "Description of the asset" + {Asset.description}
/* abstract relationShips */
/* apply for each non ML specific security requirement */
IsSupportedBy ISB1 lowerBound 1 from P2S4 to P2G1.X
```

```
/* apply for each ML specific security requirement formally verifiable */
IsSupportedBy ISB2 lowerBound 0 from P3S4 to P3G1.X
/* It is optional to add the justifications and assumptions of the activity */
InContextOf ICO1 lowerBound 0 upperBound 1 from P2S4 to P2A1
InContextOf ICO2 lowerBound 0 upperBound 1 from P2S4 to P2J6
/* apply for each asset */
InContextOf ICO3 lowerBound 1 from P2G1.X to P2C5.X.Y
/* choose a verification activity and argumentation strategy */
IsSupportedBy ISB3 lowerBound 0 upperBound 1 from S9 to G11
IsSupportedBy ISB4 lowerBound 0 upperBound 1 from S9 to G12
IsSupportedBy ISB5 lowerBound 0 upperBound 1 from S9 to G14
```

Listing 6.11: Security argument template for MLBS system

Template seen as a system of patterns Similarly to the SCADA template, we provide a diagram for the patterns composing the MLBS template as seen in Figure 6.4.

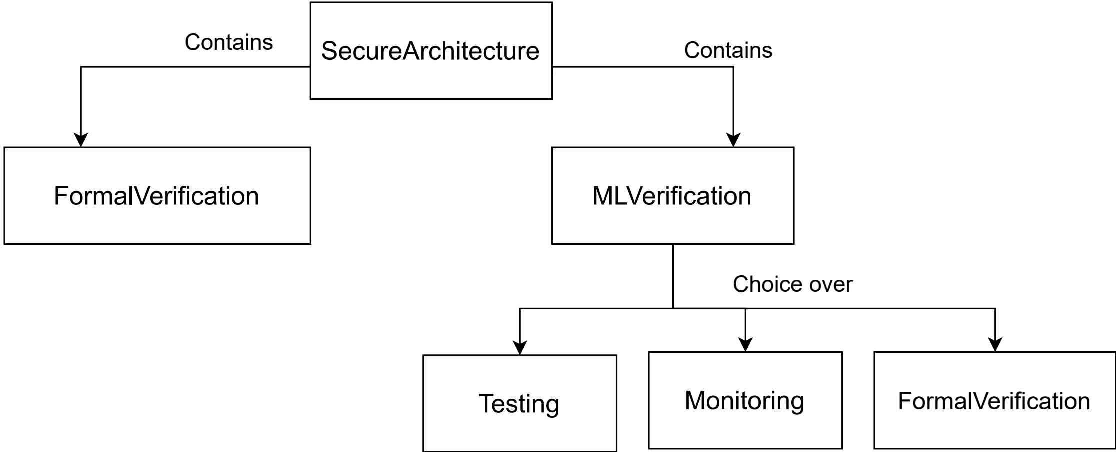


Figure 6.4: Patterns diagram for the MLBS template

6.3.3 Summary

In this section, we presented two argument templates described using SCML. The first template argues about SCADA systems, and the second argues about MLBS systems. Table 6.5 provides a summary of the features of the two templates.

Note that the SCADA template is designed specifically for the SCADA domain with no particular technology focus. It includes a fully developed reference architecture using concrete security patterns to address threats, as detailed in the referenced literature. This

Table 6.5: Comparative summary of the argument templates

Template	**Domain**	**Technology**	**Secure reference architecture**	**Abstraction**
SCADA	SCADA	/	• Defined in [34] • Defines concrete security patterns to mitigate the threats	Fully developed and easily applicable
MLBS	/	Machine learning	• Proposed in this manuscript • Proposes abstract security patterns	Parameterized with the security patterns that need to be instantiated for each particular system

template is readily applicable. In contrast, the MLBS template is geared towards machine learning systems, proposing an abstract security framework. This template, introduced in the current manuscript, requires parameterization with specific security patterns to suit each system. It emphasizes flexibility and adaptability over predefined structure. Lastly, the table indicates that having a well-defined secure reference architecture, like in the SCADA reference architecture, simplifies the development of argument templates.

6.4 Conclusion

In this chapter, we have proposed a diverse catalogue of security argument patterns, providing a comprehensive guide to various security engineering activities that help design a secure software architecture model. This catalogue serves as a valuable resource for architects, offering insights into the motivation, consequences of each pattern.

In the first part of this chapter, we began with a root argument pattern that is used as a basic for the assurance models. Then, we introduce two patterns for the same activity, namely Threat modeling. The first pattern argues on STRIDE threat modeling method. However, the second pattern argues on PASTA threat modeling. Moreover, the last pattern argues about the model checking method as a formal verification activity.

This activity requires another argument pattern about the welldefinedness of the formal model. Other patterns from our previous works are presented on the Appendix C.

The second part of the chapter is dedicated to argument template. The first template is domain dependant and argues about the SCADA systems. The second template is technology development dependant and argues about the MLBS systems.

In conclusion, this chapter provides a robust knowledge base of argumentation patterns and templates, each offering unique advantages to address specific design assurance challenges. As we proceed to the next chapters, we will build on this foundation, learning how to apply these patterns and templates effectively in real-world scenarios.

Chapter 7

Evaluation of the contributions

Contents

7.1 Introduction **131**

7.2 Case study **132**

7.2.1 Expressing the architecture of the ACAS Xu 132

7.2.2 Securing the software architecture of ACAS Xu 133

7.3 Definition of the reference data **135**

7.3.1 SecureArchitecture reference data 136

7.3.2 Formal Verification reference data 137

7.3.3 ML formal verification reference data 140

7.3.4 ML monitoring reference data 142

7.4 Security case for ACAS Xu **145**

7.5 Discussions **148**

7.5.1 Key features of the approach 148

7.5.2 Applications of the proposed approach 150

7.5.3 Generalization of the proposed approach 151

7.6 Conclusion **151**

7.1 Introduction

This chapter assesses the contributions presented in the thesis, mainly the creation of a security case from an argument template. Concretely, we follow the method presented

in 5.5 to create security cases using argument templates. We recall briefly the steps:

Step(1) The selection of an argument template. For the case study, we select the argument template for MLBS defined in Listing 6.11.

Step(2) We select the automatic application of the template to showcase the use of the reference data.

Step(3) The definition of the reference data after collecting the secure software architecture model, security engineering activities, and artifacts. for details about using the reference data.

Step(5) Validation of the semantic rules is required to have a correct security case with convincing arguments.

Step(6) Finally, we document the generated security case.

The rest of the chapter is organized as follows. Section 7.2 presents the case study: an autonomous drone under assurance. Then, we present the reference data created to apply the template in Section 7.3. After, Section 7.4 presents the security case of the ACAS Xu software architecture. Later, we present discussions on the features, the potential application, and the generalization of the proposed approach in Section 7.5. Finally, we conclude by summing up the contribution of the chapter in Section 7.6.

7.2 Case study

We use ACAS Xu system to exemplify the proposed approach. The aim of this case study is to provide a methodology to improve existing approaches to define security cases for secure systems using MDE. We evaluate the proposed approach in the construction of a methodology that is adapted for assuring secure systems focusing on the architecture design.

7.2.1 Expressing the architecture of the ACAS Xu

We start by defining the component types and the interfaces and connectors using the software architecture depicted in Figure 3.4 and the secure reference architecture of MLBS defined in Listing 7.1.

```
SoftwareArchitecture SA_ACAS "This is a description for the  architecture of ACAS Xu"
    generatedFrom SRA_MLBS
    relateTo ACASXu
    use {SoftwareAsset Sensors implement InputSender type "component" containother GPS,
        Velocity
        SoftwareAsset GPS implement InputSender  type "component"
        SoftwareAsset Velocity  implement InputSender type "component"
        SoftwareAsset Actuator implement OutputReceiver type "component"
        SoftwareAsset Planning  implement MLcomponent type "component"
        SoftwareAsset Processor implement OutputReceiver type "component"
        SoftwareAsset connect1 implement DataFlow7   type "connector" connects Sensors,
            Processor
        SoftwareAsset connect2 implement DataFlow7  type "connector" connects Processor,
            Planning
        SoftwareAsset connect3 implement DataFlow5  type  "connector" connects Planning,
            Actuator
    }
    System ACASXu domain "aeronautical"  technology "MLBS"
```

Listing 7.1: Secure Architecture of ACAS Xu system

7.2.2 Securing the software architecture of ACAS Xu

Recalling that the secure reference architecture prescribes abstract security mechanisms that protect the reference assets. To define concrete security mechanisms that protect the assets of ACAS Xu architecture, we extract a selection of security requirements from [123, 67, 8] that prescribe a secure operation of the system.

SR1: The drone system should be able to ensure that the sensor's information is genuine and has not been intentionally or unintentionally altered.

SR2: All the components of the system should be guaranteed to perform their required functions under defined spatial and temporal circumstances such that the system sustains its availability without disruption during its operational period.

SR3: The drone system should employ mechanisms to mitigate unauthorized disclosure of the information during its transmission.

SR4: The system should be able to guarantee the authenticity of its software components. For example, spoofing GPS signals could make the drone think it's in a different location, leading to navigation errors.

SR5: The system has to prevent attackers injecting falsified inputs to the planning which makes this last plan wrong paths.

SR6: The planning (ML component) has to be protected from attackers that aim to infer the functionality or discovering the structure or parameters of the model by observing its predictions.

Security mechanisms In our work, we consider countermeasures in the form of security architectural patterns. In the following, we present a set of security patterns from [34, 128] as appropriate solutions to fulfill the security requirements.

1. Authenticator: When a subject identifies itself to the system, the Authenticator pattern allows verification that the subject intending to access the system is who or what it claims to be.

2. Encryption: Encryption protects message confidentiality by making a message unreadable to those who do not have access to the key.

3. Adversarial training: The adversarial training pattern aims to improve the resilience of machine learning models by exposing them to adversarial examples during the training process. By training the model with these examples, it becomes better at recognizing and resisting such attacks in real-world scenarios.

4. Policy-Based Access control: The Policy-Based Access control pattern describes how to decide whether a subject is authorized to access an object according to policies defined in a central policy repository.

5. Intrusion Detection System: it alerts the system in real time when an intruder is trying to attack it.

6. Authorization: it describes who is authorized to access specific resources in a system, in an environment in which we have resources whose access needs to be controlled. The model indicates, for each active subject, which resources the subject can access and what it can do with them.

In Table 7.1, we summarize the identified security threats against drones, the corresponding violated security requirements, and possible mitigation techniques.

Requirement	**Threat**	**Assets**	**SecurityPatterns**	**Mechanism**
SR.1	**th.10** Tampering	Sensors + connect1	Authenticator + Encryption	**SM9**

SR.2	**th.11** Denial Of Service	Sensors + Planning + Processor + Actuator	Policy-Based Access control + Authorization	**SM11**
SR.3	**th.9** Information disclosure	connect1 + connect2 + connect3	Policy-Based Access control + Authorization	**SM9**
SR.4	**th.19** Spoofing	Sensors + Planning + Processor + Actuator	Authorization + Authenticator	**SM19**
SR.5	**th.18** Adversarial examples	Planning	Adversarial training + Authorization	**SM18**
SR.6	**th.17** Model extraction	Planning	Encryption + Policy-Based Access control	**SM17**

Table 7.1: ACAS Xu Security: Threats, Requirements, Security Patterns and Mechanisms

7.3 Definition of the reference data

The generation of reference data consists of finding values for the abstract parameters in the template. These values are recovered from secure architecture engineering artifacts. The reference data is a tree-based structure of data elements. The reference data is built to be similar to the argument pattern or template structure where we can have a parallel in-depth scan of the reference data elements and the assurance model elements. The algorithm 1 shows how reference data is built for a pattern argumentation. Similarly, we define reference data for the application of an argument template.

The MLBS template has the following abstract parameters shown in Listing 7.2.

```
Template MLBS
patterns SecureArchitecture , FormalVerification
parameters {SoftwareArchitecture.name, System.name, EAL.value , Activity.assumption ,
    Activity.justification , Asset.name, Requirement.name, Activity.require , Method.
   description , Requirement.formalReq , Artifact.link , Activity.produce ,
   Requirement.description , Asset.description}
```

Listing 7.2: Abstract parameters from the MLBS template

Algorithm 1 Definition of the reference data structure from a pattern

Require: *pattern* , an empty *RefData*
  Initialize the root of *RefData* within the *topgoal* of the *pattern*
  current is the variable used to scan the pattern
  for All the *AssuranceElement* linked to *current* **do**
    if The element has *parameters* **then**
      Add a new *dataelement* to the *RefData* using the *name* and *parameters* of the *AssuranceElement*
    end if
    if The type of the element is *Goal* or *Strategy* **then**
      Create a new branch in *RefData* starting from the assurance element {It's a recursive call}
    end if
  end for

A manual application of the template requires replacing each of the parameters with a concrete value from the reference architecture. A fully automated application of the template requires an automatic definition of the reference data (the structure and the content). At the current state of our work, we propose a semi-automated application of the template, i.e., an automated definition of the structure of the reference data, and an automatic mapping of the values of the parameters.

7.3.1 SecureArchitecture reference data

The parameters of the pattern *SecureArchitecture* are obtained from the system description. First, the top goal **P1G1** requires the name of the system, its software architecture and its level of assurance. Then, the context elements **P1C1** and **P1C2** are instantiated with links to artifacts that describe the ACAS Xu and its software architecture, respectively. Then, the context **P1C3** is instantiated with the assurance level for which compliance is required. The system requires higher assurance levels involving formal design and verification methods. Thus, EAL6 is appropriate for ML systems handling high-value or sensitive data, such as autonomous airborne. The reference data of the assurance elements from *SecureArchtiecture* pattern is given in Listing 7.3.

```
reference data ACASXuRefData for Template MLBS
  data element P1G1 { SoftwareArchitecture.name="SA_ACAS"   System.name= "ACASXu"  EAL.
       value="EAL6" {
         data element P1C1 { Artifact.link = "Path/To/Artifact/SystemDescription.scml" }
         data element P1C2 { Artifact.link ="Path/To/Artifact/SoftwareArchitecture.scml" }
         data element P1C3 { EAL.value="EAL6" }
 ......
 }
```

Listing 7.3: Excerpt of the reference data for ACAS Xu system (SecureArchtiecture)

7.3.2 Formal Verification reference data

The *FormalVerification* pattern is used to verify the non-ML-specific security requirements. It is a generic pattern related to an activity and method independent. Among the security requirements listed in Table 6.4, four are non related to Machine learning, namely *SR1, SR2, SR3, SR4.* Formal methods, particularly theorem Proving, are widely used to verify the security requirements at the architectural level. Listing 7.4 describes the theorem proving method using SCML language.

```
Activity FormalVerification "  "
    goal "verify the security requirements"
    justification "Formal verification is required by the evaluation assurance level EAL6
        "
    assumption: "The system can be executed symbolically with symbolic inputs, allowing
        the exploration of multiple paths simultaneously."
    phase ArchitecturalDesign
    type Verification
    require {SoftwareArchitecture, SecurityRequirements}
    produce {proofObligations, SRFormalSpec}
    carriedBy "Architect"
    use "TheoremProver"
Method TheoremProver "It is a deductive formal method, used for the formal verification
    of security requirements"
    executedBy Rodin
Artifact SoftwareArchitecture
    link "Path/To/Artifact/SoftwareArchitecture.scml"
    type Documentation
    describeAsset SoftwareArchitecture
    language SCML
Artifact SecurityRequirements
    link "Path/To/Artifact/SecurityRequirements.scml"
    type Documentation
    language SCML
Artifact FormalSpecification
    link "Path/To/Artifact/FormalSpec"
    type Documentation
    language Event_B
Artifact: ProofObligation
    link "Path/To/Artifact/FormalResults.scml"
    type VerificationResults
    language Event_B
```

Listing 7.4: Theorem Proving Activity

Among the artifacts used by the activity described in the aforementioned Listing 7.4 and written using SCML, we have the artifact listing the security requirements. Listing 7.5

describes the security requirements according to Table 7.1.

```
Requirement SR1 "The drone system should be able to ensure that the sensors information
    are genuine and have not been intentionally or unintentionally altered."
    mitigate th.10
    fulfilledBy Authenticator + Encryption
    allocatedToAsset Sensors, connect1
    formalreq "formalSR3"
Requirement SR2 "All the components of the system should be guaranteed to perform their
    required functions under defined spatial and temporal circumstances"
    mitigate th.11
    fulfilledBy Policy−Based Access control, Authorization
    allocatedToAsset  Sensors, Planning, Processor, Actuator
    formalreq "formalSR2"
Requirement SR3 " The drone system should employ mechanisms to mitigate unauthorized
    disclosure of the information during its transmission."
    mitigate th.9
    fulfilledBy Policy−Based Access control, Authorization
    allocatedToAsset connect1, connect2, connect3
    formalreq  "formalSR3"
Requirement SR4 " The system should be able to guarantee the authenticity of its software
     components "
    mitigate th.19
    fulfilledBy Authorization, Authenticator
    allocatedToAsset Sensors, Planning, Processor, Actuator
    formalreq "formalSR4"
    }
```

Listing 7.5: List of the non-ML-specific security requirements

The artifacts required to apply the pattern *FormalVerification* used in the template are collected. Thus, the content of the related reference data is depicted in the following Listing 7.6

```
reference data ACASXuRefData for Template MLBS
    data element P1G1 {
     ........
        data element P2G5 {
            data element P2C4 { Artifact.link ="Path/To/Artifact/NonMLSecurityRequirement
                .scml" }
            data element P2S4 {
                data element P2A1 {
                Activity.assumption = "The system can be executed symbolically with
                    symbolic inputs, allowing
                the exploration of multiple paths simultaneously  " }
                data element P2A3  { Tool.name = Rodin }
                data element P2J6{ Activity.justification = "Formal verification is
                    required by the evaluation assurance level EAL6 " }  }
                data element P2G1.1 { Asset.name = "Sensors, connect1" Requirement.name =
                     "SR.1"
                    data element P2C1.1 {Requirement.description = "The drone system
                        should be able to ensure that the sensors informations are
                        genuine and have not been intentionally or unintentionally
                        altered"}
```

```
    data element P2C5.1.1 {Asset.description = "Sensors collect different
         information like: GPS, velocity"}
    data element P2C5.1.2 {Asset.description = "connect1 is the connector
         linking the sensors to the Processor component"}
    data element P2S1.1 { Method.name = "TheoremProver"
        data element P2C2.1 { Method.description = "It is a deductive
            formal method, used for the formal verification of security
            requirements "}
        data element  P2G2.1 { Requirement.formalReq = "formalSR1"
            Requirement.name = "SR.1"
            data element P2Sn1.1 {
            Artifact.link = "Path/To/Artifact/ExpertsFormalSpec.txt" }   }
        data element P2G3.1 { Requirement.formalReq= "formalSR1"
            data element P2Sn2.1 {
            Activity.produce = "Path/To/Artifact/ProofObligation"
            }  }    }    }
data element P2G1.2 { Asset.name = "Sensors", "Planning", "Processor", "
    Actuator"
    Requirement.name = "SR.2"
    data element P2C1.2 {Requirement.description = "All the components of
         the system should be guaranteed to perform their required
        functions under defined spatial and temporal circumstances"}
    data element P2C5.2.1 {Asset.description = "Planning is responsible
        for route and task planning "}
    data element P2C5.2.2 {Asset.description = "Processor is the central
        computing unit that processes data received from the sensors "}
    data element P2C5.2.3 {Asset.description = "Actuator translates the
        planning decisions into physical actions"}

    data element P2S1.2 { Method.name = "TheoremProver"
        data element P2C2.2 { Method.description = "It is a deductive
            formal method, used for the formal verification of security
            requirements "}
        data element  P2G2.2 { Requirement.formalReq = "formalSR2"
          Requirement.name = "SR.2"
            data element P2Sn1.2 {
            Artifact.link = "Path/To/Artifact/ExpertsFormalSpec.txt"
            }}
        data element P2G3.2 { Requirement.formalReq= "formalSR2"
            data element P2Sn2.2 {
            Activity.produce =  " Path/To/Artifact/ProofObligation"
            }  }    }  }
data element P2G1.3 { Asset.name = "connect1", "connect2", "connect3"
    Requirement.name = "SR.3"
    data element P2C1.3 {Requirement.description = "The drone system
        should comply mechanisms to mitigate unauthorized disclosure of
        the information during its transmission"}
    data element P2C5.3.1 {Asset.description = "connect2 is the connector
         linking the Processor and the Planning components"}
    data element P2C5.3.2 {Asset.description = "connect3 is the connector
         linking the Planning and the Planning components"}
    data element P2S1.3 { Method.name = "TheoremProver"
        data element P2C2.3 { Method.description = "It is a deductive
            formal method, used for the formal verification of security
```

```
                    requirements  "}
                data element   P2G2.3 { Requirement.formalReq = "formalSR3"
                  Requirement.name = "SR.3"
                    data element P2Sn1.3 {
                    Artifact.link = "Path/To/Artifact/ExpertsFormalSpec.txt"
                    }}
                data element P2G3.3 { Requirement.formalReq= "formalSR3"
                    data element P2Sn2.3 {
                    Activity.produce = "Path/To/Artifact/ProofObligation"
                    }   }    }   }
        data element P2G1.4 { Asset.name = "Sensors, Planning, Processor, and
            Actuator"
            Requirement.name = "SR.4"
            data element P2C1.4  {Requirement.description = "The system should be
                able to guarantee the authenticity of the software components"}
            data element P2C5.4.1 {Asset.description = "Planning is responsible
                for route and task planning "}
            data element P2C5.4.2 {Asset.description = "Processor is the central
                computing unit that processes data received from the sensors "}
            data element P2C5.4.3 {Asset.description = "Actuator translates the
                planning decisions into physical actions"}
            data element P2C5.4.4 {Asset.description = "Sensors collect different
                information like: GPS, velocity"}

            data element P2S1.4 { Method.name = "TheoremProver"
                data element P2C2.4 { Method.description = "It is a deductive
                    formal method, used for the formal verification of security
                    requirements  "}
                data element   P2G2.4 { Requirement.formalReq = "formalSR4"
                  Requirement.name = "SR4"
                    data element P2Sn1.4 {
                    Artifact.link = "Path/To/Artifact/ExpertsFormalSpec.txt"
                    }   }
                data element P2G3.4 { Requirement.formalReq= "formalSR4"
                    data element P2Sn2.4 {
                    Activity.produce = "Path/To/Artifact/ProofObligation"
                    }  }  }   }
......
  }
```

Listing 7.6: Excerpt of the reference data for ACAS Xu system (Formal Verification)

7.3.3 ML formal verification reference data

We have discussed previously in Section 6.3.2 that verifying ML-specific security requirements differs from the verification of the rest of the security requirements. Three methods are possible to verify the fulfillment of the ML-specific security requirements: *formal verification*, *Testing*, and *Monitoring*.

Listing 7.7 describes the security requirements according to Table 7.1.

```
Requirement SR5 "The drone system have to prevent attackers injencting falsified inputs
    to the planning which makes this last plan wrong paths"
    mitigate th.18
    fulfilledBy Adversatial training, Authorization
    allocatedToAsset Planning
    formalreq "formalSR5"
Requirement SR6 "The planning (ML component) has to be protected from attackers that aim
    to infer the functionality or discovering the structure or parameters of the model by
     observing its predictions."
    mitigate th.17
    fulfilledBy Encryption, Policy-Based Access control, Access control, Anomalous
        Continuous Inquiries
    allocatedToAsset   Planning
    formalreq "formalSR6"
    }
```

Listing 7.7: List of the ML-specific security requirements

On one hand, adversarial examples attempt to modify the inputs in specific intervals to lead to output range violation. Thus, many formal verification techniques are used to verify the safety and security properties of DNN. Among these techniques, we have interval analysis. Interval analysis studies the arithmetic operations on intervals rather than concrete values. The DNN computations only include additions and multiplications (linear transformations) and an activation function (simple nonlinear operations). Consequently, by setting input features as intervals, we could follow the exact arithmetic performed in the DNN to compute the output intervals. Based on the output interval, we can verify if the input perturbations will finally lead to violations or not [120]. The description of interval analysis using SCML is given in Listing 7.8

```
Activity FormalVerification " "
    goal "verify the security requirements"
    justification "Formal verification is required by the evaluation assurance level EAL6
        "
    assumption: "The inputs of the ML components must have clear, finite ranges. Interval
         analysis requires that each input feature can be expressed as an interval."
    phase ArchitecturalDesign
    type Verification
    require {SoftwareArchitecture, MLSecurityRequirements}
    produce {MLSRFormalSpec}
    carriedBy "Architect"
    use "IntervalAnalysis"
Method  IntervalAnalysis "It is used to analyze and verify the behavior of systems,
    particularly those involving numerical computations and continuous variables"
    executedBy ReluVal
Artifact SoftwareArchitecture
    link "Path/To/Artifact/SoftwareArchitecture.scml"
    type Documentation
    describeAsset SoftwareArchitecture
    language SCML
Artifact MLSecurityRequirements
```

```
    link "Path/To/Artifact/MLSecurityRequirements.scml"
    type Documentation
    language SCML
Artifact MLSRFormalSpec
    link "Path/To/Artifact/MLSRFormalSpec"
    type Documentation
```

Listing 7.8: Interval Analysis Activity

Consequently, the reference data related to the verification of the ML-specific security requirement, namely adversarial examples, is depicted in Listing 7.9.

```
reference data ACASXuRefData for Template MLBS
    data element P1G1 {
    .....
        data element P3C4 { Artifact.link ="Path/To/Artifact/MLSecurityRequirement.scml"
            }
        data element P3S4 {
            data element P3A1 {
                Activity.assumption = " The ML inputs must have clear, finite ranges
                    where each input feature can be expressed as an interval" }
            data element  P3A2 {Tool.name = "ReluVal"}
            data element P3J6{ Activity.justification = "Formal verification is required
                by the evaluation assurance level EAL6 " }  }
        data element P3G1.1 { Asset.name = "Planning" Requirement.name = "SR.5"
            data element P3C1.1 {Requirement.description = "The drone system have to
                prevent attackers injencting falsified inputs to the planning which makes
                 this last plan wrong paths"}
            data element P3C5.1.1 {Asset.description = "Planning is the ML component that
                 defines the path and task to follow"}
            data element P3S1.1 { Method.name = "IntervalAnalysis"
                data element P3C2.1 { Method.description = "It is used to analyze and
                    verify the behavior of systems, particularly those involving
                    numerical computations and continuous variables"}
                data element  P3G2.1 { Requirement.formalReq = "formalSR5" Requirement.
                    name = "SR.5"
                data element P3Sn1.1 {
                        Artifact.link = "Path/To/Artifact/ExpertsFormalSpec.txt" }  }
                data element P3G3.1 { Requirement.formalReq= "formalSR5"
                data element P3Sn2.1 {
                        Activity.produce = "Path/To/Artifact/MLSRFormalSpec"
                        }  }    }     }
    .......
    }
```

Listing 7.9: Excerpt of the reference data for ACAS Xu system (Adversarial examples)

7.3.4 ML monitoring reference data

On the other hand, some ML-specific security requirements are not easily verifiable using formal methods because the nature of ML components makes it difficult to specify pre-

cise security properties. ML models operate in a probabilistic and data-driven manner, introducing complexities that traditional software does not encounter. Due to these challenges in formal specification and verification, monitoring becomes a crucial alternative. By continuously monitoring the ML components, it is possible to detect security violations in real-time, providing a practical means to ensure that the system behaves securely under various conditions. The fulfillment of security requirements that mitigate model extraction threats is guaranteed using monitoring policies as described in Listing 7.10

```
Activity Monitoring
    goal "Monitor the ML component to detect security violations"
    justification "Monitoring is an alternative verification activity, adapted to the ML
        particularities"
    phase ArchitecturalDesign
    type Design
    require {SoftwareArchitecture, MLSecurityRequirements, MonitoringPolicy}
    produce {MonitoringTest}
    carriedBy "Architect"
Method MonitoringQuery" It monitors the queries sent to the ML component in order to
    detect data extraction attack queries"
    executedBy VarDetect
Artifact SoftwareArchitecture
    link "Path/To/Artifact/SoftwareArchitecture.scml"
    type Documentation
    describeAsset SoftwareArchitecture
    language SCML
Artifact MLSecurityRequirements
    link "Path/To/Artifact/MLSecurityRequirements.scml"
    type Documentation
    language SCML
Artifact MonitoringPolicy
    link "Path/To/Artifact/MonitoringPolicy"
    type Documentation
Artifact MonitoringTest
    link "Path/To/Artifact/MonitoringTest"
    type TestingResults
```

Listing 7.10: Monitoring Activity

Consequently, Listing 7.11 depicts the reference data that contain values of the parameters related to the monitoring of the ML component.

```
reference data ACASXuRefData for Template MLBS
    data element P1G1 {
    .....
        data element G14 {
            data element S5 {
                data element A6 { Tool.name = "VarDetect"}
                data element G8 {
                    data element Sn2 { Artifact.link = "Path/To/Artifact/Standards"}
                    data element Sn3 { Artifact.link = "Path/To/Artifact/Experts"}  }
                data element G9 {
```

```
                data element Sn1 { Artifact.link = "Path/To/Artifact/
                    PerformanceMetric"}  }
            data element G10 {
                data element Sn4 { Artifact.link = "Path/To/Artifact/MonitoringTest"}
                        }
        }  }
    .....
    }
```

Listing 7.11: Excerpt of the reference data for ACAS Xu system (Monitoring)

Summary The diagram presented in Figure 7.1 is an adaptation of Figure 6.4 that shows the composition of the MLBS template using argument patterns. The new figure is extended within excerpts of the reference data related to each pattern.

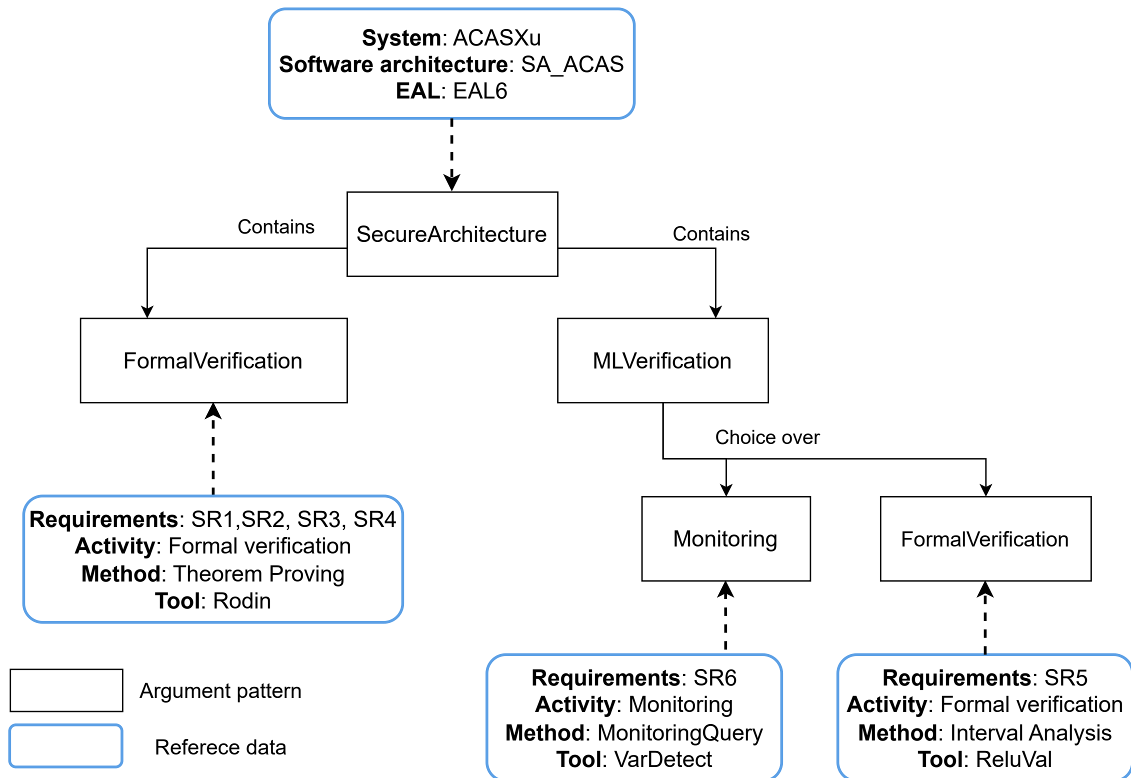


Figure 7.1: Patterns diagram with the reference data for the MLBS template

7.4 Security case for ACAS Xu

The security case of the system is presented in Listing 7.12. The underlined words point to security artifacts like the list of the security requirements and the description of the software architecture.

```
Assurance case ACASXu application  MLBS using data ACASXuRefData
metadata  "This is a security assurance case for the ACAS Xu software architecture"
      {
/*  SecureArchitecture pattern */
top goal P1G1  The software architecture  SA_ACAS  of the system  ACASXu  is
    secure in level EAL6 links {P1C1, P1C2, P1C3, P1S1}
strategy P1S1  Argue through verifying the security requirements  links{P1J1, P1G2,
    P1A4}
justification P1J1  A software architecture design is secure if it satisfies the
    security requirements
assumption P1A1  All the security requirements are identified
assumption P1A4  The system model is well defined
goal P1G2  The software architecture model satisfies the security requirements  links
    {P1A1, S2}
strategy S2  Argue through ML security requirements  links{J2, P2G5, P2G6}
justification J2  The focus in this template is the satisfaction of ML security
    requirements
/* Formal verification pattern */
goal P2G5  The software architecture satisfies the non-ML-specific security
    requirements  links {P2C4, P2S4}
strategy P2S4  Argue through each non ML security requirement  links{P2G1.X, P2J6,
    P2A1, P2A2, P2A3}
assumption P2A1  The system can be executed symbolically with symbolic inputs,
    allowing the exploration of multiple paths simultaneously
assumption P2A2  The formal model of the system is well-defined
assumption P2A3  The tool  Rodin  is correct
justification P2J6 Formal verifications required by the evaluation assurance level
    EAL6


/*  Security Requirement 1 */
goal P2G1.1  The architecture asset Sensors, connect1 satisfy  the security
    requirement SR1 links{P2S1.1, P2C1.1, P2C5.1}
strategy P2S1.1  Argue over  formal verification TheoremProver links{P2C2.1, P2G2.1,
    P2G3.1}
goal  P2G2.1  Formal requirement formalSR1 is the appropriate formal specification of
    the security requirement SR1 links{P2Sn1.1}
solution Sn1.1  Validation report by formalization experts: Path/To/Artifact/
    ExpertsFormalSpec.txt
goal  P2G3.1  The formal model satisfies the formal requirement SR1 links{P2Sn2.1}
solution Sn2.1   Results of formal verification: Path/To/Artifact/ProofObligation


/*  Security Requirement 2 */
goal P2G1.2  The architecture assets Sensors, Planning, Processor, Actuator satisfy
    the security requirement SR2 links{P2S1.2, P2C1.2, P2C5.2}
strategy P2S1.2  Argue over  formal verification TheoremProver links{P2C2.2, P2G2.2,
```

```
    P2G3.2}
goal  P2G2.2 Formal requirement  formalSR2 is the appropriate formal specification of
    the security requirement SR2 links{P2Sn1.2}
solution Sn1.2  Validation report by formalization experts: Path/To/Artifact/
    ExpertsFormalSpec.txt
goal  P2G3.2  The formal model satisfies the formal requirement SR2 links{P2Sn2.2}
solution Sn2.2   Results of formal verification: Path/To/Artifact/ProofObligation


/*  Security Requirement 3 */
goal P2G1.3   The architecture assets connect1, connect2, connect3 satisfy  the
    security requirement SR3 links{P2S1.3, P2C1.3, P2C5.3}
strategy P2S1.3   Argue over  formal verification TheoremProver links{P2C2.3, P2G2.3,
    P2G3.3}
goal  P2G2.3    Formal requirement formalSR3 is the appropriate formal specification
    of the security requirement SR3 links{P2Sn1.3}
solution Sn1.3  Validation report by formalization experts: Path/To/Artifact/
    ExpertsFormalSpec.txt
goal  P2G3.3  The formal model satisfies the formal requirement SR3 links{P2Sn2.3}
solution Sn2.3 Results of formal verification: Path/To/Artifact/ProofObligation


/*  Security Requirement 4 */
goal P2G1.4  The architecture assets Sensors, Planning, Processor, Actuator satisfy
    the security requirement SR4 links{P2S1.4, P2C1.4, P2C5.4}
strategy P2S1.4   Argue over  formal verification TheoremProver links{P2C2.4, P2G2.4,
    P2G3.4}
goal  P2G2.4  Formal requirement formalSR4 is the appropriate formal specification of
    the security requirement   SR4 links{P2Sn1.4}
solution Sn1.4  Validation report by formalization experts: Path/To/Artifact/
    ExpertsFormalSpec.txt
goal  P2G3.4  The formal model satisfies the formal requirement SR4 links{P2Sn2.4}
solution Sn2.4   Results of formal verification  Path/To/Artifact/ProofObligation


/* The verification of ML-specific security requirements */
goal G6  The software architecture satisfies ML-specific security requirements  links
    {C7, S9}
strategy S9  Argue through the choice of the verification method  links{G12, G11, G14
    }


/*  Formal verification of ML-specific security requirements   formalVerification
    pattern */
goal G12  The  ML-specific security requirements are formally verified  links{S3}
strategy S3  Argue through each ML security requirement  links {P3G1.1, P3A1, P3J6,
    P3A2}
assumption P3A1 " The ML inputs must have clear, finite ranges where each input
    feature can be expressed as an interval"
assumption  P3A2 The tool ReluVal is correct
justification P3J6  "Formal verification is required by the evaluation assurance
    level EAL6 "
goal P3G1.1  The architecture asset Planning satisfies  the security requirement SR.5
    links{P3S1.1, P3C1.1, P3C5.1}
```

```
strategy P3S1.1   Argue over  formal verification   IntervalAnalysis links{P3C2.1,
    P3G2.1, P3G3.1}
goal  P3G2.1 Formal requirement formalSR5 is the appropriate formal specification of
    the security requirement  SR5 links{P3Sn1.1}
solution P3Sn1.1   Validation report by formalization experts: Path/To/Artifact/
    ExpertsFormalSpec.txt
goal  P3G3.1   The formal model satisfies the formal requirement formalSR5 links{
    P3Sn2.1}
solution P3Sn2.1  Results of formal verification:  Path/To/Artifact/MLSRFormalSpec


/* Monitoring the ML-specific security requirements */


goal G14   Monitoring the ML component satisfies the ML-specified security
    requirements  links{S5}
strategy S5  Argue through the monitoring policy  links{J7, G8, G9, G10, A6}
Assumption A6   The tool  VarDetect  is correct
justification J7   Monitoring policies help ensure that the model behaves safely
    under all circumstances, including during abnormal inputs or adversarial attacks
goal G8  The monitoring policy is well-defined  links{Sn2, Sn3}
solution Sn2    List of the standards used to define the policies Path/To/Artifact/
    Standards
solution Sn3    Qualification of the experts who defined the monitoring policy: Path/
    To/Artifact/Experts
goal G9  The monitoring does not impact the system performance  links{Sn1}
solution Sn1    Results of performance metrics: Path/To/Artifact/PerformanceMetric
goal G10  The monitoring detects extraction queries  links{Sn4}
solution Sn4    Test results of the monitoring policy:   Path/To/Artifact/
    MonitoringTest
/* Contextual assurance elements  */
context P1C1 The system description Path/To/Artifact/SystemDescription.scml
context P1C2   The software architecture model: Path/To/Artifact/SoftwareArchitecture
    .scml
context P1C3 The software architecture is defined according to   EAL6
context P2C1.1  Description of the security requirement SR1 "The drone system should
    be able to ensure that the sensors informations are genuine and have not been
    intentionally or unintentionally altered"
context P2C1.2  Description of the security requirement SR2 "All the components of
    the system should be guaranteed to perform their required functions under defined
     spatial and temporal circumstances"
context P2C1.3  Description of the security requirement SR3 " The drone system should
     comply mechanisms to mitigate unauthorized disclosure of the information during
    its transmission"
context P2C1.4  Description of the security requirement SR4 "The system should be
    able to guarantee the authenticity of the software components"
context P2C1.5  Description of the security requirement SR5 "The drone system have to
     prevent attackers injencting falsified inputs to the planning which makes this
    last plan wrong paths"

context P2C2.1  Description of the method TheoremProver "It is a deductive formal
    method, used for the formal verification of security requirements"
context P3C2.1  Description of the method   IntervalAnalysis "It is used to analyze
    and verify the behavior of systems, particularly those involving numerical
```

```
    computations and continuous variables"
context P2C4  The list of non ML-specific security requirements: Path/To/Artifact/
    NonMLSecurityRequirement.scml
context C7  The list of ML-specific security requirements: Path/To/Artifact/
    MLSecurityRequirement.scml
context P2C5.1.1 Description of Sensors "Sensors collect different information like:
    GPS, velocity"
context P2C5.1.2 Description of connect1 "connect1 is the connector linking the
    sensors to the Processor component"
context P2C5.2.1 Description of Planning "Planning is responsible for route and task
    planning "
context P2C5.2.2 Description of  Processor "Processor is the central computing unit
    that processes data received from the sensors "
context P2C5.2.3 Description of Actuator "Actuator translates the planning decisions
    into physical actions"
```

Listing 7.12: Security case for ACAS Xu system

7.5 Discussions

In this section, we discuss the assessments and potential applications of the proposed approach, as well as the potential for its generalization and extension.

7.5.1 Key features of the approach

The proposed approach exhibits several features to assist non-savvy engineers, e.g., system architects, in building secure architectures via capturing and integrating security assurance expertise in the System Engineering process. In line with these aspects, contributions are made in the scope of the approach to model and analyze security assurance models in unison. The distinctive features of the approach herein proposed lay in following a set of recommendations proposed in [61] to advance toward more effective security assurance cases:

- **Modular security assurance** Developing modular and incremental security cases aids in minimizing the time, effort, and resources expended on generating them. The suggested approach promotes modular security cases through reusable templates, which are crafted by composing modular pattern arguments. Utilizing these argument patterns facilitates swift adaptation to changes in the system under assurance, as only the relevant argument pattern needs updating.
- **Effectiveness documented demonstrations** Determining which activities, techniques, or tools are suitable for addressing security challenges in the development

of high-assurance software-dependent systems remains a challenge for practitioners. Therefore, it is essential to provide demonstrations showing whether proposed solutions are broadly applicable or limited and not recommended. Our approach defines a set of semantic rules that mandate documentation as justification for each choice. For example, each technique is linked to an evaluation assurance level that necessitates its use, serving as justification for the technique's selection. These rules assist practitioners in making informed decisions about when and how to apply specific activities, methods, techniques, and tools.

- **Support tool integration** In the context of security case creation, the different approaches require specialized tools, such as editors and modelers, for creating security cases, as well as tools for generating supporting evidence. Our approach is presented with a tool designed to support the methods advocated by the approach. This tool seamlessly incorporates the depiction of activities within the security engineering process, and their associated artifacts, into the development of security assurance models.

- **Collaboration among stakeholders in the SDLC** Creating security cases involves several stakeholders with different backgrounds and expertise. Therefore, we need to include and improve the collaboration between developers and other critical stakeholders, such as domain experts and regulators, in the process of engineering high-assurance software-dependent systems. We knowingly suggest a an application scenario of the approach from the stakeholders' perspective 3.1. We map out the roles and responsibilities, showing how each stakeholder contributes to different aspects of the approach. This allocation of activities ensures a clear understanding of individual duties and collaborative efforts among the stakeholders.

- **Security and assurance awareness training** Creating security cases necessitates a high level of expertise, presenting a significant challenge due to the specialized knowledge required. There is a pressing need for comprehensive training and education for individuals involved in this process to ensure effective security case development. Our proposed approach addresses this issue by utilizing security argument patterns that encapsulate expert knowledge. By recording and standardizing this expertise within argument patterns, we reduce the overall expertise required to generate security cases. This allows us to shift the focus of training and education towards the creation and utilization of these security patterns, ensuring that all involved profiles are adequately prepared to contribute effectively to the development

process.

A concrete summary of our work regarding the existing state-of-the-art discussed in Chapter 2 based on the characterization attributes used therein, is provided in Table 7.2.

Stage	**Argumentation**	**Evidence**	**Tool**	**Case study**	**Reuse.**	**Eval.**
Design	Security Engineering activities	Security Engineering artifacts	SCML editor based on MDE	ACAS Xu based on Machine learning	Catalogue of argument patterns and templates	Structure and content evaluation

Table 7.2: Our result: Model-based approach for Security Case Development.

7.5.2 Applications of the proposed approach

The proposed approach yields a number of useful capabilities for system architects and assurance case developers. First, it enables them to verify security bugs early in the development of a system. Correcting software errors and defects, especially those related to system security, at late stages of development is very costly. By employing the proposed approach, system architects, while collecting the evidence that support the security case, can take corrective actions before the implementation phase.

Second, it enables them to better understand their secure architectures. The approach provides a clear trace of the security activities, their artifacts and the security requirements. Ultimately, the proposed approach helps system architects and designers to study the interplay between security requirements, threats, and mitigations. By employing the proposed approach, system architects and designers can determine whether they have a complete set of security requirements that can effectively address the known threats in their systems.

Furthermore, the approach guide the architects to follow security evaluation and assurance activities by providing evidence that systematic analyses have been performed at the Architecture Design phase of the SDLC and that identified threats have been adequately mitigated through the introduction of security requirements that are satisfied by the modeled software architecture.

7.5.3 Generalization of the proposed approach

We have demonstrated the practical application of the proposed approach in the context of the Royce waterfall model (SDLC) where we used STRIDE and PASTA threat modeling methods, model checking as a formal verification method for the definition of argument patterns. Moreover, we used SCADA as a domain application and machine learning as technology development for the definition of argument templates.

However, as described in Chapter 3 the proposed approach can be generalized, enabling it to be applied in alternative SDLCs and using other security engineering activities, domain applications, and technology development.

For example, we can envision the proposed approach to be applied in another SDLC, such as an Agile process with additional methods for the threat modeling (e.g., CAPEC) and formal verification (e.g., theroem proving) which will be used to define new argument patterns. In addition, we can extend the SDLC within domain-specific threat model (e.g., IoT-based big data environment [122], MPSoC-based embedded applications [110], etc.) which will help to define new argument templates. Such a realization of the proposed approach would simply require a new definition of the DSL using appropriate domain-specific modeling environments. After applying the approach with the accompanying tool support, the resulting reusable security argument patterns and templates can then be incorporated to create security cases as part of the Agile development process in future sprints.

With this generalization, the proposed approach can find use among practitioners familiar with a wide variety of model-driven engineering tools and environments.

7.6 Conclusion

In this chapter, we have demonstrated the application of our contributions to ensure the security of the architecture of our use case. First, we present the secure architecture of the system based on the secure reference architecture of MLBS. Then, we collect the artifacts that describe the security engineering activities conducted during the architecture design architecture. We also collect the artifacts produced by the previous activities. All these data, will be used to define the reference data that will be used lastly to apply the template.

In conclusion, this chapter has illustrated the effectiveness of our methodology in creating a security assurance case for the system ACAS Xu. By leveraging predefined patterns and constructing a tailored argument template, we provided a clear, systematic,

and repeatable process for security assurance. This approach not only strengthens the security architecture of ACAS Xu but also offers a scalable framework that can be applied to other systems within the same domain, enhancing overall security assurance practices.

Part IV

Conclusion

Chapter 8

Conclusion & future works

Contents

8.1 Summary and contributions 155
8.2 Limitations and future works 157
8.3 Perspectives 158

This last chapter is organized as follows. Section 8.1 highlights and assesses the contributions that are made by this thesis. Section 8.2 suggests avenues for future work resulting from the proposed framework and its applications and tools. Finally, Section 8.3 concludes with some perspectives.

8.1 Summary and contributions

In this thesis, we addressed the Problems concerning a lack of methodological support for an integrated software security assurance, reconciling respective domain expertise during the early design stages of the system engineering process. The inherent complexities of standalone security engineering and software assurance practices and semantic gaps pose difficulties in conducting a consistent analysis considering their mutual influence. To this end, we proposed a model-based approach and tooled framework to better support and ease the definition of security assurance cases. The approach has been instantiated in the context of machine learning-based systems. It relied upon 1) the Model-Driven Engineering (MDE) paradigm allowing the use of models for conducting the system design incorporating security assurance, and 2) the rigorousness of formal-based techniques facilitating security argument models analysis. Accordingly, the specific contributions of this work are five-fold, as shown in Table 8.1 and summarized in the following sub-sections.

Table 8.1: Summary of the Thesis Contributions.

Problem	Contribution	Language/Tool/Technique Used
P1: Lack of approach to facilitate incorporation of security assurance cases in the system engineering process	**C1:** A method to build security cases modeling framework, supporting the analysis of the security argument models (Chapter 3)	Model-Driven Engineering (MDE) and formal-based techniques
P2: Lack of modeling language, consolidating system, security, and assurance domain concepts specification in unison	**C2:** A Domain-Specific Modeling Languages (DSML) for security assurance of the software architectures. It is extended with semantic rules to guide the security case definition process. (Chapter 4)	Unified Modeling Language (UML) + Xtext grammar + Object Constraint Language (OCL)
P3: Error-prone design-stage security assurance and lack of expertise in creating non-refutable security arguments	**C3:** Methods to create security argument patterns and argument templates. Methods to create security cases re-using argument patterns and templates. (Chapter 5)	Conceptualization via step-centric methodologies
P4: Time-consuming design-stage security assurance and lack of reusable argument models.	**C4:** A catalogue of reusable argument patterns and argument templates created within the developed framework (Chapter 6)	SCML editor
P5: Lack of automated tool support for integrated security case definition	**C5:** An operational tool support integrating MDE and formal-based techniques (Chapter 5)	Eclipse , OCLinEcore

In Chapter 2, we relate a systematic review of the state-of-the-art approaches to security assurance concerning the problems addressed in this research work. Specifically, we focus on existing contributions creating and analyzing security assurance cases. Based on the extraction of the features they exhibit, a lack of methodological support for an integrated assurance of security is observed, which provides a rationale for the contributions of this work.

In Chapter 3, we provide an overview of our approach to tackle the identified lack of effective security assurance methods. One of the main contributions is providing a method to build a security assurance modeling framework, supporting the security engineering process. This also includes a tool-chain prototype architecture relying upon existing MDE and formal-based techniques to support the different phases of the approach, followed by an introduction to the ACAS use case to illustrate the approach's applicability. Moreover, we construct a conceptual model that offers concepts for security cases.

In Chapter 4 we present the security case modeling framework. First, based on the conceptual model, we define a modeling language to assist security assurance modeling. The framework gives the stakeholders a concrete syntax for modeling security assurance cases. The framework is completed by semantic rules for analyzing the security assurance models.

In Chapter 5, we detail methodologies that are proposed in the framework to build security cases. A Scratch methodology allows for building security case models, and reuse-based methodologies for building security cases by applying existing patterns and templates.

In Chapter 6, we develop a catalogue of reusable argument patterns and templates

created using the framework.

In Chapter 7, we illustrate the application of the proposed approach and the developed tool to generate security assurance cases of *ACAS*, reusing the developed security assurance argument patterns and templates.

Finally, Chapter 8 concludes the dissertation and proposes some future works and perspectives.

8.2 Limitations and future works

The proposed approach and tooled-framework aim to provide methodological support to assist non-savvy architects in secure system architecture assurance. Despite the features they offer, we identified certain lacks and, accordingly, defined the following future objectives to consolidate our work:

1. *Patterns:* Possible improvements can be added to foster reusing argument patterns to create security cases. We plan to define more argument patterns and expand the catalogue by identifying and defining additional patterns related to secure architectures. Moreover, it is important to study the impact of changes in system architecture. Changes occur for several reasons: new or changed requirements, performance improvement, system expansion to accommodate more users, technological advances in the platform devices and others.

2. *Templates:* Regarding the argument templates, we plan to define more argument templates to consider specialized applications of systems built using the same technology. Some applications have unique requirements, and building them using a general approach may not result in the most secure system. Typical applications that require their models include financial, medical, transportation, and smart grid applications. The application environment is another factor that impacts the argument template definition, which we plan to consider in future templates. Some types of environments have a large effect on the applications running on them, and methodologies must be modified to consider that effect.

3. *Tool support:* Regarding the tool-chain support prototype architecture described in Sections 5.3.2, 5.4.3, 5.5.3, we seek to study the integration of the proposed approach with other modeling frameworks and tools based upon MDE and formal-based techniques to support the following:

   (a) *Automation:* We would like to improve manual and semi-automated parts of the tool with more automation. For example, we plan to automatize the generation of reference data and, thus, apply the argument patterns and templates. The aim is to simplify the system security assurance modeling task conforming to the DSML meta-models. This can be achieved by developing a textual or graphical editor using Xtext [12] for instance.

   (b) *Model transformation:* We plan to incorporate transformation engines in the functionality of the tool-chain support architecture for 1) automatic or incremental [15] generation of the graphical and tabular representation of security cases, and 2) reporting mechanism to the end-user regarding any violation of the semantic rules.

   (c) *Reuse.* We plan to improve the tool support to cover more aspects of the Krueger [70] fundamental reuse issue. For instance, we seek to handle the storage and the selection of the reusable models [47].

4. *Assessment:* Shortly, we plan to develop a systematic evaluation framework to assess the effectiveness, relevance, and usability of the proposed approach, identified patterns and templates. This could involve quantitative metrics, case studies, or expert evaluations.

5. *Approach Generalization:* We will seek new opportunities to apply the proposed approach to other domains. This task requires instantiating the complete software engineering tool and method and evaluating the experiences of many users across many domains. We would like to enhance the proposed approaches for the integration with other model-based approaches, asset-based approaches, standard-based approaches, architecture models, security models, formal methods,

8.3 Perspectives

Perspectives emerging from this thesis are manifold. These perspectives are long-term objectives and consist of enhancements related to the proposed approach concerning the following aspects:

1. **Implementation phase:** Our approach focused on the architectural design phase of the software engineering process. However, security issues due to the violation of security countermeasures may also arise once the system is implemented [102], which is an even more complex undertaking. In such cases, testing practices can

reveal some of them. This requires extending the approach with code generation facilities and complying with the domain standards to move to the implementation phase. Accordingly, we need to inspect a way to model the system under assurance while being implemented. Moreover, further work is needed to collect the evidence elements resulting from the security engineering activities at the code level.

2. **Modeling language:** In this thesis, the SCML language proposed is based on GSN notation extended with the pattern and template notations. However, the recent version of GSN standard [3] defined two new extensions. We plan to add the GSN modular extension that will allow the division of an overall argument structure into separate argument structures focusing on particular aspects of the overall argument. For example, some assumptions will be presented as *away assumpiton* which means that there is a self-contained unit of an argument that can be developed and verified independently to assure the targeted assumption. Additionally, the GSN confidence argument notation can be used to indicate that a confidence argument is associated with an assertion in a risk argument. For example, an assurance case includes a goal stating that "The system is secure against unauthorized access." The corresponding evidence might include results from penetration testing. Using the confidence notation, the assurance case can also include a confidence node that states, "High confidence due to comprehensive testing covering all critical components," with an accompanying justification explaining the thoroughness and independence of the testing process.

3. **Integration of formal methods results** Formal model-based assurance cases are the ones that contain a formal model from which evidence for the top-level claims is derived. Formal methods refer to mathematical techniques in designing and analyzing computer hardware and software [98]. Certification authorities recognize using formal methods as an integral part of the development process of critical systems [14]. We aim to include an automatic generation of security cases reusing the formal verification activity description and collecting the formal verification results to use them as evidence elements.

4. **Standardization and certification:** In this thesis, we relied upon a model-based approach to create security cases. However, an important aspect of security assurance, certification, and evaluation comes from standards and guidelines related to the secure development of systems and their constituent components. These standards are domain-independent. For example, in the domain of smart electricity

grids, standards and guidelines outlining the minimum security requirements for smart grid components have been published by the IEC [58], and NIST [86]. On the one hand, we plan to align the argumentation and the evidence to the standard processes and procedures. Standards and guidelines can promote trends and actions that lower barriers to adopting more integrated, security-by-design approaches, ensuring that the costs of security assurance are justified and evaluated fairly. On the other hand, certification of security-critical systems is of utmost importance and standards (e.g., ISO 26262 [2], ISO/SAE 21434 [4], EN 50128 [20], and the UK Defence Standard 00-56 [1]) require providing security assurance cases to support qualification of such systems. Accordingly, we need to investigate the integration of our approach into certification processes. Aligning assurance cases to prescriptive standards is already proposed for safety cases in works [40] [26]

5. **Evaluation of security cases** Existing assurance-case tools primarily automate syntactic analysis, focusing on structural completeness, while providing limited or no support for semantically evaluating the logical aspects of the assurance case. We aim to include more evaluation criteria, such as confidence, which serves as a crucial metric for gauging trust in the underlying reasoning and supporting evidence. Another potential criterion is related to the consistency of the arguments. The goal is to avoid providing contradictory justifications.

Bibliography

[1] UK Defence Standard 00-56: Safety Management Requirements for Defence Systems, 2007.

[2] ISO 26262 Road vehicles – Functional safety, 2018.

[3] GSN Community Standard Version 3, 2021.

[4] ISO/SAE 21434 Road vehicles – Cybersecurity engineering, 2021.

[5] EUROCAE ED 275. Minimum operational performance standards for airborne collision avoidance system xu (acas xu) volume i, volume ii algorithm design description. European Organization for Civil Aviation Equipment (EUROCAE), 2020.

[6] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in event-b. *International journal on software tools for technology transfer*, 12(6):447–466, 2010.

[7] Robert David Alexander, Richard David Hawkins, and Timothy Patrick Kelly. From safety cases to security cases. 2017.

[8] Riham Altawy and Amr M Youssef. Security, privacy, and safety aspects of civilian drones: A survey. *ACM Transactions on Cyber-Physical Systems*, 1(2):1–25, 2016.

[9] Robert C Armstrong, Ratish J Punnoose, Matthew H Wong, and Jackson R Mayo. Survey of existing tools for formal verification. *SANDIA REPORT SAND2014-20533*, 2014.

[10] Rob Ashmore, Radu Calinescu, and Colin Paterson. Assuring the machine learning lifecycle: Desiderata, methods, and challenges. *ACM Computing Surveys (CSUR)*, 54(5):1–39, 2021.

[11] Sundramoorthy Balaji and M Sundararajan Murugaiyan. Waterfall vs. v-model vs. agile: A comparative study on sdlc. *International Journal of Information Technology and Business Management*, 2(1):26–30, 2012.

[12] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[13] J. Bézivin. Towards a precise definition of the OMG/MDA framework. In *Proceedings of ASE*, pages 273–280. IEEE Computer Society Press, 2001.

[14] Ben Brosgol and Cyrille Comar. Do-178c: A new standard for software safety certification. Technical report, ADA CORE TECHNOLOGIES NEW YORK NY, 2010.

[15] Thomas Buchmann. Bxtend-a framework for (bidirectional) incremental model transformations. In *MODELSWARD*, pages 336–345, 2018.

[16] RA Bull and K Segerberg. Basic modal logic. handbook of philosophical logic, v. ii, 1984.

[17] Fran Buschmann, Kelvin Henney, and Douglas C Schmidt. *Pattern-oriented Software Architecture: a Pattern Language for Distributed Computing, Volume 4*, volume 4. John Wiley & Sons, 2007.

[18] Carmen Cârlan, Lydia Gauerhof, Barbara Gallina, and Simon Burton. Automating safety argument change impact analysis for machine learning components. In *2022 IEEE 27th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 43–53. IEEE, 2022.

[19] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE symposium on security and privacy*, pages 39–57. IEEE, 2017.

[20] CENELEC. EN 50128: Railway Applications - Communications, Signalling and Processing Systems - Software for Railway Control and Protection Systems, 1999.

[21] Xi Chen, Lei Qiao, Hongbiao Liu, Zhi Ma, and Jingjing Jiang. Security verification method of embedded operating system semaphore mechanism based on coq. In *2021 2nd International Conference on Big Data & Artificial Intelligence & Software Engineering (ICBASE)*, pages 392–395. IEEE, 2021.

[22] Khana Chindamaikul, Toshinori Takai, and Hajimu Iida. Retrieving information from a document repository for constructing assurance cases. 2014.

[23] Thomas Chowdhury, Alan Wassyng, Richard F Paige, and Mark Lawford. Criteria to systematically evaluate (safety) assurance cases. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 380–390. IEEE, 2019.

[24] Luis-Pedro Cobos, Alastair R Ruddle, and Giedre Sabaliauskaite. Cybersecurity assurance challenges for future connected and automated vehicles. In *Proceedings of the 31 st European Safety and Reliability Conference (ESREL 2021)*, 2021.

[25] Alberto Rodrigues Da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155, 2015.

[26] Raghad Dardar, Barbara Gallina, Andreas Johnsen, Kristina Lundqvist, and Mattias Nyberg. Industrial experiences of building a safety case in compliance with iso 26262. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops*, pages 349–354, 2012.

[27] M Ann Garrison Darrin and William S Devereux. The agile manifesto, design thinking and systems engineering. In *2017 Annual IEEE International Systems Conference (SysCon)*, pages 1–5. IEEE, 2017.

[28] David Déharbe, Pascal Fontaine, Yoann Guyot, and Laurent Voisin. Integrating smt solvers in rodin. *Science of Computer Programming*, 94:130–143, 2014.

[29] Hervé Delseny, Christophe Gabreau, Adrien Gauffriau, Bernard Beaudouin, Ludovic Ponsolle, Lucian Alecu, Hugues Bonnin, Brice Beltran, Didier Duchel, Jean-Brice Ginestet, et al. White paper machine learning in certified systems. *arXiv preprint arXiv:2103.10529*, 2021.

[30] Ewen Denney, Ganesh Pai, and Josef Pohl. Advocate: An assurance case automation toolset. In *Computer Safety, Reliability, and Security: SAFECOMP 2012 Workshops: Sassur, ASCoMS, DESEC4LCCI, ERCIM/EWICS, IWDE, Magdeburg, Germany, September 25-28, 2012. Proceedings 31*, pages 8–21. Springer, 2012.

[31] Éric Dubois, Patrick Heymans, Nicolas Mayer, and Raimundas Matulevičius. A systematic approach to define the domain of information system security risk manage-

ment. *Intentional perspectives on information systems engineering*, pages 289–306, 2010.

[32] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE software*, 19(1):42–51, 2002.

[33] E.B. Fernandez. *Security patterns in practice: Building secure architectures using software patterns*. Software Design Patterns. Wiley, ISBN 978-1-119-99894-5, 2013.

[34] Eduardo Fernandez-Buglioni. *Security patterns in practice: designing secure architectures using software patterns*. John Wiley & Sons, 2013.

[35] Anita Finnegan and Fergal McCaffery. Towards an international security case framework for networked medical devices. volume 9337, 2015.

[36] OWASP Foundation. Owasp threat dragon. https://owasp.org/www-project-threat-dragon/, 2024. Accessed: 2024-06-10.

[37] R. B. France and B. Rumpe. Domain specific modeling. *Software and System Modeling*, 4(1):1–3, 2005.

[38] Lidia Fuentes-Fernández and Antonio Vallecillo-Moreno. An introduction to uml profiles. *UML and Model Engineering*, 2(6-13):72, 2004.

[39] Andrew Gacek, John Backes, Darren Cofer, Konrad Slind, and Mike Whalen. Resolute: an assurance case language for architecture models. *ACM SIGAda Ada Letters*, 34(3):19–28, 2014.

[40] Barbara Gallina. A model-driven safety certification method for process compliance. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, pages 204–209, 2014.

[41] Erich Gamma. Design patterns: elements of reusable object-oriented software. *Person Education Inc*, 1995.

[42] Fernando Silvano Goncalves, David Pereira, Eduardo Tovar, and Leandro Buss Becker. Formal verification of aadl models using uppaal. In *2017 VII Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 117–124. IEEE, 2017.

[43] Divya Gopinath, Guy Katz, Corina S Păsăreanu, and Clark Barrett. Deepsafe: A data-driven approach for assessing robustness of neural networks. In *Automated Technology for Verification and Analysis: 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings 16*, pages 3–19. Springer, 2018.

[44] J. Gray, J-P. Tolvanen, S. Kelly, A. Gokhale, S. Neema, and J. Sprinkle. Domain-Specific Modeling. In P. Fishwick, editor, *Handbook of Dynamic System Modeling*, chapter 7, pages 1–20. Chapman & Hall/CRC, 2007.

[45] GSN Working Group. GSN community standard version 3. Available: `https://scsc.uk/r141C:1?t=1`, May 2021.

[46] Brahim Hamid. Modeling of secure and dependable applications based on a repository of patterns: the semco approach. *Reliability*, pages 9–17, 2014.

[47] Brahim Hamid. A model repository description language - MRDL. In Georgia M. Kapitsaki and Eduardo Santana de Almeida, editors, *Software Reuse: Bridging with Social-Awareness*, pages 350–367. Springer International Publishing, 2016.

[48] Brahim Hamid. A model-driven approach for developing a model repository: Methodology and tool support. *Future Generation Computer Systems*, 68:473–490, 2017.

[49] Muhammad Abdullah Hanif, Faiq Khalid, Rachmad Vidya Wicaksana Putra, Semeen Rehman, and Muhammad Shafique. Robust machine learning systems: Reliability and security for deep neural networks. In *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design*, pages 257–260. IEEE, 2018.

[50] D. Harel and B. Rumpe. Modeling Languages: Syntax, Semantics and All That Stuff Part I: The Basic Stuff. Technical report, 2000.

[51] Mark Harris. NTSB investigation into deadly uber self-driving car crash reveals lax attitude toward safety. *IEEE Spectrum*, November 2019.

[52] Richard Hawkins, Ibrahim Habli, Tim Kelly, and John McDermid. Assurance cases and prescriptive software safety certification: A comparative study. *Safety science*, 59:55–71, 2013.

[53] Richard Hawkins, Ibrahim Habli, Dimitris Kolovos, Richard Paige, and Tim Kelly. Weaving an assurance case from design: A model-based approach. volume 2015-January, 2015.

[54] Richard Hawkins, Colin Paterson, Chiara Picardi, Yan Jia, Radu Calinescu, and Ibrahim Habli. Guidance on the assurance of machine learning in autonomous systems (amlas). *arXiv preprint arXiv:2102.01564*, 2021.

[55] C Michael Holloway. Safety case notations: Alternatives for the non-graphically inclined? In *2008 3rd IET International Conference on System Safety*, pages 1–6. IET, 2008.

[56] Sanghyun Hong, Michael Davinroy, Yiğitcan Kaya, Stuart Nevans Locke, Ian Rackow, Kevin Kulda, Dana Dachman-Soled, and Tudor Dumitraş. Security analysis of deep neural networks operating in the presence of cache side-channel attacks. *arXiv preprint arXiv:1810.03487*, 2018.

[57] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30*, pages 3–29. Springer, 2017.

[58] IEC. IEC Standard: 62351.: International Electrotechnical Commission:, 2007.

[59] Standardization IOf. Iso 27799: 2016-health informatics–information security management in health using iso, 2016.

[60] Jason Jaskolka. Challenges in assuring security and resilience of advanced metering infrastructure. In *Proceedings of the 18th Annual IEEE Canada Electrical Power and Energy Conference*, EPEC 2018, pages 1–6, Toronto, ON, Canada, 2018. IEEE.

[61] Jason Jaskolka. Recommendations for effective security assurance of software-dependent systems. In *Intelligent Computing: Proceedings of the 2020 Computing Conference, Volume 3*, pages 511–531. Springer, 2020.

[62] Jason Jaskolka, Brahim Hamid, Alvi Jawad, and Joe Samuel. Secure development decomposition – an argument pattern for structured assurance case models. In *Proceedings of the 28th Conference on Pattern Languages of Programs*, PLoP 2021, pages 1–12, Online, 2021.

[63] Jason Jaskolka, Alvi Jawad, Joe Samuel, and Brahim Hamid. A security property decomposition argument pattern for structured assurance case models. In *26th European Conference on Pattern Languages of Programs*, pages 1–10, 2021.

[64] Jason Jaskolka, Alvi Jawad, Joe Samuel, and Brahim Hamid. A security property decomposition argument pattern for structured assurance case models. In *Proceedings of the 26th European Conference on Pattern Languages of Programs*, pages 1–10, 2021.

[65] Florian Kammüller. A formal development cycle for security engineering in isabelle. *arXiv preprint arXiv:2001.08983*, 2020.

[66] Tim P Kelly. Arguing safety-a systematic approach to safety case management. *DPhil Thesis York University, Department of Computer Science Report YCST*, 1999.

[67] Kristine Kiernan, Robert Joslin, and John Robbins. Standardization roadmap for unmanned aircraft systems, version 2.0. 2020.

[68] Barbara Kitchenham. Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33(2004):1–26, 2004.

[69] A. G. Kleppe. A language description is more than a metamodel. In *Fourth International Workshop on Software Language Engineering*, 2007.

[70] C.W. Krueger. Software Reuse. *ACM Compututing Survey*, 24(2):131–183, 1992.

[71] Ram Shankar Siva Kumar, David O Brien, Kendra Albert, Salomé Viljöen, and Jeffrey Snover. Failure modes in machine learning systems. *arXiv preprint arXiv:1911.11034*, 2019.

[72] Nikola Luburić, Goran Sladić, Branko Milosavljević, and Aleksandar Kaplar. Demonstrating enterprise system security using an asset-centric security assurance framework. In *8th International Conference on Information Society and Technology*, volume 16, 2018.

[73] Franck Mamalet, Eric Jenn, Gregory Flandin, Hervé Delseny, Christophe Gabreau, Adrien Gauffriau, Bernard Beaudouin, Ludovic Ponsolle, Lucian Alecu, Hugues Bonnin, et al. *White paper machine learning in certified systems.* PhD thesis, IRT Saint Exupéry; ANITI, 2021.

[74] Guido Manfredi and Yannick Jestin. An introduction to acas xu and the challenges ahead. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–9. IEEE, 2016.

[75] V Manni et al. Baseline for the common certification language. *Open Platform for EvolutioNary Certification Of Safety-critical Systems (OPENCOSS), Report D*, 4:1, 2012.

[76] Laurent Mauborgne. Astrée: Verification of absence of runtime error. In *Building the Information Society: IFIP 18th World Computer Congress Topical Sessions 22–27 August 2004 Toulouse, France*, pages 385–392. Springer, 2004.

[77] Russ McRee. Microsoft threat modeling tool 2014: identify & mitigate. *ISSA Journal*, 39:42, 2014.

[78] Hui Miao, Ang Li, Larry S Davis, and Amol Deshpande. Modelhub: Deep learning lifecycle management. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1393–1394. IEEE, 2017.

[79] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Daniel Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, et al. Evolving deep neural networks. In *Artificial intelligence in the age of neural networks and brain computing*, pages 293–312. Elsevier, 2019.

[80] Threat Modeling. Threat modeling tool, aristiun. `https://threat-modeling.com/threat-modeling-tool/`, 2024. Accessed: 2024-06-10.

[81] Parastoo Mohagheghi and Vegard Dehlen. Where is the proof?-a review of experiences from applying mde in industry. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 432–443. Springer, 2008.

[82] Mazen Mohamad, Örjan Askerdal, Rodi Jolak, Jan-Philipp Steghöfer, and Riccardo Scandariato. Asset-driven security assurance cases with built-in quality assurance. In *2021 IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)*, pages 1–8. IEEE, 2021.

[83] Mazen Mohamad, Rodi Jolak, Örjan Askerdal, Jan Philipp Steghöfer, and Riccardo Scandariato. Cascade: An asset-driven approach to build security assurance cases for automotive systems. *ACM Transactions on Cyber-Physical Systems*, 7, 2023.

[84] Anas Motii, Agnès Lanusse, Brahim Hamid, and Jean-Michel Bruel. Model-based real-time evaluation of security patterns: A scada system case study. In *Computer Safety, Reliability, and Security: SAFECOMP 2016 Workshops, ASSURE, DECSoS, SASSUR, and TIPS, Trondheim, Norway, September 20, 2016, Proceedings 35*, pages 375–389. Springer, 2016.

[85] Kateryna Netkachova, Kevin Müller, Michael Paulitsch, and Robin Bloomfield. Investigation into a layered approach to architecting security-informed safety cases. 2015.

[86] NIST. Special Publication 800-82, Guide to Industrial Control Systems (ICS) Security, 2011.

[87] OMG. MetaObject Facility (MOF), Version 2.4.2. http://www.omg.org/spec/MOF/2.4.2/, 2014. [Accessed: January-2015].

[88] Robert Palin and Ibrahim Habli. Assurance of automotive safety–a safety case approach. In *Computer Safety, Reliability, and Security: 29th International Conference, SAFECOMP 2010, Vienna, Austria, September 14-17, 2010. Proceedings 29*, pages 82–96. Springer, 2010.

[89] Nathan Pape and Christopher Mansour. Pasta threat modeling for vehicular networks security. In *2024 7th International Conference on Information and Computer Technologies (ICICT)*, pages 474–478. IEEE, 2024.

[90] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18, 2017.

[91] Chiara Picardi, Colin Paterson, Richard David Hawkins, Radu Calinescu, and Ibrahim Habli. Assurance argument patterns and processes for machine learning in safety-related systems. In *Proceedings of the Workshop on Artificial Intelligence Safety (SafeAI 2020)*, pages 23–30. CEUR Workshop Proceedings, 2020.

[92] Débora Pina, Adriane Chapman, Daniel De Oliveira, and Marta Mattoso. Deep learning provenance data integration: a practical approach. In *Companion Proceedings of the ACM Web Conference 2023*, pages 1542–1550, 2023.

[93] Shirley Radack. The system development life cycle (sdlc). Technical report, National Institute of Standards and Technology, 2009.

[94] Arnab Ray and Rance Cleaveland. Security assurance cases for medical cyber-physical systems. *IEEE Design and Test*, 32, 2015.

[95] Mark Richters and Martin Gogolla. Ocl: Syntax, semantics, and tools. In *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, pages 42–68. Springer, 2002.

[96] Quentin Rouland, Brahim Hamid, Jean-Paul Bodeveix, and Mamoun Filali. A formal methods approach to security requirements specification and verification. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 236–241. IEEE, 2019.

[97] Quentin Rouland, Brahim Hamid, and Jason Jaskolka. Specification, detection, and treatment of stride threats for software components: Modeling, formal methods, and tool support. *Journal of Systems Architecture*, 117:102073, 2021.

[98] John Rushby. *Formal methods and the certification of critical systems*, volume 37. SRI International, Computer Science Laboratory, 1993.

[99] John Rushby. Logic and epistemology in safety cases. In Friedemann Bitsch, Jérémie Guiochet, and Mohamed Kaâniche, editors, *Computer Safety, Reliability, and Security*, pages 1–7, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[100] Marius Schlegel and Kai-Uwe Sattler. Management of machine learning lifecycle artifacts: A survey. *ACM SIGMOD Record*, 51(4):18–35, 2023.

[101] D. Schmidt. Model-Driven Engineering. *in IEEE computer*, 39(2):41–47, 2006.

[102] Christoph Schmittner, Zhendong Ma, and Erwin Schoitsch. Combined safety and security development lifecylce. In *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, pages 1408–1415. IEEE, 2015.

[103] SecurityWeek. Attackers alter water treatment systems in utility hack: Report, 2015. [Accessed: August-2024].

[104] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

[105] Nungki Selviandro, Richard Hawkins, and Ibrahim Habli. A visual notation for the representation of assurance cases using sacm. In *Model-Based Safety and Assessment: 7th International Symposium, IMBSA 2020, Lisbon, Portugal, September 14–16, 2020, Proceedings 7*, pages 3–18. Springer, 2020.

[106] Fahad Siddiqui, Rafiullah Khan, Sena Yengec Tasdemir, Henry Hui, Balmukund Sonigara, Sakir Sezer, and Kieran McLaughlin. Iso/sae 21434.

[107] Raymond M Smullyan. *First-order logic*. Courier Corporation, 1995.

[108] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[109] Kenji Taguchi, Daisuke Souma, and Hideaki Nishihara. Safe & sec case patterns. In *Computer Safety, Reliability, and Security: SAFECOMP 2015 Workshops, ASSURE, DECSoS. ISSE, ReSA4CI, and SASSUR, Delft, The Netherlands, September 22, 2015, Proceedings 34*, pages 27–37. Springer, 2015.

[110] B. Tan, M. Biglari-Abhari, and Z. Salcic. Towards decentralized system-level security for MPSoC-based embedded applications. *Journal of Systems Architecture*, 80:41–55, 2017.

[111] NCS TIB. 04-1; technical information bulletin 04–1, supervisory control and data acquisition (scada) systems. *National Communications System: Arlington, VA, USA*, 2004.

[112] Roman Trentinaglia. Deriving model-based safety and security assurance cases from design rationale of countermeasure patterns. 2022.

[113] Caterina Urban and Antoine Miné. A review of formal methods applied to machine learning. *arXiv preprint arXiv:2104.02466*, 2021.

[114] Olga Villagrán-Velasco, Eduardo B Fernández, and Jorge Ortega-Arjona. Refining the evaluation of the degree of security of a system built using security patterns. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, pages 1–7, 2020.

[115] Serena VILLATA, Guido BOELLA, and Leendert VAN DER TORRE. Argumentation patterns. *University of Luxembourg and Turin*, 2011.

[116] José Luis Vivas, Isaac Agudo, and Javier López. A methodology for security assurance-driven system development. *Requirements Engineering*, 16, 2011.

[117] Lars Vogel. Eclipse RCP, 2015. [Accessed: August-2016].

[118] Sicherheit von Informationstechnik. Common criteria. *Universität Karlsruhe*, page 95.

[119] Nikolaos S Voros, Wolfgang Mueller, and Colin Snook. An introduction to formal methods. In *UML-B Specification for Proven Embedded Systems Design*, pages 1–20. Springer, 2004.

[120] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal security analysis of neural networks using symbolic intervals. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1599–1614, 2018.

[121] Andrzej Wardziski and Aleksander Jarz bowicz. Automated generation of modular assurance cases with the system assurance reference model. *Formal Aspects of Computing.*

[122] M. Wazid, A. Kumar Das, R. Hussain, G. Succi, and Joel J.P.C. Rodrigues. Authentication in cloud-driven IoT-based big data environment: Survey and outlook. *Journal of Systems Architecture*, 97:185–196, 2019.

[123] Ludwig Weber. *International Civil Aviation Organization (ICAO).* Kluwer Law International BV, 2021.

[124] Ran Wei, Tim P Kelly, Xiaotian Dai, Shuai Zhao, and Richard Hawkins. Model based system assurance using the structured assurance case metamodel. *Journal of Systems and Software*, 154:211–233, 2019.

[125] Charles B Weinstock and Howard F Lipson. Evidence of assurance: laying the foundation for a credible security case. *Software Engineering Institute Report*, 2013.

[126] Biao Xu, Minyan Lu, and Dajian Zhang. A layered argument strategy for software security case development. 2017.

[127] Zhi Xu, Deming Zhong, Weigang Li, Hao Huang, and Yigang Sun. Formal verification of dynamic hybrid systems: a nusmv-based model checking approach. In *ITM Web of Conferences*, volume 17, page 03026. EDP Sciences, 2018.

[128] Mingfu Xue, Chengxiang Yuan, Heyi Wu, Yushu Zhang, and Weiqiang Liu. Machine learning security: Threats, countermeasures, and evaluations. *IEEE Access*, 8:74720–74742, 2020.

[129] Marwa Zeroual, Brahim Hamid, Morayo Adedjouma, and Jason Jaskolka. Towards logical specification of adversarial examples in machine learning, 2022.

[130] Marwa Zeroual, Brahim Hamid, Morayo Adedjouma, and Jason Jaskolka. Constructing security cases based on formal verification of security requirements in alloy. In *International Conference on Computer Safety, Reliability, and Security*, pages 15–25. Springer, 2023.

[131] Marwa Zeroual, Brahim Hamid, Morayo Adedjouma, and Jason Jaskolka. Formal model-based argument patterns for security cases. In *Proceedings of the 28th European Conference on Pattern Languages of Programs*, pages 1–12, 2023.

[132] Marwa Zeroual, Brahim Hamid, Morayo Adedjouma, and Jason Jaskolka. Security argument patterns for deep neural network development. In *Proceedings of the 30th PATTERN LANGUAGES OF PROGRAMS CONFERENCE*, pages 1–12, 2023.

Appendices

Appendix A

Security Engineering Activities

A.1 Threat modeling

A.1.1 STRIDE

Spoofing. Spoofing refers to the impersonation of a component in the system for the purpose of misleading other system entities into falsely believing that an attacker is legitimate. Spoofing threats violates the authentication objectives of a system. In the context of message passing communication, spoofing threats take the form of message senders falsely claiming to be other system components, to entice other components to believe that the spoofed component is the originator of the message.

Tampering. Tampering refers to the unauthorized modification of data. Tampering threats violates the integrity objectives of a system. They often involve the modification of data in transit. Therefore, a tampering threat can be identified by verifying whether the message that was sent by a sender is the same message that was received by the receiver, i.e., *a message is not altered in transit.*

Repudiation. Repudiation refers to a component claiming to have not performed an action that was in fact performed. Repudiation threats result from a lack of audit ability and accountability in the system. Mitigating repudiation threats requires an ability to separate legitimate claims from false claims made by system components. Very often, this involves constructing an audit log that records what happened during system operation and which components were involved. Therefore, a repudiation threat can be identified by verifying whether every sent or received message trace exists in the system, i.e., *for every sent or received message the system holds a component accountable for this action.*

Information disclosure. Information disclosure refers to the unauthorized exposure of information to a component for which it is not intended. Information disclosure threats violate the confidentiality objectives of a system. With consideration to message-passing communication, information disclosure threats occur when components other than those for which a message was intended are able to receive the sent message. Therefore, an information disclosure threat can be identified by verifying whether a component other than the intended receiver(s) is able to receive a message, i.e., *all messages are delivered only to the intended receiver(s).*

Denial of service. Denial of service refers to the unauthorized withholding of a service to system components. Denial of service threats violates the availability objectives of a system. In the context of message passing communication, denial of service threats may involve blocking the transmission of messages from senders to receivers such that the receiver never receives any of the messages before their freshness expires, i.e., before the deadline where the message becomes irrelevant is reached. Therefore, a denial of service threat can be identified by verifying whether sent messages are delayed, destroyed, or deleted in transit, preventing them from being received before the freshness of the message expires or from being received by the intended receiver at all.

Elevation of privilege. Elevation of privilege refers to the ability of a component to gain capabilities without proper authorization to have such capabilities. Elevation of privilege threats violate the authorization objectives of a system. Typically, an elevation of privilege threat involves a system component being able to perform an action for which they are not authorized according to the system access control policy. Very often, mitigating elevation of privilege attacks involves enforcing the system's access control policy. Therefore, an elevation of privilege threat can be identified by verifying whether it is possible for a component to perform any actions without having the proper authorizations, i.e., *every component that performs an action (send/receive) has the allowable permissions at the time the action is performed.*

A.1.2 P.A.S.T.A.

It is composed by seven stages:

1. **Definition of Objectives:** Identify the business and security objectives to align threat modeling efforts with the organization's goals.

2. **Technical Scope:** Define the technical architecture and the environment, including the technology stack, network infrastructure, and application components.

3. **Application Decomposition:** Break down the application to understand its structure, data flow, and interactions, which helps identify areas of potential risk.

4. **Threat Analysis:** Identify potential threats by examining the application from an attacker's perspective. Use threat libraries and attack vectors to uncover security risks.

5. **Vulnerability Analysis:** Examine the application for existing vulnerabilities that could be exploited based on the previously identified threats.

6. **Attack Modeling:** Simulate potential attack scenarios to understand how vulnerabilities might be exploited and assess the impact on the system.

7. **Risk Analysis and Management:** Evaluate the identified risks, prioritize them based on their potential impact, and develop a risk management plan to address and mitigate these threats.

A.2 Formal methods

Formal methods involve using techniques for mathematically analyzing the description of a system, including its properties, e.g., safety and security, via the application of formal specification languages. Formal specification involves using precise and unambiguous mathematical concepts, like logic and algebra, with added semantic rules that foster reasoning and verification of properties [119]. For example, First-Order Logic (FOL) [107] describes the system's state in the form of pre-conditions and post-conditions, and Modal logic [16] considers the interpretation of formula within a defined context using modal operators, e.g., possibility and necessity. Likewise, Event-B [6] is a formal language based upon set theory notions for specifying and analyzing systems by modeling them as abstract state machines. Formal-based tools, e.g., Rodin [28] and Alloy analyzer [96], are particularly useful in designing and verifying high-end industrial systems. They broadly cover 1) automated model checkers that rely upon algorithms for exhaustively verifying the desired properties of the system, given a model's state space, and 2) theorem provers (also proof assistants) that involve the use of human expertise for guiding the proof of correctness [9].

A.3 Evaluation Assurance Level

- **EAL1:** Functionally Tested - Basic assurance through function testing.
- **EAL2:** Structurally Tested - Requires a low to moderate level of independently conducted tests.
- **EAL3:** Methodically Tested and Checked - Requires a moderate level of testing and review.
- **EAL4:** Methodically Designed, Tested, and Reviewed - Requires substantial independent testing and design review.
- **EAL5:** Semiformally Designed and Tested - Involves semiformal design verification and thorough testing.
- **EAL6:** Semiformally Verified Design and Tested - Provides high assurance with extensive testing and formal verification.
- **EAL7:** Formally Verified Design and Tested - Highest level of assurance with formal methods for design and comprehensive testing.

Appendix B

Concrete Syntaxes

B.1 Reference Data

```
1  ReferenceData :=  "reference data" Id description "for" ("Pattern" [Pattern|Id] | "
       Template" [Template|Id]) "{" DataElement+ "}"
2
3  DataElement := "data element" [AssuranceElement|Id] "{"
4                   (Paramater "=" STRING )*
5                    DataElement* "}"
6
7  Parameter := [Concept|ID]"."Attribute
8  Concept := SecurityConcept | ProcessConcept | SystemConcept
9  Attribute := enumeration
10     }
```

Listing B.1: Excerpt of the textual syntax of SCML (Reference Data)

B.2 Process Package

```
1  ProcessConcept :=  Activity | Tool | Role | Method | Artifact
2
3  Activity := "Activity" Id descripiton ("goal" STRING)? ("assumption" STRING)? ("
       justification" STRING)? ("phase" TypePhase)? ("type" TypeActivity)?
4     ("require" [Artifact|Id]*)?  ("produce" [Artifact|Id]*)?  "carriedBy" [Role|Id]+ | (
       "include" [Activity|Id]*)?  ("use" [Method|Id]*)?
5
6
7  Artifact := "Artifact" Id descripiton "link" STRING ("type" TypeArtifact)? ("
       describeAsset" [Asset|Id])?
8  Role := "Role" Id descripiton "skill" STRING
9  Method := "Method" Id descripiton ("executedBy" [Tool|Id])?
10 Tool := "Tool" Id descripiton ("version" STRING)?
```

Listing B.2: Excerpt of the textual syntax of SCML (Process package)

B.3 System Package

```
1  SystemConcept := System | SoftwareArchitecture | Asset | ReferenceArchitecture
2
3  Asset := (ReferenceAsset | SoftwareAsset) "type" TypeAsset "connectedTo" [Asset|Id]* ("
       containother" [Asset|Id]+)? ("protectedBy" [SecurityPattern|Id]+)?
4
5  System := "System" Id description "domain" STRING "technology" STRING
6
7  SoftwareArchitecture:= "SoftwareArchitecture" Id description "generatedFrom" [
       ReferenceAsset|Id] "relateTo" [System|Id] use {" [SoftwareAsset|role]+ "}"
8
9  ReferenceArchitecture:= "ReferenceArchitecture" Id description "domain" STRING "
       technology" STRING "use {" [ReferenceAsset|Id]+ "}"
10
11 SoftwareAsset:= "SoftwareAsset" Id description? "concreteRole" STRING ("implement" [
       ReferenceAsset|role])?
12
13 ReferenceAsset:= "ReferenceAsset" Id description? "role" STRING
```

Listing B.3: Excerpt of the textual syntax of SCML (Process package)

B.4 Security Package

```
1  SecurityConcept := Requirement | SecurityPattren | EAL | Objective | Threat |
       Vulnerability
2
3  Requirement := "Requirement" Id description "fillfuledBy" [SecurityPattern|Id]+ ("
       allocatedToAsset" [Asset|Id]*)? ("formal requirement" STRING)? ("assignedTo" [EAL|Id
       ])?
4      ("realize" [Objective|Id]*)? ("mitigate" [Threat|Id]*)?
5
6
7  EAL := value "recommendMethod" [Method|Id]+
8
9  value := "EAL1" | "EAL2" | "EAL3" | "EAL4" |"EAL5" | "EAL6" | "EAL7"
10
11 Threat := "Threat" Id description ("class" STRING)?
12
13 SecurityPattren := "SecurityPattren" Id description ("mechanism" STRING)?
14
15 Objective := "Objective" Id description
16
17 Vulnerability:= "Vulnerability" Id description
```

Listing B.4: Excerpt of the textual syntax of SCML (Security package)

B.5 Security case for the toy example

Appendix C

Catalogue of argument patterns

C.1 Graphical presentation of the patterns presented in the manuscript

We present a graphical representation of the patterns already presented in the manuscript. The graphical representation are created using ADVOCATE [30].

C.1.1 SecureArchitecture argument pattern

C.1.2 STRIDEThreatModeling argument pattern

C.1.3 PASTAThreatModeling argument pattern

C.1.4 ModelChecking argument pattern

C.1.5 ModelWellFormedness argument pattern

C.1.6 FormalVerification argument pattern

C.2 Argument patterns from our previous works

In this section, we present argument patterns published presented in international conferences.

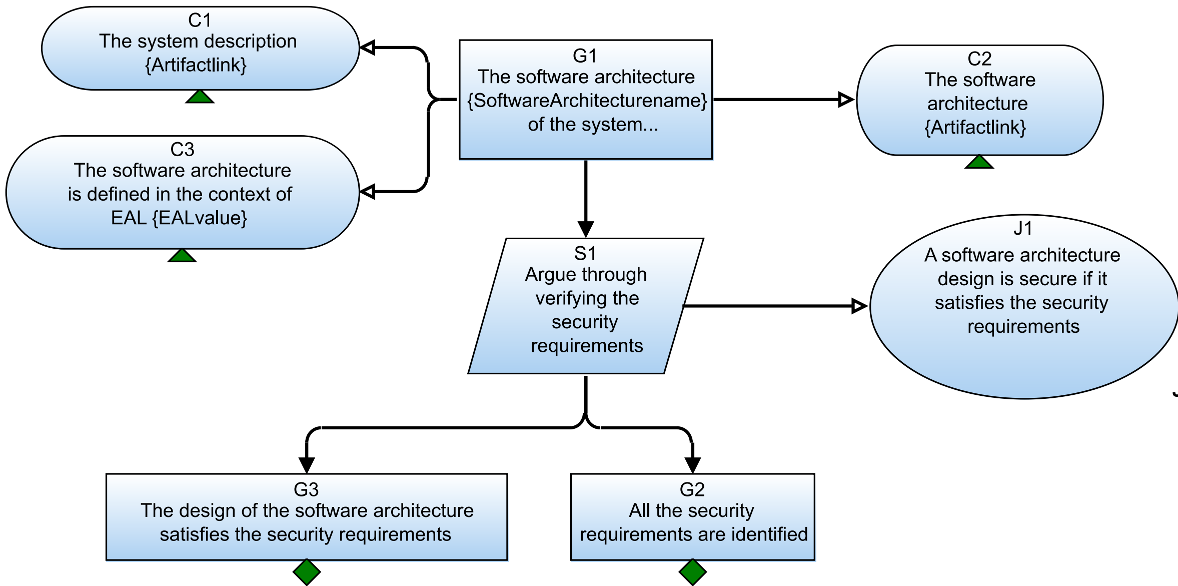


Figure C.1: Secure architecture argument pattern

C.2.1 Threat identification argument pattern

The pattern is taken from our previous work [131].
The goal of this pattern is to provide a convincing argument that when developing a secure system, if it can be shown that all the relevant threats have been identified and mitigated, then the system is acceptably secure [114]. This pattern is divided into two parts, as shown in Fig. C.7.

The initial part denoted by the red dashed box is inspired by the pattern proposed in [62]. Their pattern provides a claim decomposition to argue that if the system under consideration satisfies its asset protection requirements and secure development process requirements as prescribed by a relevant security standard, it is adequately secure and compliant with the standard. Our pattern starts with a claim that the system satisfies SRs G0. The argumentation in this pattern is based on the fact that a system is considered secure if it satisfies its SRs J0. Context node C0 provides additional information about the system functionality and the context node C1 refers to the list of SRs. Initially, we assume that the given list of SRs is completed and there is no interference between the SRs A0. Under the top-level claim, strategy node S0 argues that the system satisfies its SRs if it satisfies requirements for asset protection G1 and secure development process G2. We assume that an asset inventory is established A1. This includes identifying

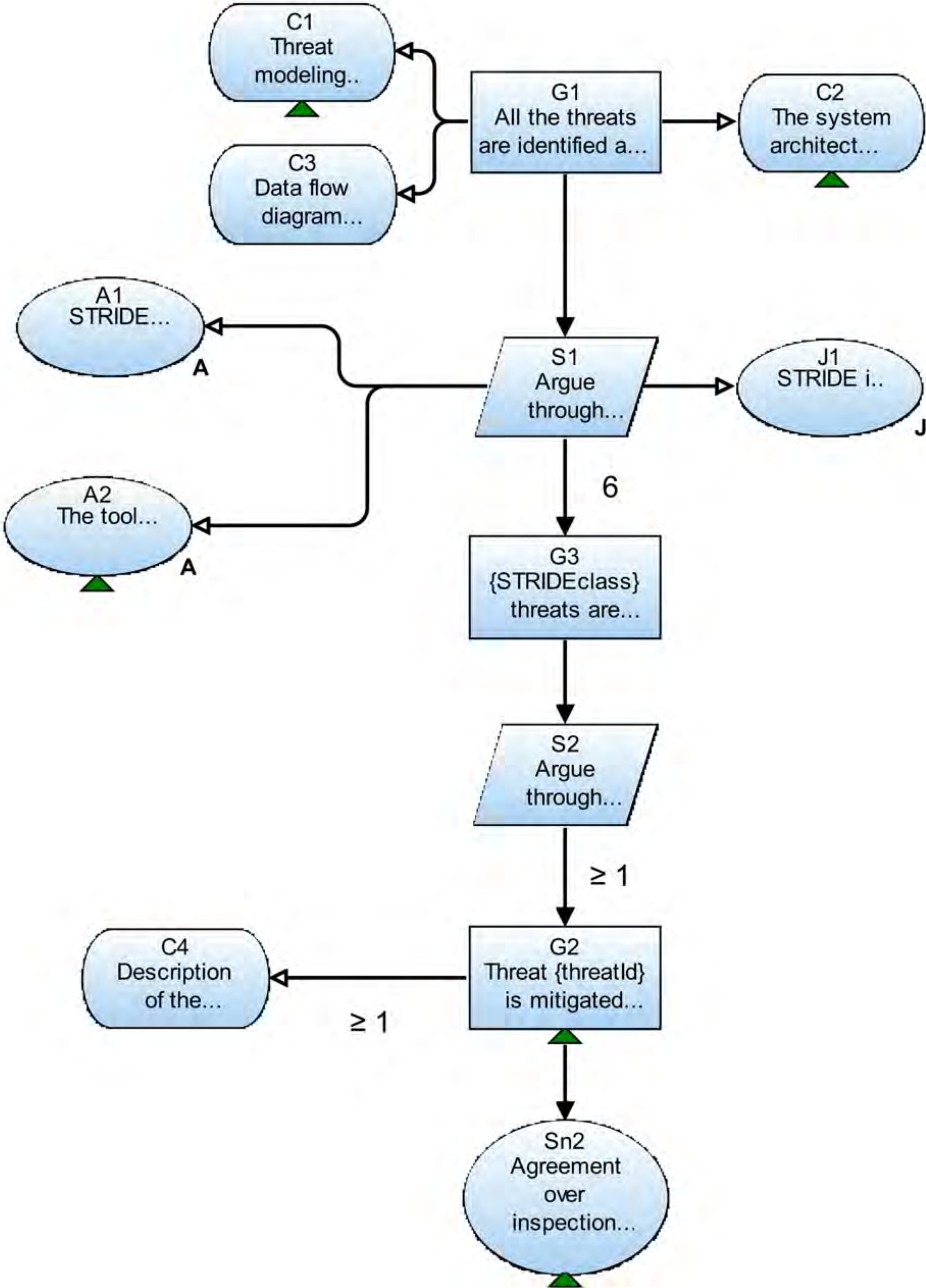


Figure C.2: STRIDE threat modeling argument pattern

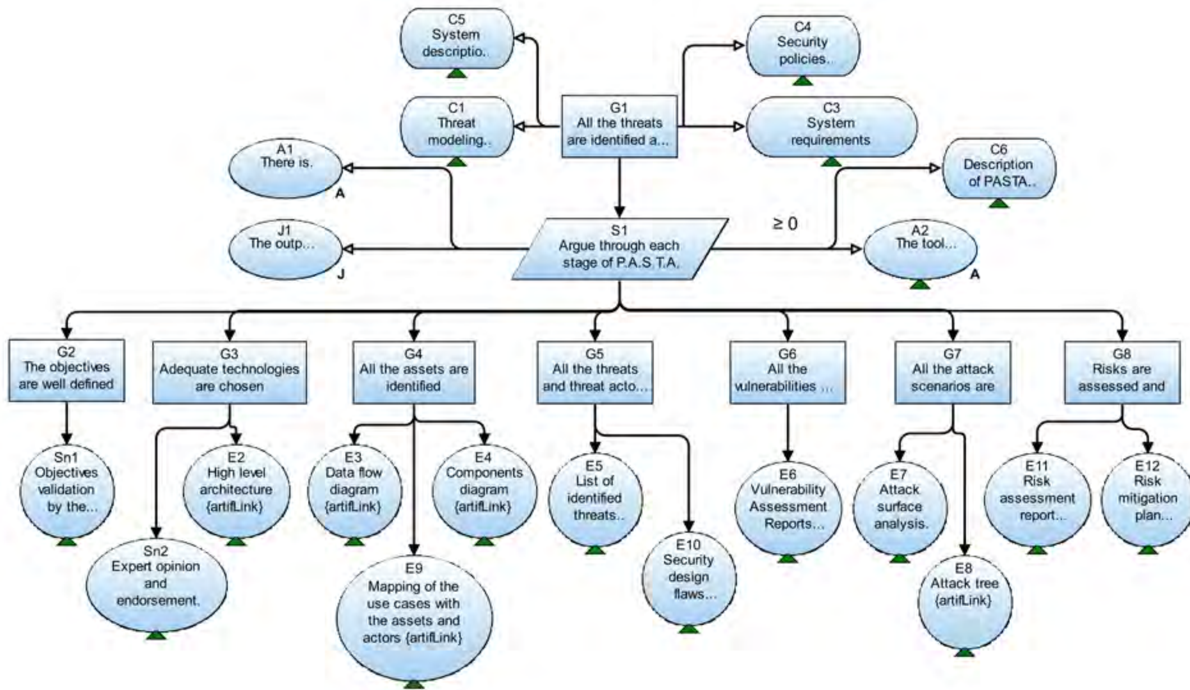
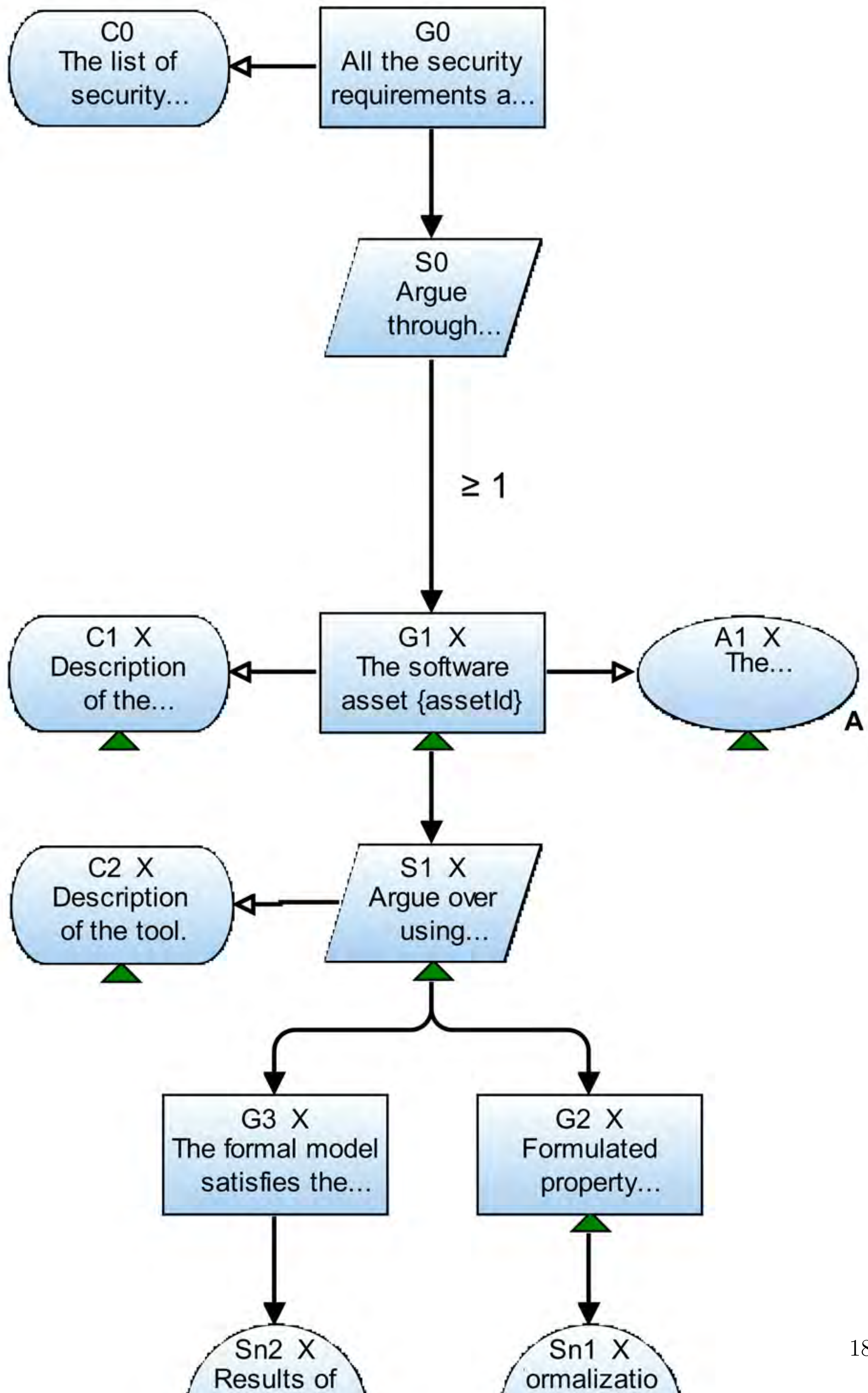


Figure C.3: PASTA threat modeling argument pattern

the types of assets that system will process, store, or transmit, and their criticality to the business or organization.

The blue dashed box in Fig. C.7 helps in deriving security threats from asset protection requirements. We follow the strategy S1, which is to meet potential SRs at the different stages of the system development life cycle. In this work, we address the architecture design stage G3. The sub-claim G4 covers other development phases, which will remain undeveloped in this work. The claim G3 can be fulfilled by the strategy S2 which refers to deriving security threats. Detecting and mitigating the threats help in the fulfilment of the SRs J1. We assume that all relevant threats have been identified A2, and we claim that the system is protected against these threats G5. The strategy S3 divides the claim G5. An instance of the goal G0.X is created for each threat (STX) against which the system must be protected, where X denotes the order of the threat.

The last sub-claim G6 is about the validation of the architecture. It comes after the verification phase while using testing, analysis, or other means to ensure that it actually meets the security requirements and effectively protects the assets. We refer to the formal model of the architecture in node C3. Since the pattern claims that the system architecture model is protected against a threat, it is wise to assume that the system architecture


C0
The list of
security...
G0
All the security
requirements a...
S0
Argue
through...
≥ 1
C1 X
Description
of the...
G1 X
The software
asset {assetId}
A1 X
The...
A
C2 X
Description
of the tool.
S1 X
Argue over
using...
G3 X
The formal model
satisfies the...
G2 X
Formulated
property...
Sn2 X
Results of
Sn1 X
ormalizatio

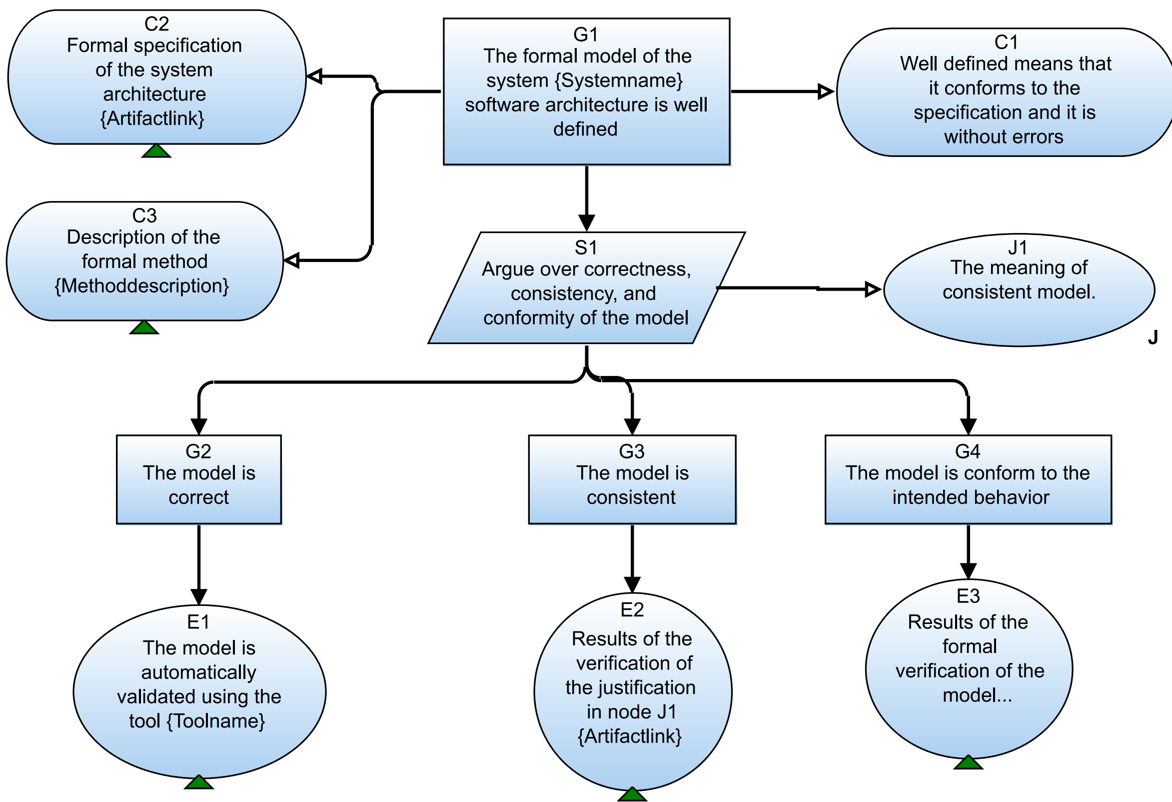


Figure C.5: Model well-formedness argument pattern

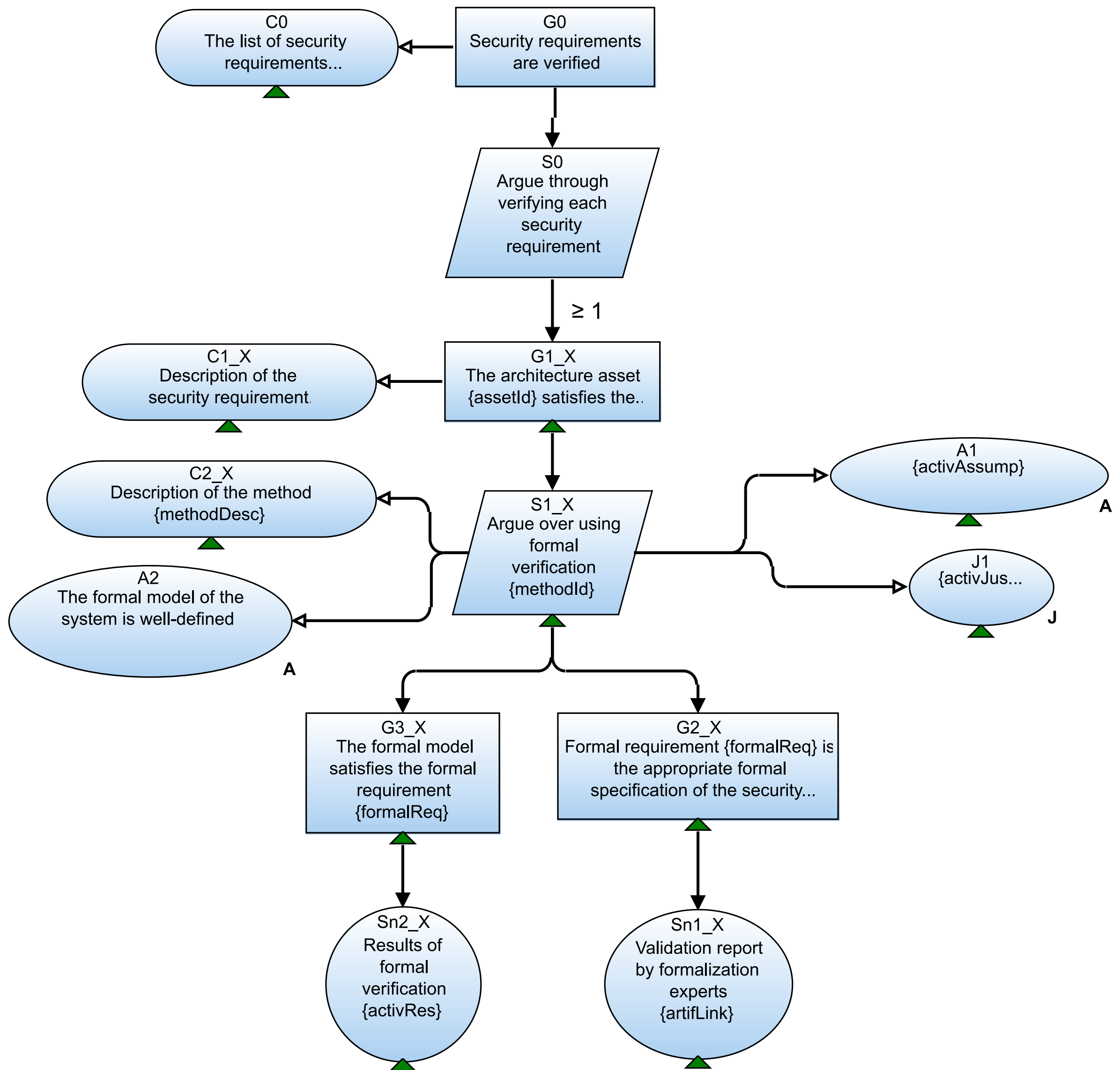


Figure C.6: Formal verification argument pattern

model is well-defined A3.

C.2.2 Threat mitigation argument pattern

The pattern is taken from our previous work [131].
The goal of this pattern is to provide a convincing argument about using formal methods to both formalize and detect security threats. The structure of the pattern is shown in Fig. C.8. The root claim G0.X of the pattern is about the protection against the threat (STX). We refer to the threat in the node C0.X. The proposed strategy S0.X involves arguing through formal specification and verification of the threat. The argument implies that if the sub-claims G1.X, G2.X are satisfied, then the system architecture is protected against the threat, thus fulfilling the corresponding requirement. Sub-claim G1.X claims that the proposed formalization *property* C1.X is the correct formalization of the absence of threat according to restrictions imposed in the formal language C2.X. We rely on domain and formalization experts' inspection Sn0.X as evidence that the formulated expression is the proper specification of the requirement (absence of the threat). Consequently, the violation of the property indicates the presence of the threat. According to claim G2.X, the model satisfies the property {property}. Thus, the threat is absent, and the architecture is protected against it. This claim is supported by the results of formal verification on *property*(Sn1.X).

C.2.3 Alloy Model well-fromedness argument pattern

The pattern is presented in our work [130] The goal of this pattern in Fig. C.9 is to claim the well-definedness of the system model according to Alloy language semantics. A consistent model has at least one instance that resolves all the facts and the declarations. The system model describes assumptions about the world in which a system operates, requirements that the system is to achieve, and a design to meet those requirements. The root claim in G0 resumes the main goal of the pattern in the context C0, C1. According to Alloy language rules C2, a model is inconsistent if it does not have any core instances S0,J0,J1. The goal G1 claims that the model has an instance that resolves constraints formed by the conjunction of the facts C3 and the declarations C4. The analysis results Sn0 form the evidence to support the claim.

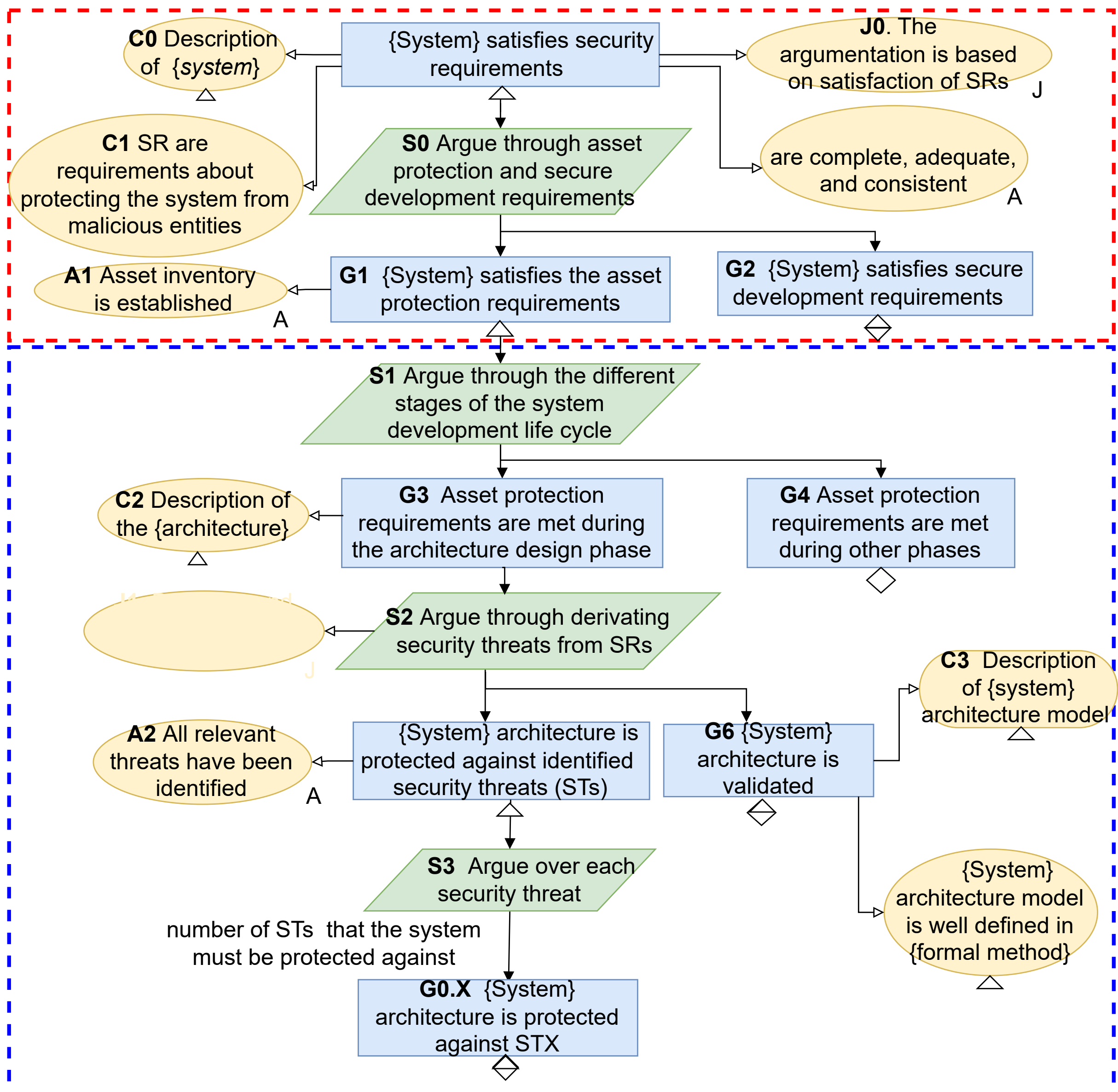


Figure C.7: Argument pattern for threats identification

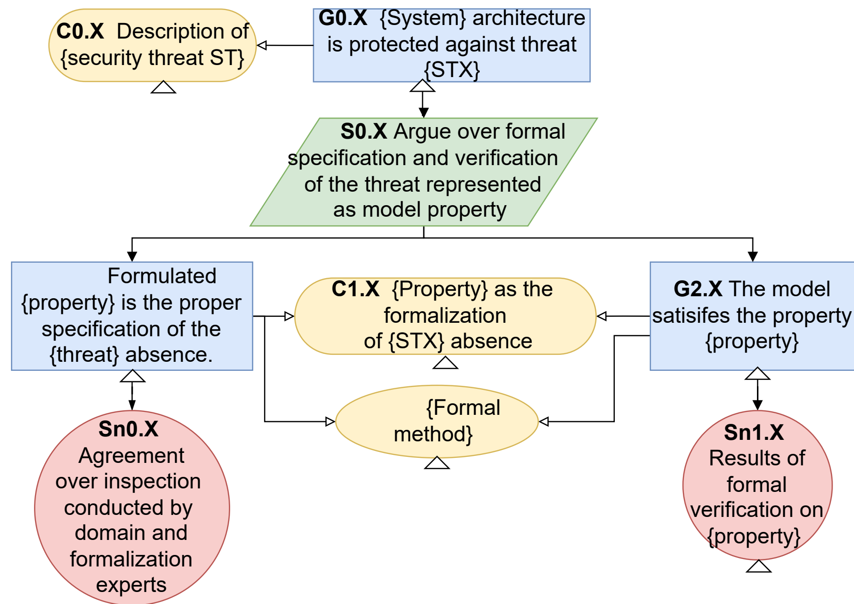


Figure C.8: Argument pattern for the protection against threats

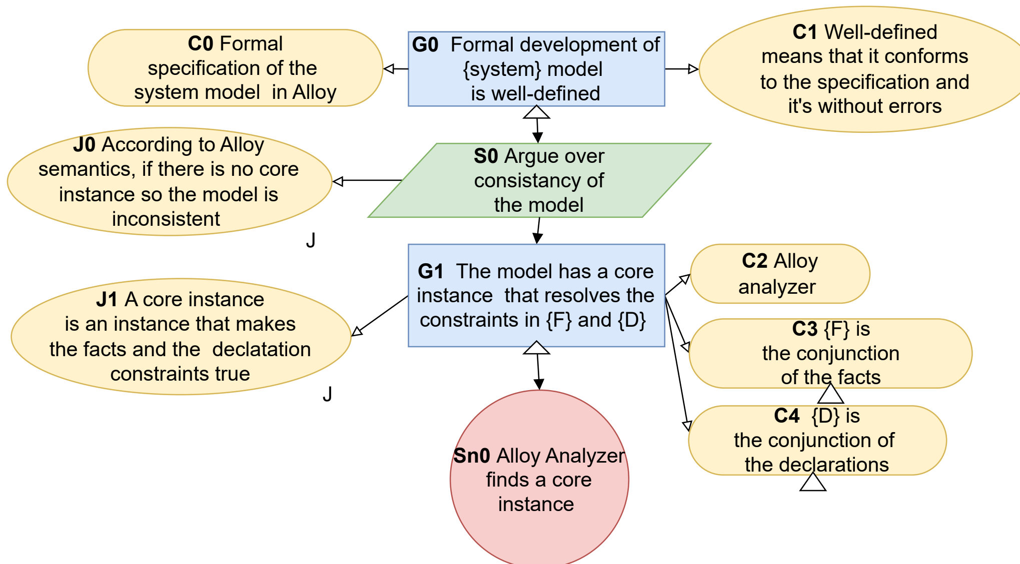


Figure C.9: Pattern for the well-definedness of the system model

C.2.4 DNN secure development argument pattern

In this section, we present a security assurance argument pattern called *DNN Secure development argument pattern*. It is presented in our work entitled [132].

Pattern name DNN Secure development argument pattern

Context You are developing a DNN model that realizes a task and will be deployed in a security-critical system. In addition, to prove that your DNN model performs well, you need to guarantee that as a developer, your DNN satisfies its security requirements.

Intent The pattern provides a claim decomposition to argue that the development of a DNN satisfies its security requirements if the security requirements are satisfied in the different activities of the DNN development process. The goal is to provide a convincing argument that the DNN under development complies with the necessary processes and activities to support claims that the DNN is adequately secure. The aim is to help designers and developers construct such an argument while reducing the effort required to do so.

Motivation Multiple activities are undertaken while developing DNN. Each activity generates some artifacts whose security should be ensured. Consequently, we will have many security assurance fragments. The DNN secure development argument pattern is a preliminary brick that indicates how the DNN-related patterns should be assembled.

Applicability The pattern should be applied when compliance is required in the secure development processes for the DNN under assurance. The pattern assumes that the relevant development methodology provides sufficient context for its instantiation.

Structure The argumentation strategy is captured by the argument pattern shown in Fig. C.7. It's represented using GSN pattern notation.

Participants The pattern participants are as follows:

Goal: G0 The overall objective of the argument; to support the claim that the DNN satisfies its security requirements in the operating environment.

Context: C0 The description of the the DNN for which the security assurance case is being developed.

Context: C1 The description of the system in which the DNN will be deployed.

Assumption: A0 The DNN must satisfy all the elicited security requirements. This elicitation must be made as part of the system requirements engineering process. If an argument to support this assumption exists, then that argument can be linked here in place of the assumption.

Strategy: S0 The strategy requires splitting the goal into two sub-goals regarding the DNN development stages.

Goal: G1 This goal states that the data oriented stage must satisfy security requirements. This claim remains undeveloped in this work.

Goal: G2 This goal states that the model oriented stage satisfies the security requirements.

Context: C2 This context node refers to the list of security requirements.

Strategy: S1 The argumentation strategy involves going through the different model oriented activities and ensuring that that the security requirements are met in each activity.

Context: C3 This context node refers to the description of the DNN model oriented activities.

Goal: G3 The first sub-goal states that the selected DNN architecture is appropriate for the given set of DNN security requirements.

Goal: G4 The second sub-goal states that the DNN training is appropriate for the given set of DNN security requirements.

Goal: G5 The last sub-goal states that the trained DNN satisfies the security requirements.

Strategy: S2 The decomposition strategy argues through verifying each security requirement.

Goal: G6.X This goal states that the DNN satisfies one security requirement. {X} is used to enumerate the security requirements.

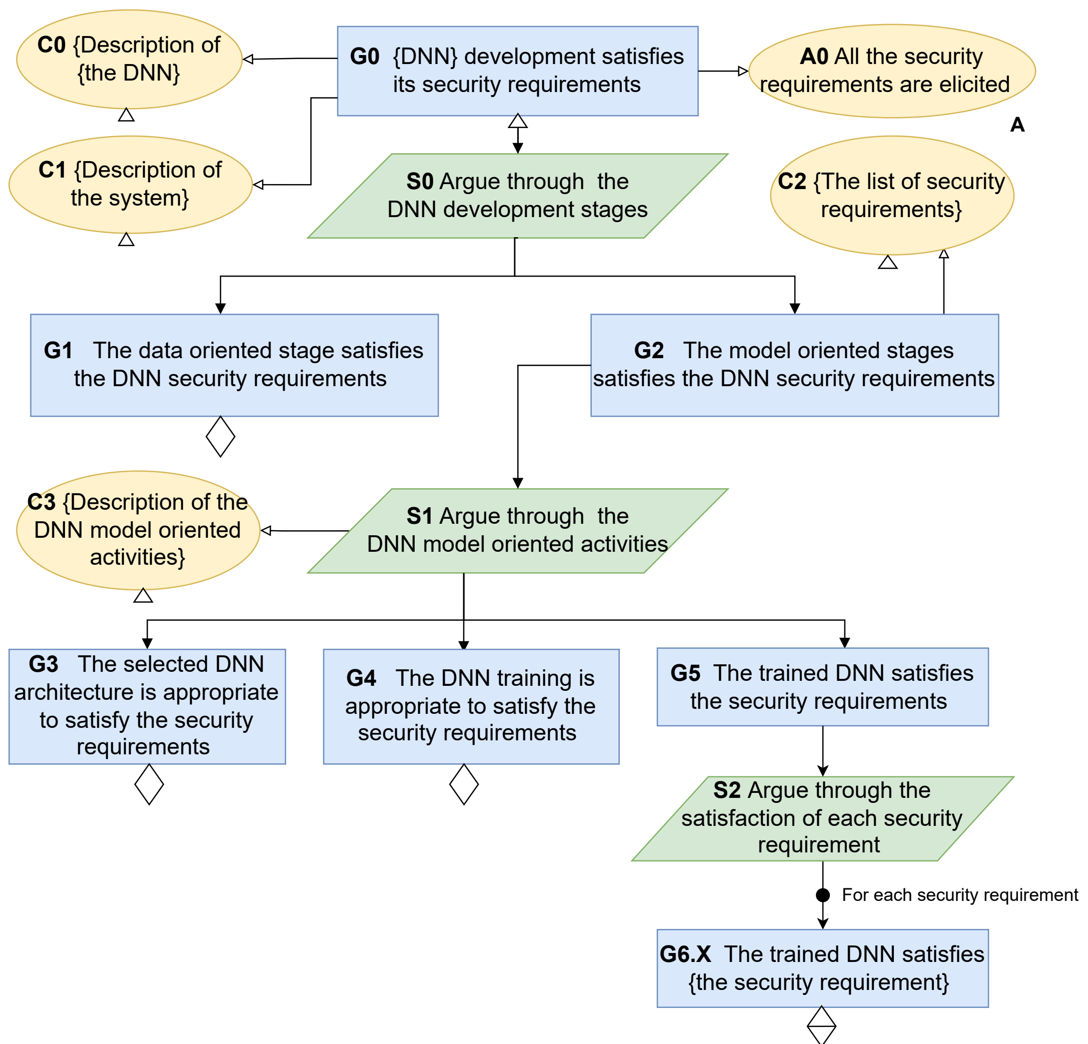


Figure C.10: DNN Secure development argument pattern

Consequences The pattern enables a separation of concerns in the argumentation about the fulfillment of DNN security requirements. Decomposition of the security requirements allows us to focus on security requirements related to the model-oriented stages, and more specifically to each activity in this stage, one at a time. However, after instantiating the pattern, some undeveloped goals will remain. Specifically, each goal of the model-oriented activities will require further decomposition and support. One of the key challenges in constructing assurance arguments is determining the granularity by which each goal is decomposed so that sufficient evidence can be generated to fully support the goal [60]. Because there are many different ways to decompose goals, it is not immediately evident when to stop the decomposition or what sufficient granularity would be to ensure that the required evidence could be provided. In some cases, these goals can be decomposed by adopting other argument patterns recast in the context of security.

C.2.5 DNN security requirements satisfaction argument pattern

This pattern is defined as a development for the leaf node from the previous pattern. It appears on our work [132].

Pattern name Security requirement satisfaction argument pattern

Context You have developed a DNN that realizes a task and you will deploy it in a security-critical system. In addition, to prove that your DNN performs well, you need to evaluate it regarding the security requirements. You can either use test methods or formal verification to prove the fulfillment of the security requirement.

Intent The goal of this pattern is to provide a convincing argument about the verification of the security requirements before deploying the DNN. The pattern indicates what are the evidence elements necessary to support the claims about the satisfaction of security requirements. The pattern shows also that we can use tests and/or formal verification to generate the evidence elements.

Motivation The motivation behind verifying the fulfillment of security requirements specifically for DNN model stems from the potential risks associated with these complex machine learning models. DNN are increasingly utilized in critical applications such as autonomous vehicles, healthcare systems, financial services, and cybersecurity, where the

consequences of security breaches can be severe. Consequently, the assurance of the security requirements fulfillment is of utmost importance.

Applicability The pattern should be applied when compliance is required to verify the fulfillment of security requirements. The pattern assumes that the relevant standard or development methodology provides sufficient context for its instantiation.

Structure The argumentation strategy is captured in the argument pattern shown in Fig. C.8. It's represented using GSN pattern notation.

Participants The pattern participants are as follows:

Goal: G6.X This goal states that the DNN satisfies one security requirement.

Context: C4.X This context node refers to the security requirement to satisfy.

Strategy: S3 The argumentation strategy provides a choice over how the claim can be supported. The choice in the argument should be interpreted as "at-least-1".

Justification: J0 The justification node states that the choice of the verification approach to use should be justified.

Goal: G7.X This goal states that the test data must demonstrate that the DNN security requirement is satisfied.

Context: C6.X This context node refers to the test data: a sufficient range of inputs representing the operating environment, that are not included in the data used in the DNN learning stage.

Solution: Sn0.X This node refers to the evidence from the test results.

Goal: G8.X This goal states that formal verification must demonstrate that the DNN security requirement is satisfied.

Context: C5.X This context node refers to the formal verification method that is used.

Strategy: S4 The argumentation strategy involves arguing through the formal specification and verification of the security requirement.

Goal: G9.X This goal states that formal specification of the security requirement is the appropriate one.

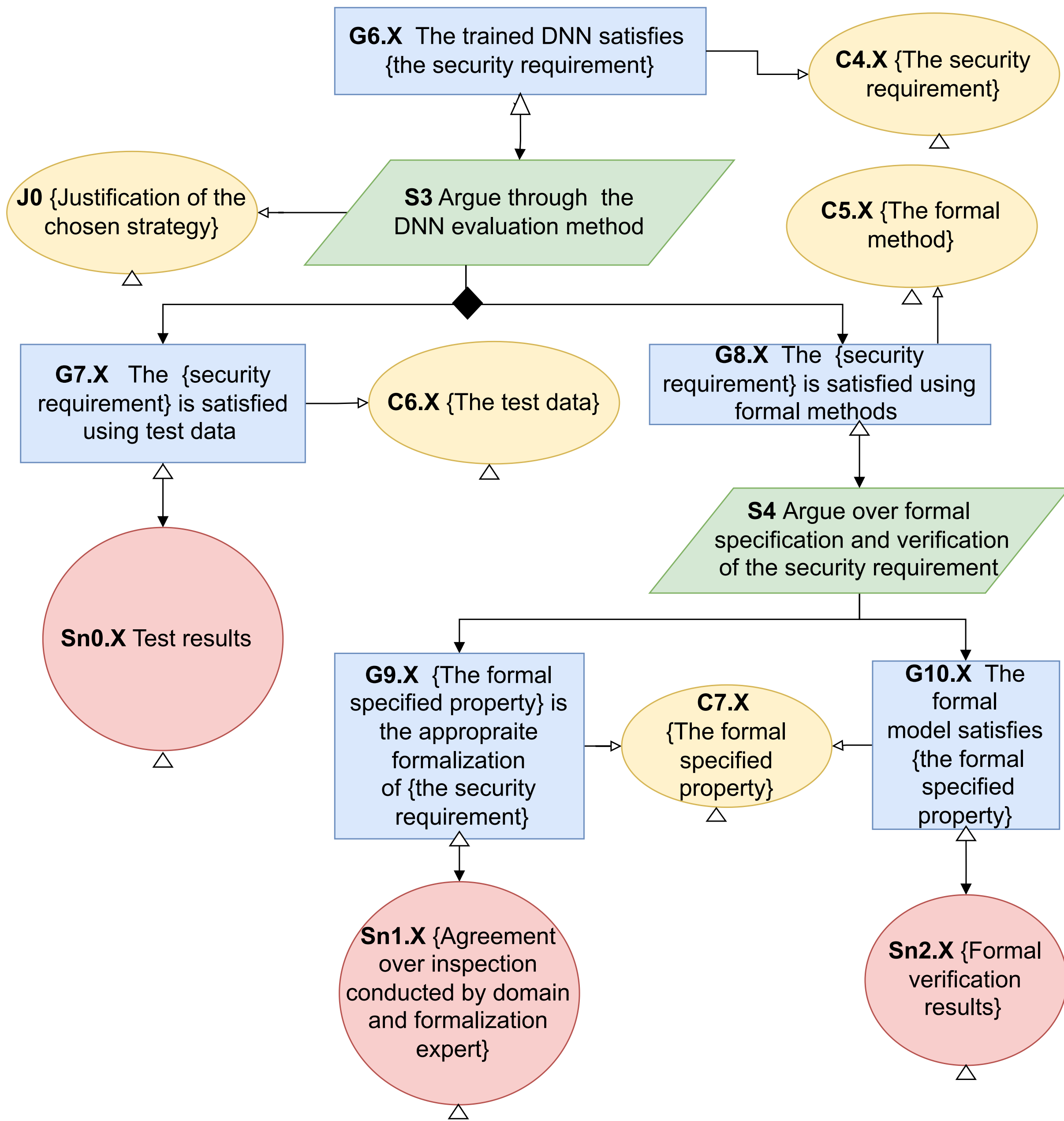


Figure C.11: Security requirement satisfaction argument pattern

Context: C7.X This context node refers to the formal specified property proposed as the formalization of the security requirement.

Solution: Sn1.X The solution node refers to the expertise required to justify the correctness of the security requirement formal specification.

Goal: G10.X This goal states that formal model satisfies the specified property.

Solution: Sn2.X This node refers to the evidence from the formal verification results

Consequences The pattern demonstrates that the DNN model will meet its security requirements when exposed to inputs not present during the development of the model. It also shows the relationship between the verification evidence and the security requirements. However, after instantiating the pattern, some goals will remain undeveloped. Specifically, goals related to DNN testing will require further decomposition and support. The further decomposition can be based on the specific requirements that must be satisfied in support of the instantiated security requirement.

Appendix D

Deep Neural Networks

D.1 Overview of DNN

DNN are a type of artificial neural network. DNN can be used to extract some mapping relations from training data. This relation will later enable matching, as best as possible, new data inputs to the correct output, namely for classification and regression tasks. The processing node in DNN is called a neuron. A neuron processes some received data and forwards it to neurons in the next layer. DNN comprise one input layer, one output layer, and one or more hidden layers. In this work, we will focus only on DNN built through supervised learning and used for the classification of tabular inputs. Tabular inputs comprise a finite discrete set of features (attributes) [129]. *An input neuron* is a neuron of data ingestion; it corresponds to one input feature. *Output neuron* is a neuron of data outlet; each neuron corresponds to one class label. *A bias neuron* is a neuron of data injection; each neuron adds a constant value to neurons in the next layer. The value is called bias and is used with the weights products sum and neuron inputs to produce an output. *A hidden neuron* is a neuron of data processing; each neuron accepts input data, performs calculations, and produces output. *An edge* is a weighted link to connect neurons in successive layers. Weights increase or decrease the impact of the neuron's inputs on its output. *An activation* function transforms the neuron's output before passing it to the next layer.

D.2 DNN LifeCycle

Deep learning as a sub-field of machine learning is also data-driven, i.e., the behavior of a DNN-based system is learned from data. Consequently, DNN require new development

activities such as data-oriented activities [29]. Assuring the use of DNN in critical systems requires a deep understanding of the DNN lifecycle and the artifacts generated during it. Thus, we studied existing machine learning lifecycle stages from several works [10, 100, 91]. We compared them to works focusing on deep learning lifecycle [92, 78]. In our work, we revise the work of [10] and adopt it by defining the DNN lifecycle. Fig. D.1 shows the critical lifecycle stages of DNN. Several activities are associated with each stage.

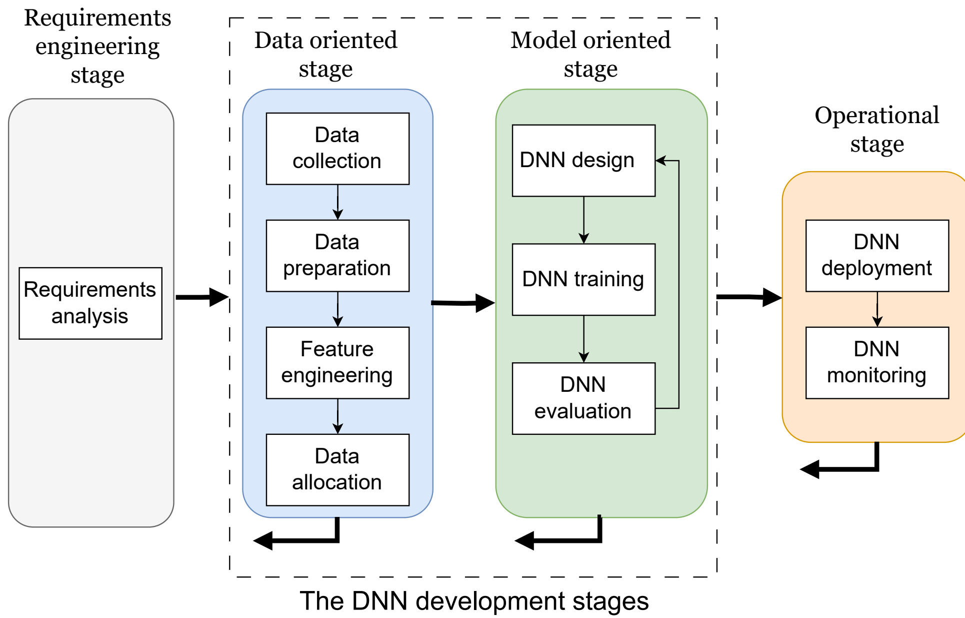


Figure D.1: Deep Neural Network Life cycle revised from [10]

In the rest of this section, we describe each stage in the lifecycle, and then we discuss the development activities associated with it.

D.2.1 Requirements engineering stage

This stage aims to define the DNN requirements. The requirements elicitation technique must allow the identification of all the requirements. A part of the DNN requirements is generally allocated from the system requirements (which functionality and interfaces to realize). The rest of the requirements should constrain the best data and model choice to realize the allocated requirements.

D.2.2 Data-oriented stage

This stage focuses on obtaining the data-sets necessary to develop the DNN. It includes the following activities:

Data collection This activity concerns collecting data from different sources (web scraping, manual data entry, data acquisition from sensors or devices), or re-using preexisting data.

Data preparation The preparation of the collected data involves cleaning the data of any errors and inconsistencies. After data cleaning, the data engineer proceeds to the labeling: assigning relevant tags to data instances.

Feature engineering This activity involves transforming the prepared data into a suitable representation or set of features that DNN can effectively utilize. It involves extracting, selecting, and creating informative features from the available data.

Data allocation This activity splits the data into training, validation, and test data. The training data is used to optimize the model's parameters, and the validation set helps fine-tune the model and select hyperparameters. Moreover, the test data can be used for the DNN evaluation activity in the following stage.

D.2.3 Model oriented process

When the data becomes available, a deep learning developer uses the labeled dataset to learn and optimize the DNN parameters, enabling the DNN to make accurate predictions or perform a specific task.

DNN design The success of DNN depends heavily on design choices, such as finding the most efficient architecture for a given task [79]. Much of the recent work in deep learning has indeed focused on proposing different architectures for different learning tasks(e.g., Recurrent neural network (RNN), Convolutional neural network (CNN), Feedforward neural network (FNN)) [56]. The DNN architecture involves the distribution of neurons through different layers and different connections linking them. It also involves the activation function that will be used.

DNN training The training activity aims to optimize the performance of the DNN model concerning an objective function that reflects the requirements for the model. The training data is used to find internal model parameters (e.g., the weights of a neural network) that minimize an error metric for the given dataset. The validation

data are then used to assess the DNN model's generalization ability. These two steps are typically iterated many times, with the training hyperparameters (parameters initialization and activation function) tuned between iterations to further improve the DNN model's performance.

DNN evaluation This activity is crucial to ensure that the trained DNN model meets the desired criteria and performs effectively on new, unseen data. It helps identify and address any issues, bugs, or inconsistencies early in the development cycle, minimizing potential risks and ensuring the DNN operates as intended before moving to the deployment phase.

D.2.4 DNN operational process

At this stage, the DNN is deployed in the target system and monitored for metrics and errors during execution. This stage will be out of the assurance scope of this work.

D.3 DNN from the case study

The ownership is equipped with ACAS Xu using the DNN shown in Figure D.2. ACAS Xu, uses as inputs, various parameters related to the state of the ownership and the intruder, such as the distance between them (ψ), their velocities, respectively (v_{own}, v_{intr}), the angle to the intruder (θ), and the heading of the intruder (ψ). This information is used to predict a set of recommended actions to avoid a collision. The first option is to do nothing, also known as the "Clean of Conflict"(COC) option. However, we also have the option to make a weak adjustment by turning left (WL) or right (WR). Additionally, we can make a strong adjustment by turning left (SL) or right (SR).

D.4 Security verification of DNN

D.4.1 Test-based verification

it utilizes test cases to demonstrate that the ML component generalizes to cases not present in the ML model learning stage. This shall involve an independent evaluation of the requirements considered during the model learning stage. Specifically, those security requirements associated with ensuring the robustness of models are evaluated on the independent verification data set, i.e., that the performance is maintained in the presence of adverse conditions or signal perturbations. The test team should examine those cases

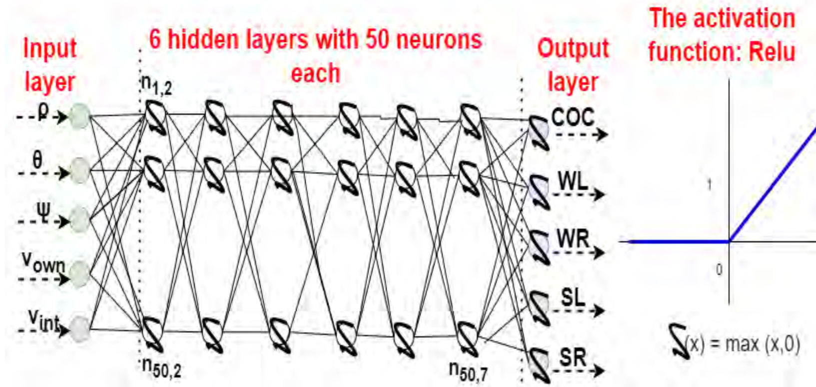


Figure D.2: ACAS Xu neural network

which lie on boundaries or which are known to be problematic within the context to which the DNN model is to be deployed [54]. Beyond using test data, several works developed tools to test the Deep Neural Network (DNN). Work in [90] generates test inputs for a DNN. It identified incorrect corner case behaviors in autonomous systems(e.g., self-driving cars crashing into guard rails and malware masquerading as benign software). Evidence is then the inputs used for testing and the results of testing.

D.4.2 Formal verification

formal verification methods like deductive methods, abstract interpretation, or model checking aim at verifying that an ML model of a system satisfies some properties on a mathematical basis [73]. In the current context, the objective is to demonstrate the compliance of an ML model to its security requirements in all possible situations without explicitly testing the behavior of the ML component in each of them. Authors from [113] studied the state of the art of the methods proposed for verifying properties of neural networks such as Satisfiability Modulo Theory (SMT), Mixed Integer Linear Programming (MILP), and Symbolic intervals. To verify the local robustness of DNN, work from [57] verified that for a given DNN input, there is a region around the input where the DNN output remains unmodified. In another work from [43], an additional binary classifier is trained to distinguish between the adversarial and original samples that can be regarded as the detector model. This method can be used to build robust DNN to adversarial perturbations. The formally-specified properties shall be a sufficient representation of the DNN security requirements in the context of the defined operating environment. An explicit justification shall be documented for the sufficiency of the translation to formal properties [54]. Evidence is then the result of the formal verification of the security

requirements.

D.4.3 Monitoring

Monitoring machine learning components in security-critical systems is essential to ensure they function reliably and safely. These components can be targeted by adversarial attacks, data drift, or unexpected behavior that may compromise security. Continuous monitoring allows for real-time detection of anomalies, misclassifications, or performance degradation, enabling timely responses to potential threats. Effective monitoring includes tracking input data quality, model outputs, and system performance metrics to maintain security and robustness throughout the system's lifecycle.

Glossary

Alloy

Alloy is an open source language and analyzer for software modeling. It has been used in a wide range of applications, from finding holes in security mechanisms to designing telephone switching networks. (taken from `http://alloytools.org/`)

API

Application Programming Interface

CAE

see: Claim Argument Evidence

Claim Argument Evidence

A graphical notation commonly used in assurance cases to visually represent structured arguments that justify the assurance of a system's properties, such as security, safety, or reliability.

CLASP

see: Comprehensive, Lightweight Application Security Process

Comprehensive, Lightweight Application Security Process

The Comprehensive, Lightweight Application Security Process (CLASP) is a process, developed by the OWASP Foundation, providing a well-organized and structured approach for moving security concerns into the early stages of the software development life cycle, whenever possible.

Coq

Coq is a formal proof management system. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an en-

vironment for semi-interactive development of machine-checked proofs. (taken from `https://coq.inria.fr/`)

Domain-Specific Modeling Language (DSML)

A language to create models that are specific to a certain domain.

DSL

Domain Specific Language

DSM

Domain Specific Modeling

DSML

see: Domain-Specific Modeling Language (DSML)

EBNF

Extended Backus-Naur Form

Eclipse Modeling Framework Technology

The Eclipse Modeling Framework Technology (EMFT) project exists to incubate new technologies that extend or complement EMF. *see also:* Eclipse Modeling Framework

Eclipse Modeling Framework

The Eclipse Modeling Framework is an Eclipse-based modeling framework and code generation facility for building tools and other applications based on a structured data model.

Ecore

Ecore is a reference implementation of OMG's EMOF. *see also:* Eclipse Modeling Framework

EMF

see: Eclipse Modeling Framework

EMFT

see: Eclipse Modeling Framework Technology

EMOF

Essential MOF

Formalization

The process of creating formalized structure.

Goal Structure Notation

Graphical argumentation notation that can be used to document explicitly the individual elements of any argument (claims, evidence and contextual information) and, perhaps more significantly, the relationships that exist between these elements.

GSN

see: Goal Structure Notation

IDE

see: Integrated Development Environment

IEEE

Institute of Electrical and Electronics Engineers

Integrated Development Environment

An Integrated Development Environment is software that consolidates the basic tools needed for software testing and writing.

ISO

International Organization for Standardization

MBE

Model-Based Engineering

MDE

see: Model-Driven Engineering (MDE)

Meta-Object Facility

The Meta-Object Facility (MOF) is an Object Management Group standard for model-driven engineering, defining a meta-metamodel.

Microsoft STRIDE

Microsoft STRIDE is security model which categorizes different types of threats and simplifies the overall security conversations. It classifies threats into six categories: *Spoofing*, *Tampering*, *Repudiation*, *Information Disclosure*, *Denial of Service*, and *Elevation of Privilege*.

Microsoft Security Development Life cycle

Microsoft Security Development Life cycle (SDL) is a security-oriented development lifecycle proposed by Microsoft.

MLBS

Machine Learning Based System

Model-Driven Engineering (MDE)

A software development methodology that focuses on creating and exploiting models for abstract representations of the knowledge governing a particular domain.

MOF

see: Meta-Object Facility

NIST

National Institute of Standards and Technology

Object Constraint Language

Object Constraint Language (OCL), a declarative language for describing rules applying to OMG-style metamodels and providing constraint and object queries.

Object Management Group

Object Management Group (OMG), an international, open membership, not-for-profit technology standards consortium.

OCL

see: Object Constraint Language

OMG

see: Object Management Group

Open Web Application Security Project

The Open Web Application Security Project is a nonprofit foundation that works to improve the security of software. Through community-led open-source software projects, hundreds of local chapters worldwide, tens of thousands of members, and leading educational and training conferences, the OWASP Foundation is the source for developers and technologists to secure the web. (taken from https://owasp.org/

OWASP

see: Open Web Application Security Project

PASTA

The Process for Attack Simulation and Threat Analysis is a risk-centric

Proof-Obligation (PO)

A theorem stating that a certain property must hold in order for a formal specification to be internally consistent.

S&D

Security and Dependability

SACM

Structured Assurance Case Metamodel

SCML

Security Case Modeling Language

SDL

see: Microsoft Security Development Life cycle

SDLC

Security Development LifeCycle

SLR

see: Systematic Litterature Review

SR

Security Requirement

Systematic Litterature Review

SLR is a structured and rigorous method for identifying, evaluating, and synthesizing existing research on a specific topic. It is commonly used in scientific and engineering fields to provide a comprehensive overview of the state of the art, identify research gaps, and support evidence-based conclusions.

UML

see: Unified Modeling Language

Unified Modeling Language

Unified Modeling Language (UML) is a general-purpose modeling language in the field of software engineering and is defined by the Object Management Group.

Xtend

Xtend a statically typed programming language that produces understandable Java code. Xtend is based on the Java programming language in terms of syntax and semantics, however it improves on several aspects. This aspects made Xtend well-suited to the task of code generation. It is fully integrated into Xtext.

Xtext

Xtext is a framework for development of programming languages and domain specific languages. It covers all aspects of a complete language infrastructure, from parsers, over linker, compiler or interpreter to fully-blown top-notch Eclipse IDE integration. It comes with great defaults for all these aspects which at the same time can be easily tailored to your individual needs. (taken from `https://eclipse.org/Xtext/`)