RÉPUBLIQUE FRANÇAISE
Liberté Égalité Fraternité

Université de Toulouse

Doctorat de l'Université de Toulouse

délivré en co-tutelle avec Université Carleton

préparé à l'Université Toulouse - Jean Jaurès

Modélisation formelle d'architectures sécurisées dans une perspective positive à l'aide d'Event-B : propriétés, modèles, analyse et outillage

---

Thèse présentée et soutenue, le 2 décembre 2025 par

Loïc THIERRY

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 et Jason JASKOLKA

Composition du jury
M. Jean-Paul BODEVEIX, Président, Université de Toulouse
M. Dominique MERY, Rapporteur, TELECOM Nancy
M. Marc FRAPPIER, Rapporteur, Université de Sherbrooke
Mme Olga KOUCHNARENKO, Examinatrice, Université de Franche-Comté
M. Yvan LABICHE, Examinateur, Carleton University
Mme Felty AMY, Examinatrice, University of Ottawa
M. Brahim HAMID, Directeur de thèse, Université Toulouse - Jean Jaurès
M. Jason JASKOLKA, Co-directeur de thèse, Carleton University

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

l'Université Toulouse - Jean Jaurès et l'Université de Carleton - Canada

---

Présentée et soutenue le *02/12/2025* **par :**

Loïc THIERRY

Model-based design and formal analysis of secure architectures using Event-B

JURY

Jean-Paul Bodeveix	Professeur des Universités	Président du Jury
Dominique MERY	Professeur des Universités	Rapporteur
Marc FRAPPIER	Professeur titulaire	Rapporteur
Brahim HAMID	Professeur des Universités	Co-directeur de Thèse
Jason JASKOLKA	Associate Professor	Co-directeur de Thèse
Amy FELTY	Professor Emeritus	Examinateur
Olga KOUCHNARENKO	Professeur des Universités	Examinateur
Yvan LABICHE	Professor	Examinateur

École doctorale et spécialité :

MITT : Informatique et Télécommunications

Carleton University: Department of Systems and Computer Engineering

Unité de Recherche :

Institut de Recherche en Informatique de Toulouse (UMR 5055)

Cyber Security and Evaluation (CyberSEA) Research Lab de Carleton

Directeur(s) de Thèse :

Brahim HAMID et *Jason JASKOLKA*

Rapporteurs :

Dominique MERY et *Marc FRAPPIER*

Abstract

The development of distributed computing systems and of their usage in domains such as the Internet of Things, Big Data, etc., raises numerous questions on the tools available to model the complexity of such systems. Non-formal modeling methods fail to create a rigorous way to describe and analyze these systems. In this thesis, we propose to find an approach to model and analyze both functional and non-functional aspects of these systems using formal techniques.

The objectives of this work include: (1) exploring the use of proof-based formal methods for the specification and verification of both structural and behavioral properties of component-based systems, (2) describing and validating security properties in reusable libraries, (3) describing and classifying the relationships between security properties, and (4) developing a set of supporting tools. To achieve these objectives, we seek to: (a) describe high-level concepts for system architecture in a component-port-connector fashion as a metamodel; (b) develop an Event-B interpretation of the metamodel adding communication characteristics such as buffering, first-in-first-out (FIFO), and synchronicity; (c) extend this formal model to represent security properties; and (d) create an automated tool suite using the formal model that can be integrated into a systems development lifecycle.

The approach leverages both model-driven engineering (MDE) and formal techniques to create methods and tools for respectively the design and the analysis of complex system architectures. The methods and tools will not only focus on functional requirements, but also on non-functional requirements by creating reusable libraries of security properties that can be used during the design phase. To validate our work, we explore a set of representative security objectives extracted from the Confidentiality, Integrity, Availability, Authenticity, Accountability (CIAAA) classification and a case study on a secure Gateway system.

Remerciements

Une étape importante de mon parcours s'achève aujourd'hui. Je souhaite donc, en premier lieu, remercier sincèrement toutes les personnes qui ont rendu possible, ou contribué d'une manière ou d'une autre, à l'aboutissement de ce manuscrit.

Je tiens tout particulièrement à adresser mes plus chaleureux remerciements à mes directeurs de thèse, Brahim Hamid et Jason Jaskolka, sans lesquels ce travail n'aurait pu voir le jour. Leur disponibilité, leur soutien constant et leurs précieux conseils tout au long de ces années ont été déterminants pour la réalisation de cette thèse.

Je remercie également Jean-Paul Bodeveix pour les nombreux échanges, retours et discussions, qui se sont révélés essentiels et ont grandement contribué à la qualité et à l'achèvement de ce manuscrit. J'adresse aussi mes remerciements à l'ensemble des personnes que j'ai eu le plaisir de côtoyer à l'IRIT — chercheurs, enseignants, personnels administratifs et techniques — pour l'environnement de travail stimulant et agréable qu'ils ont su créer.

Je souhaite ensuite remercier Marc Frappier et Dominique Méry pour l'intérêt qu'ils ont porté à mon travail en acceptant d'en être les rapporteurs. Leurs remarques, questions et perspectives ont été particulièrement enrichissantes. Je remercie également Jean-Paul Bodeveix pour avoir accepté de présider le jury de ma soutenance.

Je tiens enfin à remercier l'ensemble des membres du jury — Jean-Paul Bodeveix, Amy Feltier, Olga Kouchnarenko, Yvan Labiche, Jason Jaskolka, Brahim Hamid, Marc Frappier et Dominique Méry — pour leurs questions pertinentes et leurs commentaires stimulants lors de la soutenance.

Pour conclure, je remercie toutes les personnes extérieures au cadre académique qui m'ont soutenu durant ces années. Je pense en particulier à ma famille, pour son soutien indéfectible, ainsi qu'à mes amis, dont la présence et la bonne humeur ont été une source quotidienne d'énergie et de motivation.

Acknowledgements

An important stage of my academic journey comes to an end today. I would therefore like, first and foremost, to sincerely thank all those who made possible, or in one way or another contributed to, the completion of this manuscript.

I would like to express my deepest gratitude to my PhD supervisors, Brahim Hamid and Jason Jaskolka, without whom this work would not have been possible. Their constant support, availability, and invaluable advice throughout these years were essential to the successful completion of this thesis.

I also wish to thank Jean-Paul Bodeveix for the numerous discussions, feedback, and exchanges, which proved to be crucial and greatly contributed to the quality and completion of this manuscript. I further extend my thanks to all the people I had the pleasure of working with at IRIT—researchers, faculty members, administrative staff, and technical personnel—for providing such a stimulating and pleasant working environment.

I would then like to thank Marc Frappier and Dominique Méry for the interest they showed in my work by agreeing to serve as reviewers of this thesis. I am grateful to them for their insightful remarks, questions, and perspectives. I also thank Jean-Paul Bodeveix for accepting to chair the examination committee.

I would like to extend my sincere thanks to all members of the jury—Jean-Paul Bodeveix, Amy Feltier, Olga Kouchnarenko, Yvan Labiche, Jason Jaskolka, Brahim Hamid, Marc Frappier, and Dominique Méry—for their thoughtful questions and stimulating comments during the defense.

Finally, I would like to thank all those outside the academic sphere who supported me throughout these years. In particular, I am deeply grateful to my family for their unwavering support, and to my friends, whose presence and good spirits were a constant source of motivation and encouragement.

Table of contents

List of figures

List of tables

List of listings

Chapter 1

Introduction

Contents

In this chapter, we present the overall context in which this thesis is situated. The objective is to highlight the motivation behind this work, define the research problem it seeks to address, and clearly articulate the corresponding research objectives. Furthermore, this chapter provides an overview of the publications made throughout the course of this work. Finally, we conclude with an outline of the structure of the manuscript.

1.1 Context

Recently, we have seen a significant shift in the development of software applications. We have transitioned from traditional standalone computer systems towards distributed systems, where multiple interconnected devices operate via the Internet. Consequently, Machine-to-Machine communication and other distributed computing paradigms have become central issues in modern systems engineering. This shift has introduced new risks specific to distributed systems while the existing concerns tied to enabling technologies, such as virtualization, containerization, and cloud infrastructures, persist. These challenges demand robust development methodologies that address both functional and non-functional properties, such as reliability, scalability, and security.

To meet these demands, new production methods have emerged that integrate and

extend existing engineering paradigms, including Model-Driven Engineering (MDE) [6, 7], Component-based Software Engineering (CBSE) [8], and formal methods [9]. CBSE emphasizes the use of reusable software components to accelerate development, reduce costs, and improve system quality. MDE focuses on creating and exploiting domain-specific models as primary artifacts throughout the engineering process. It aims to increase automation, consistency, and abstraction by transforming models into executable code or other system artifacts. Meanwhile, formal methods provide mathematically rigorous techniques to specify, model, and verify systems, particularly in ensuring correctness and detecting errors early in the design process. These paradigms, when combined, offer a powerful foundation for designing secure, reliable, and efficient systems. This is particularly relevant for industrial systems with diverse use cases, as well as for web applications driven by the recent proliferation of cloud-based computing systems. Security plays a critical role in these environments, where vulnerabilities can lead to severe consequences.

When studying distributed systems, we often rely on models to abstractly represent their behavior and structure. Distributed computing programs typically employ well-established communication models [10] such as message passing, remote procedure calls, and distributed shared memory. These communication models enable system components—each with a limited, localized view—to interact explicitly with their neighbors. A system's program at each node consists of state variables and a finite set of actions. Components interact with their own state variables and those of their neighbors following specified communication protocols.

In this work, we model software architectures primarily using message passing and remote procedure calls. Our models capture the architectural communication styles and allow us to analyze their behavior rigorously. The primary goal of this modeling and verification process is to ensure that architecture models satisfy desired properties, such as security and correctness, while avoiding undesired properties, such as deadlock or starvation.

Security is often addressed during the operational phase of a system, with risk assessments performed by analysts to evaluate the deployed system's vulnerabilities. This involves collaboration between security experts and system stakeholders to identify risks, recommend security strategies, and implement mitigation measures. However, this reactive approach has limitations:

1. Developers are often constrained by functional requirements, making significant architectural changes impractical due to cost or complexity.

2. Architects and developers typically lack advanced knowledge in security engineering, limiting their ability to implement effective security strategies.

To address these challenges, security must be integrated into the system design process from the early stages. Our work focuses on the architecture development phase, where design decisions remain flexible. We leverage MDE [11] to incorporate formality into system design, enabling rigorous analysis of both functional and security-related properties.

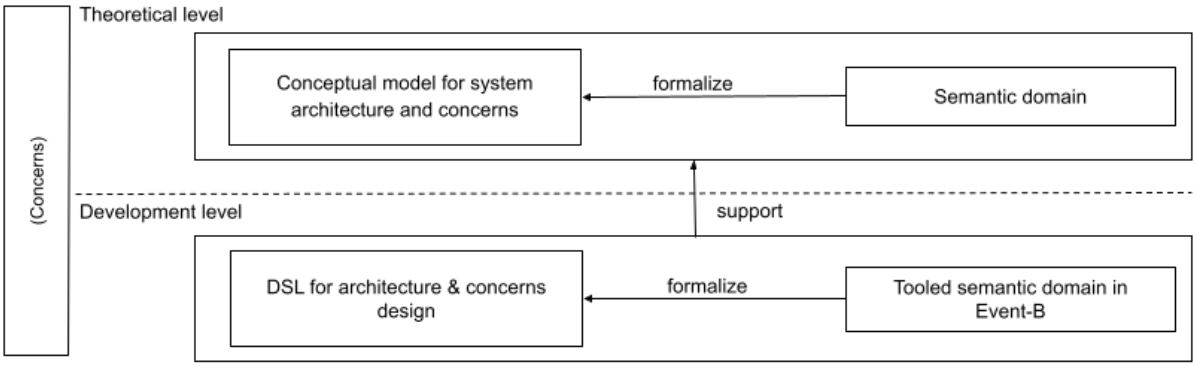

Figure 1.1: Conceptual vision

Moreover, reusable models, such as security policies and design patterns, can bridge the knowledge gap for developers and architects. By embedding security considerations into reusable components and formally analyzing their interactions, we can promote best practices and ensure robust, secure, and adaptable system designs.

1.1.1 Conceptual Vision

This research is part of our research team effort to promote the use of model and pattern-based engineering approaches to improve the design, implementation, configuration and deployment of multi-concerns systems. In the context of our work, we use a similar definition of a *concern* about an architecture as in the work by Rozanski and Woods [12]: a concern is a requirement, a constraint, or an objective that a stakeholder has for that architecture. Ultimately, the goal is to improve the Pattern-based system engineering (PBSE) framework [13] considering security and safety requirements within software architectures built on top of the reusable models described in this proposal. Capturing and providing expertise by the way of security patterns has become an active area of research in the last several years. Security can be captured within patterns that provide reusable generic solutions for recurring security problems, here dealing with architectural problems. A complete catalog of security patterns has been introduced by Fernandez [14]. We plan to transform our Pattern-based system engineering (PBSE) pattern modeling concepts to formal specifications to ensure semantic validation.

As shown in Figure 1.1, the conceptual vision is composed of two levels:

- **Theoretical level**: Creation of a conceptual model of a system and its concerns, as well as the semantics. A conceptual model of a system and its concerns provides a common understanding of all concepts employed in the development level, while the semantic domain ensures a precise description of the addressed problems and solutions approaches. A conceptual model of a system and concerns should capture the main concepts and relationships to describe the concerns within the system in the context of different standards and domain-specific practices. A semantic domain should capture the concerns as desired properties of the system. At this level,

the conceptual model and the semantic domain are described using a technology-independent formalism.

- **Development level**: Creation of a tooled methodology for the system architect to support the design and analysis activities during the processes of building software architecture and concerns from the conceptual modeling and the semantic domain. At this level, we propose using a well-known approach in MDE: a Domain-Specific Language (DSL) for the creation of the design environment and existing tooled formal languages for the analysis environment.

1.2 Motivation

The rise of concerns over security and the ability to build trustworthy systems comes with the question of the tools used to design such systems. Every type of system needs to be able to be represented with the specifics it has and also we must understand the limits of the tools we use.

For Component-Based Systems (CBS) and even security in general, the most prevalent tools used are model checkers. Model checkers are most suitable when trying to find quickly a flaw in modelled system. However, model checkers may lack the definitive answer as they only try to find contradictions in the model. Model checkers can only say they have not found a flaw, not that a flaw does not exist. For example, the work of Rouland et al. [15, 16] shows a modelisation in the model checker Alloy.

To go beyond the limitations of model checking, we want to work on another type of formal method: proof-based languages. By using mathematical proof as a means to prove system properties, we can have a higher level of trust and assurance in the systems we design.

Our motivation for this research stems from key challenges in the development of secure and trustworthy component-based systems:

1. There is a pressing need for a modeling approach that can capture the structure and behavior of component-based systems within a rigorous, proof-based formalism, enabling early validation and reducing design errors.

2. To address growing concerns about system security, we must be able to formally express and verify meaningful security properties within these models, thereby increasing the confidence in the system's resilience against threats.

3. With multiple formal verification techniques available, it is essential to understand the unique capabilities and limitations of proof-based methods compared to model checking, so we can apply the most effective tools for each class of system properties.

1.3 Problem Statement

While the need for considering security early has been acknowledged, traditionally, security has been treated primarily as a concern during the development and deployment phases of the system life cycle, typically approached as a reactive process centered on the detection and mitigation of vulnerabilities. In formal methods, this has often been addressed using model checking techniques. However, model checkers generally provide limited guarantees, identifying counterexamples rather than offering definitive proof of security, and tend to be effective only on systems of relatively small size. This late-stage, limited approach neglects security issues that may arise during the early phases of system design and hinders the concurrent consideration of security and functional requirements. These limitations prompt the question of whether it is possible to develop tools that support the analysis of non-functional security properties using a proof-based approach. Crucially, such tools must maintain a separation of concerns, given the distinct expertise required for formal verification and security analysis.

1.4 Research Objective

Taking into account the previous discussion, we specify our research problem as an overall research goal of this thesis: **Propose an integrated approach for the specification, detection, and implementation of security requirements to support secure systems engineering at early stages of development using reusable models with tool support.** Special emphasis is devoted to promote the particularly challenging task of efficiently integrating established security concepts. Furthermore, a focus is placed on the potential benefits of the combination of MDE with reuse of existing artifact (e.g., security solution models, like security patterns). Decomposing the overall research goal, we propose the following research objectives (RO) that we address in this thesis.

RO 1: Model CBS using formal methods

Our first objective is to create a model capable of representing complex CBS using formal methods. This objective is decomposed into sub-objectives listed below.

RO 1.1: Model an abstract specification to describe CBS and the communication between components

This objective requires a specification that is abstract and independent from any implementation. By creating this abstract specification, we will be able to describe the properties and behavior of a system and then interpret those into any suitable formal language.

RO 1.2: Create a formal model using a proof-based language to represent CBS

We need to model the abstract specifications of CBS using a formal language. This will be done by creating a model in proof-based formal language (Event-B). This model will need to be able to represent both structural and behavioral properties of the systems. We must determine how to create this model in the most generic way possible to be usable for a variety of project while still being able to represent useful properties. When this objective is done, we should have a formal model representing CBS where an architect that is not an expert in formal methods can describe a component-based system.

RO 2: Formalize CBS security properties and constraints formally

Once **RO 1** is completed, we want to use the resulting model to formalize security properties and constraints in a reusable manner. This is done in three steps; first, formalize the properties in an abstract way, then refine the model to implement the properties, and finally bundle them as libraries. These three steps are detailed below.

RO 2.1: Formalize the security properties and constraints using abstract specification

Using the abstract specification language defined in **RO 1.1**, we want to describe some security properties in a model-independent language. This is done by adapting commonly accepted security properties described informally and translating them in this new language adapted to CBS.

RO 2.2: Formalize security properties and constraints using a proof-based language

Based on the model obtained by completing **RO 1.1**, we want to be able to integrate security properties and constraints. This will raise some questions on how to extend the security property model such that the properties created in the previous model are kept in the extended model. Also, work must be done on how we interpret a security property in a formal language. At this point, we should have a set of formally described security objectives as properties in our model.

RO 2.3: Bundle the security properties and constraints in reusable libraries

Once the properties are formalized we need to identify categories to regroup them. These categories can follow recognized categorizations for security objectives like the CIAAA group (Confidentiality, Integrity, Availability, Authenticity, Accountability). These categories will then be implemented into the model as independent libraries of properties that the user can import to suit their needs.

RO 3: Modelization and formalization of relationships between security properties and constraints

RO 2 focused on the specification of individual security properties and constraints. In this objective, we aim to formalize some relationships between these properties. This encompass not only finding which properties and constraints are related, but also what type of relation they have (realizes, enforces, fulfill, etc.).

RO 3.1: Find a model representation of the relationships

In this objective, we aim to extend the concept of security properties and constraints and create a metamodel representing the relationship between security concerns for both domain-specific and domain-independent applications. We aim to create a set of concepts and vocabulary that is general enough to be applicable in most cases.

RO 3.2: Instantiate some relationship using this model

This objective aims to validate the metamodel described in **RO 3.1** by demonstrating its ability to accurately and comprehensively represent relevant security concepts and their interrelationships. Specifically, we seek to identify and construct concrete instances of the relationships defined within the metamodel, drawn from realistic system scenarios or case studies. Through this instantiation process, we aim to assess the expressiveness, consistency, and applicability of the metamodel to support the specification, reasoning, and reuse of security properties and constraints in a model-driven engineering context.

RO 3.3: Formally represent the relationship between security properties

In this objective, we aim to develop a general formal representation of the relationships defined in **RO3.1**. To validate these formal representations, we will leverage the concrete examples and relationship instances identified in **RO3.2**, ensuring that the formalism is expressive enough to capture real-world use cases and sufficiently robust for potential automation and reasoning within model-driven engineering environments.

RO 4: Create a tool suite to support the architects in the process

Building upon the models defined in the previous objectives, this goal aims to make the formal approach accessible and reusable, even for users without expertise in formal methods. We seek to abstract the underlying complexity and provide a user-friendly framework that enables system architects to intuitively describe and verify their systems. At the same time, the framework remains extensible, allowing experts in security and formal verification to contribute by enriching the underlying models and libraries.

RO 4.1: Creation of the domain-specific language using eclipse modeling framework

This objective aims to develop a DSL for representing a CBS and its associated properties. The tool must support the creation of instances of the metamodel, enabling the modeling of concrete systems. The DSL must be formally specified through both an abstract syntax, which defines the structural aspects of the language, and a concrete syntax, which provides its representational form in EMF. This dual specification ensures a complete and precise definition of the domain-specific language.

RO 4.2: Transformation of the DSL into Event-B

This objective aims to enable the automated transformation of the system model into a formal Event-B specification for verification purposes. This transformation builds upon the models defined in **RO1**, **RO2**, and **RO3**, and produces a new Event-B refinement that represents a concrete formal instance of a system modeled using the proposed DSL.

RO 4.3: Verification of the generated model using Rodin

This objective aims to utilize the generated model produced during **RO4.2** and verify it using the Event-B method through the Rodin platform. It encompasses both the structural validation of the system and the formal verification of its functional and non-functional properties. The structural validation ensures that the architectural elements of the system—such as components, connectors, and interfaces—are correctly specified and conform to the defined metamodel. Meanwhile, the formal verification process focuses on discharging proof obligations generated by the Rodin platform to confirm the correctness of the specified properties. This dual-level verification reinforces the soundness and trustworthiness of the modeled system within a rigorous formal framework.

RO 5: Apply and validate the framework in a case study

Once the previous objectives have been realized we should get a complete formal model of component based system architecture. In this model we would have represented some security properties and their relationships. Finally we should have a tool suite supporting the use of the formal model for systems architects. This objective aim to validate all of this work by applying all of those tools and theories into a recognized problem in computer security applied to component-based system.

1.5 Publications

This section presents published papers related to the thesis. In the following, we list and provide concise overviews of those publications in chronological order.

- **Paper A.** [17] **Loïc Thierry**, Jason Jaskolka, Brahim Hamid, and Jean-Paul Bodeveix. "Specification and Verification of Communication Paradigms for CBSE in Event B". In: International Conference on Engineering of Complex Computer Systems (ICECCS). IEEE. 2023, pp. 157–166.
  doi: https://doi.org/10.1109/ICECCS59891.2023.00028.
  Summary: In this paper, we proposed a formal approach to specify and verify both the structural and behavioral aspects of component- based systems using a framework based on the Event-B formal method. This framework is based on a component-port- connector metamodel describing the high-level elements of a distributed system. The behavioral aspect includes a library of paradigms that can be used by the connectors for components to communicate, as well as connector properties that describe the expected behavior (FIFO, Synchronous). We used a simple library example to validate our approach and were able to specify and verify both structural and behavioral properties on its abstract representation.

- **Paper B.** [18] **Loïc Thierry**, Brahim Hamid, and Jason Jaskolka. "A Formal Approach for Verifying and Validating Security Objectives in Software Architecture". In: Verification and Evaluation of Computer and Communication Systems (VECoS). Springer. 2024, pp. 143–158.
  doi: https://doi.org/10.1007/978-3-031-85356-2_10.
  Summary: In this paper, we proposed a formal approach to specify and verify security objectives in the context of a component-based software architecture and message-passing communication, using Event-B. The proposed approach follows two levels of abstraction: (1) logical specification of security objectives using first-order and modal logic as an abstract and technology-independent formalism; (2) formalization and verification using Event-B as a semi-automated proof assistant. For each representative CIAAA security objective, an appropriate security mechanism, acting as a constraint of the model, to make the security property hold is defined and verified.

- **Paper C.** [19] **Loïc Thierry**, Brahim Hamid, and Jason Jaskolka. "Reusable Formal Model Libraries for Specifying and Analyzing Security Objectives in Event-B". In: International Conference on Model and Data Engineering (MEDI). Springer. 2024, pp. 55–63.
  doi: https://doi.org/10.1007/978-3-031-87719-3_4.
  Summary: In this work, we proposed an approach for specifying and and verifying security objectives in component-based software architecture models via reusable formal model libraries of security properties and constraints. Our solution is based on metamodeling techniques that enable the specification of the software architecture structure and on Event-B for the precise specification and verification of security aspects as properties of a modeled system. Furthermore, we walked through an MDE-based prototype to support the proposed approach and a formal-based tool, namely Rodin, to conduct verification and identify model inconsistencies. We constructed an example of this tool suite, in the form of Eclipse plug-ins, using EMF

(Eclipse Modeling Framework).

1.6 Thesis outline

This manuscript is organized as follows. Chapter 2 presents the technical framework that forms the foundation of this thesis. Chapter 3 provides a survey of the literature relevant to the various aspects addressed in this work. Chapter 4 introduces the methodology adopted to design a comprehensive framework based on formal methods and its integration into existing SDLC processes. Chapter 5 presents a model-based definition for the design of CBS of all the key concepts employed throughout this thesis. Chapter 6 provides a formal representation of the previously introduced concepts using abstract semantic and the Event-B formal method for the formal analysis. Chapter 7 describes the tool support developed to implement and apply the proposed approach. Chapter 8 illustrates the proposed approach through a detailed case study and discusses its potential for generalization and extension. Finally, Chapter 9 concludes the manuscript and outlines possible directions for future work.

Chapter 2

Technical framework

Contents

2.1 Introduction

In this chapter, we present the technical framework that supports the development and implementation of this work. We begin by introducing the toolset used for MDE, detailing its capabilities and relevance to our methodology. Subsequently, we provide an overview of the Event-B formal language, including its theoretical foundations and practical utility for formal modeling and verification. This section also describes the associated tooling environment that enables the application of Event-B within our framework.

2.2 Software Engineering

Shortly after the beginning of software development, the way software is developed has been analyzed and guidelines and best practices have evolved over time in the different application domains. Since then, software systems engineering has evolved to an engineering

domain, having impacts on the different fields of system development, such as development processes and software product life cycles. The IEEE defines in their standard ISO/IEC/IEEE 24765:2017 [20] Software Engineering as follows: *software engineering the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.* In software systems engineering, two principal concepts intervene in the construction of a system: a) the practical use and economic value, being the balance of what the customer wants and what the customer is ready to pay for, and b) the correctness, suitability and safety, being the attempt to ensure the correctness and the suitability of resulting product and the absence of safety-critical failures. Tackling these two principal concepts are the challenges of the software engineering discipline, which is based on, and evolves through, application of scientific and mathematical knowledge.

2.3 Component-based Development

Component models provide a way to cope with some limitations of the object model. While object models focus on the encapsulation of data and behavior within objects and emphasize inheritance and polymorphism [21], component models aim to promote higher levels of modularity by encapsulating functionality into reusable, replaceable, and composable units called components [22]. Components typically interact via well-defined interfaces and are deployed independently, allowing for better separation of concerns, scalability, and reusability in large systems. In contrast to object-oriented designs, which often suffer from tight coupling and limited reuse at a larger scale, component-based approaches offer improved maintainability and clearer architectural structure.

Component models complement the object model by providing an architectural view of the application. Therefore, component models provide a coarser-grained representation of the application: a component is typically implemented as a compound of objects or of other components, therefore providing different levels of abstraction for representing complex systems. The main additions of component models to object models can be summarized as follows:

- Identification of connection points between components and the associated links,
- Identification of the services required by a client (instead of just the services provided by a server)
- New communication patterns (for example event-based communication).

Component technology has become a central focus of software engineering in research and development due to its great success in the market. Reuse is a key factor that contributes to this success. The basic idea in CBSE is building systems from existing components rather than “reinventing the wheel” each time. The components are built to be reused in different systems and the component development process is separated from the system development process [23].

Szyperski gives the following definition of a component [22]: *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.* There are several component-based methodologies including Catalysis [24], KobrA [25], OPEN process framework [26]. Flex-eWare [27] is model-driven solution for designing and implementing embedded distributed systems. It combines Model-Driven Engineering and Component-Based Engineering. Bagnato et al. present a framework called EAST-AADL [1] which is an architecture description language for the automotive domain supported with a methodology compliant with the ISO 26262 standard. The language and the methodology set the stage for a high-level of automation and integration of advanced analyses and optimization capabilities to effectively improve development processes of modern cars.

Component-Based Development (CBD) processes are flexibleand can be adapted to many different software development processes (e.g., V cycle, Waterfall, Agile, etc.). In our context, the main concern is to achieve architecture design with components and foster their reuse [28]. To explain CBD processes, reference development phases corresponding to four abstraction levels of a system model are identified as shown in Figure 2.1: Requirements, System Architecture Design, Software Architecture Design and Component Internal Design.

In addition, Figure 2.1 shows the different artifacts at each phase (features, analysis functions, design functions and software components) and their m-to-n relationships [1]. Typically, "m-to-n" relationships (from n entities of a higher level to m entities of the lower level) allow refining models throughout the process for an incremental system concretization.

Requirements. During this phase, requirements are analyzed in order to construct a set of features that the system is expected to provide.

System Architecture Design. In this phase, each feature is analyzed and refined with a set of individual function units (e.g., sensing and actuating functions). Hence, the functions are abstract and independent from any hardware or software.

Software Architecture Design. Each function is refined with a set of software components (which define software components in terms of provided and required interfaces) and hardware nodes.

Component Internal Design. In this phase, each software component is realized. The realization may be done by: (1) combining existing software components or (2) from scratch. Particularly, in Component-Based Development (CBD) methods using UML such as Catalysis [24], each component is designed and implemented internally as a set of classes that implement the interfaces required externally. In addition, internal interactions are realized.

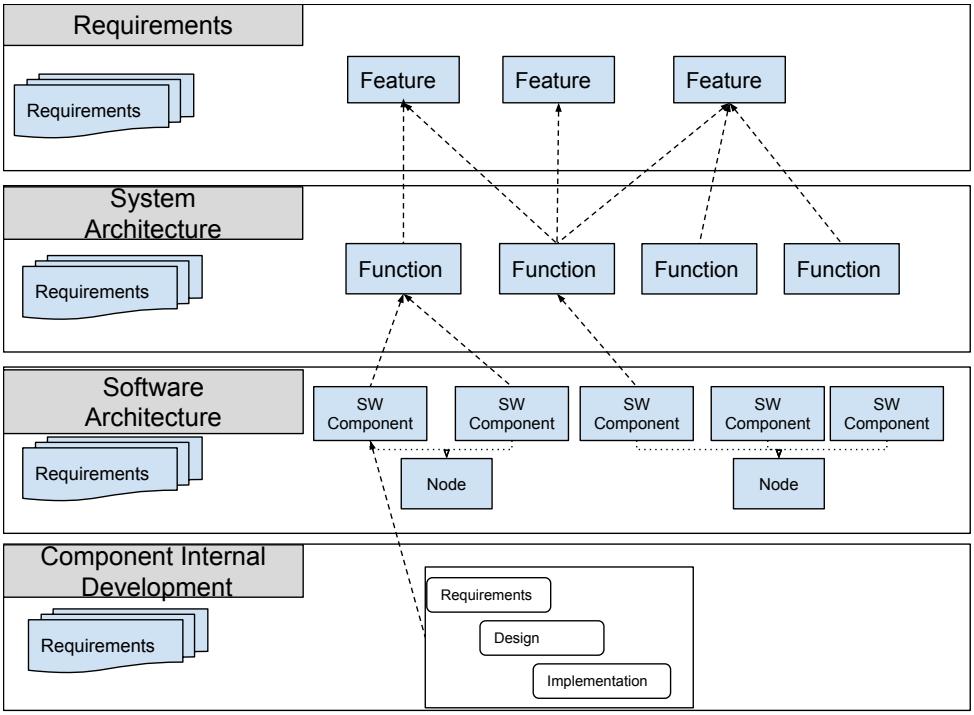

Figure 2.1: Component-Based Software Development Process and Used Artifacts [1]

2.4 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. MBE based solutions seem very promising to meet the needs of secure and dependable application 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.

2.4.1 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 [29].

A metamodel is a model that defines an abstract representation of models. As shown in Figure 2.2, the Object Management Group (OMG) identified three degrees of abstraction on top of a reality layer. The Meta Object Facility (MOF) [30], 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).

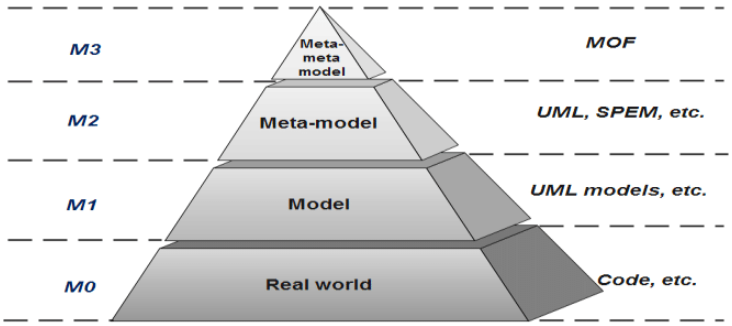

Figure 2.2: Modeling pyramid of the OMG [2]

2.4.2 Model-Driven Engineering (MDE)

The concept of a model is a major paradigm in software engineering. Its use represents a significant advance in terms of level of abstraction, continuity, generality, scalability, etc. MDE is a form of generative engineering [31], in which all or a part of an application is generated from models. MDE is an approach that 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 (M2M) 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 [32]), but in many other

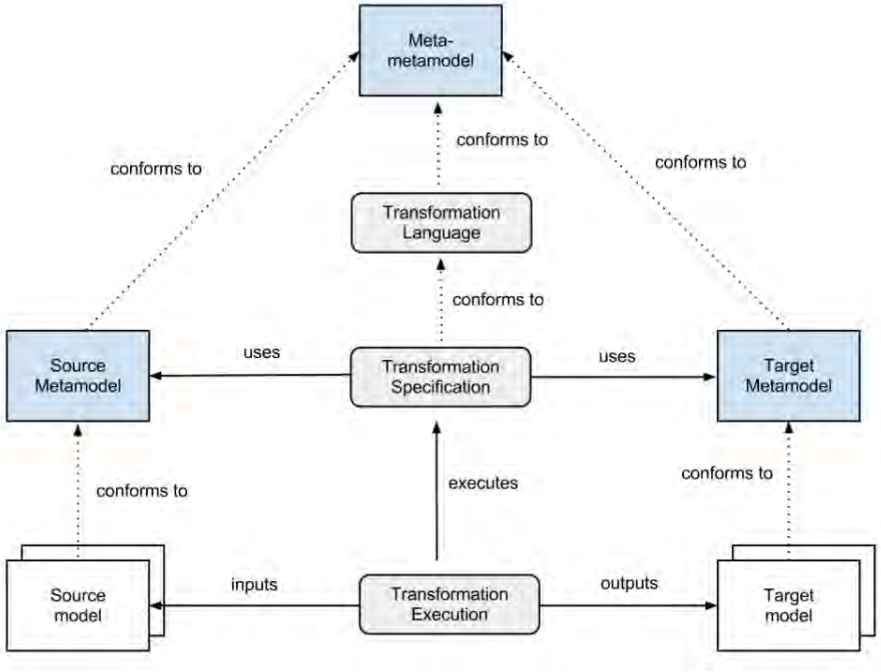

Figure 2.3: MDE: Overview on M2M Transformation [3]

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.

A M2M transformation specifies mechanisms to automatically create target models from source models. The OMG defines a M2M transformation as: *the process of converting a model into another model of the same system* [3]. As similar definition for a M2M transformation is provided in [33] as *the automatic generation of a target model from a source model, according to a transformation description.* Figure 2.3 shows the conceptual background on M2M transformations as they are used in MDE.

The Meta-Object Facility (MOF) [34] is a standard defined by the OMG to describe modeling languages such as the Unified Modeling Language (UML) [35]. Query View Transformation (QVT) [36], based on the Object Constraint Language (OCL) [37], is an OMG standard to specify M2M transformations in a formal way, between metamodels conforming to MOF.

2.4.3 Domain-Specific Modeling Language (DSML)

In software engineering, Domain Specific Modeling (DSM) [38, 39] is a methodology that uses models to specify applications within a particular domain. A Domain-Specific Modeling Language (DSML) is a language that is 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. DSMLs are frequently used in MDE [40].

A language is defined by an abstract syntax, a concrete syntax and the description of semantics [41, 38, 42]. 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. 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 [39]. 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.). 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.

2.5 Incorporating Security in System and Software 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 system architects, application domain-specific knowledge and security expertise at the same time to manage the potential consequences of design decisions on the security of a system and on the rest of the architecture. For instance, at the architectural level, security means having a mechanism (it may be a component or integrated into a component) to protect the system and its assets. Once a system is designed, it must be assessed to detect and evaluate risks in order to treat them.

Approaches taking into account security aspects in software/systems engineering are often considered as Software Security Engineering [43]. In this section, we will analyze the state-of-the-art of different approaches in software systems security engineering. In the first section, we will take a look at the integrated approaches taking into account the engineering life cycle from requirements engineering down to software release. In a second section, we will analyze more specific approaches in the MDE domain, which are in general less holistic and are specialized on different phases, such as requirements engineering or system design.

2.5.1 Generic Software Systems Security Engineering

For the study of existing general-purpose security engineering approaches, we limit ourselves to approaches covering a broad spectrum of the development life cycle and proposing an extensive set of security-oriented activities. We will focus on the three forefront representatives, namely Microsoft's Security Development Life cycle (SDL), OWASP's Comprehensive, Lightweight Application Security Process (CLASP) and McGraw's Touchpoints, as they are recognized as the major players in the field. These three secure software development approaches or processes have been extensively validated, either by usage in large-scale development projects [44], reviews by security specialized companies [45, 46] or by being inspired by industrial projects [47]. An overview will also be given on further standards or approaches.

Microsoft SDL. Microsoft's Security Development Life cycle [48] is probably the most rigorous, most tool supported and most oriented towards large organizations (e.g., Microsoft uses it internally). Microsoft defined this process in 2002 to address security issues frequently faced in development. It contains an extensive set of (security-oriented) activities, which can be used as supporting activities in development process models. These activities are often related to functionality-oriented activities and complement them by adding security aspects. Proposed activities are grouped into classical development phases (i.e., Education, Design, Implementation, Verification, Release) to ease the introduction into existing approaches. Vast guidance, such as detailed description of methods and tool support is available, enabling even less qualified practitioners to achieve the required outcome. These guidances go as far down as to give coding and compiling guidelines, which do not map to process model activities anymore.

CLASP. The Comprehensive, Lightweight Application Security Process (CLASP) [49] by the OWASP Consortium is a lightweight process containing 24 main security activities. It can be customized to fit different projects (activities can be integrated) and focuses on security as the central role of the system. The main focus of CLASP is to support engineering processes in which security takes a central role. For this approach, the foundations of a secure system are built in the architectural design and focus is given on this part of the process model. The activities proposed are developed to cover a wide range of security aspects and are conceived from a security-theoretical perspective and defined in an independent manner to allow a process designer, wishing to integrate them, a large field of flexibility. Recommendations are given on how to integrate these activities in an existing process, but there is no direct mapping and the coordination is less direct than in other approaches (such as in SDL or Touchpoints). CLASP also offers a rich set of security support resources, such as an extensive list of security vulnerabilities which can be used at different checkpoints throughout the process. The drawbacks of the approach defined by the OWASP consortium are mainly that some activity descriptions, although crucial for secure software development, fail to give detailed methodological indications, and the lack of work product descriptions for the proposed activities.

Touchpoints. McGraw's work [50] is based on industrial experience and has been validated over time. It provides a set of best practices regrouped into seven so-called

touchpoints. These touchpoints express the interactions among process developers and security and how the developer can take into account the security aspects by using the framework (e.g., Risk Management, Attack Analysis, Code Review, but also Examples and Basic Security Knowledge). The activities focus on risk management and flexibility and offer white-hat and black-hat approaches to increase security. For McGraw, risk management is of elemental importance in software security. The approach tries to enable this by providing a risk management framework supporting the security activities. In [47] McGraw offers, in addition to an extensive set of security knowledge and links to resources, a rich set of examples on security analysis activities and solutions. In giving general guidelines and adaptive activities, the approach can be tailored to most existing software processes focusing on the touchpoints of the existing process and the proposed security enhancements.

2.5.2 Model-Based Software Systems Security Engineering

Model-based software systems security engineering approaches tackle security aspects at different phases of the development. From the organizational context over requirements engineering down to system design and implementation, different independent approaches exist. Several approaches have been proposed in the literature dealing with security engineering in the requirements phase. Using abuse frames to model and develop the constraints of security requirements on functional requirements and trust assumptions is proposed in [51, 52], allowing the extension of problem frames to determine security requirements. This allows defining security requirements as constraints on functional requirements and trust assumptions. Another approach for security-oriented requirements engineering is proposed by extending use cases to misuse cases [53, 54] to elaborate security threat identification. The idea behind this approach is to describe functions the system should not allow, eliciting security requirements and the following constraints on assets.

System design model-driven software engineering processes use UML profiles such as SecureMDD [55], SecureUML [56] and UMLsec [57, 58] providing formal specifications for verification of security-oriented systems. SecureUML provides a UML profile based on Role-Based Access Control (RBAC) allowing specifying access control in the overall system design. This information can be used to generate access control infrastructures, helping the developers to improve productivity and the quality of the system-under-construction. SecureMDD proposes a methodology which allows generating platform-specific models (e.g., JavaCard) from a high-level stereotyped UML model. In addition to guidelines in modeling security aspects, the framework offers verification based on a formal approach on the produced models [59]. UMLsec is a UML profile aiming to support modeling of security-critical systems. The profile allows expressing security relevant information within the existing model and diagrams and thus taking security aspect into account in the overall system development. In addition to approaches focusing on one phase of the development life cycle, one notable holistic MDE approach is given in the literature. Lee et al. [60, 61] proposed an integration model for integrating security engineering approaches into software life cycle standards, mapping the concepts of the

software life cycle [62] to security engineering concepts (a set of concepts collected from various security engineering approaches). The approach tries to give an understanding to stakeholders where and when security activities intervene and interact with standard process life cycle activities.

2.6 Formal Techniques for Specification and Verification

Formal techniques [63] refers to mathematically rigorous techniques and tools for software and hardware system specification, design, and verification. The term 'mathematically rigorous' refers to specifications used in formal methods that are well-formed statements in a mathematical language, and formal verification that are rigorous proofs of these statements. A logic is the definition of such a language with a way to decide if a statement is valid. A formula is a sentence in the language. A formula's truth can be determined in one of two ways: syntactically or semantically. A syntactic deduction is a finite sequence of formulas that begins with an axiom (arbitrary true formula) and proceeds by adding some inference rule to each formula in the sequence. The semantic approach aims to provide an intuitive basis for formula definitions of fact. Depending on the application field, many families of logics have been investigated. Constructive logics are used in computer science with the aim of generating code from a software specification. Instead of a value, each formula has a proof that shows is validity (true or false).

In the following sections, we presented an introduction to the logics and formal techniques that will be used in this thesis.

2.6.1 Logics

First Order Logic (FOL)

First-order logic (FOL) [64], also called the quantificational logic, is the "usual" mathematical logic. FOL is a logical system for reasoning about properties of objects. The usage of quantified variables and predicates, in addition to the links between logical components, is the main characteristic of first-order logic. We can then formalize sentences by saying that if x meets the condition P, it also satisfies the property Q for all x.

FOL augments the logical connectives from propositional logic. Propositional logic is a formal system in mathematics and logic. The system is made of a set of propositions. Each proposition has a truth value, being either true or false. Propositions can be joined together using logical connectives to make new propositions :

- Negation: $\neg p$
- Conjunction: $p \wedge q$
- Disjunction: $p \vee q$
- Implication: $p \Rightarrow q$
- Biconditional: $p \Leftrightarrow q$
- True: $\top$

	... operate on ...	... and produce
Connectives($\Leftrightarrow$, $\wedge$, etc.) ...	propositions	a proposition
Predicates(=, etc.) ...	objects	a proposition
Functions ...	object	an object

- False: $\bot$

For example :

$$p \ \vee \ q \Rightarrow r$$

is a proposition indicating that p or q implies r.

In FOL, this system is extended with :

- *predicates* that describe properties of objects
- *functions* that map objects to one another
- *quantifiers* that allow to reason about multiple objects

Predicates. Predicates reason about objects in a declarative way. A predicate is applied to a set of arguments to form a proposition, which is either true or false.

For example :

$$is\_the\_partner\_of(person1, person2)$$

is a predicate indicating that *person1* is the partner of *person2*.

Equality FOL includes a particular predicate = that determines whether two objects are equivalent. Equality can only be applied to objects; it cannot be used to compare two propositions.

For example :

$$person1 = person2$$

is a predicate indicating that *person1* and *person2* are equivalent.

Functions. Functions return objects associated with other objects. Functions, like predicates, can take unlimited number of parameters but always return a single value.

For example :

$$get\_partner(person1)$$

is a function that return the *partner* of *person1*.

Quantifiers. Each quantifier has two parts: (1) the variable that is introduced, and (2) the statement that's being quantified. The variable introduced is scoped just to the statement being quantified.

Existential Quantifier A statement of the form $\exists \ x \ \cdot \ p$ is true if , for *some* choice of x, the statement p is true when that x is plugged into it.

For example :

$$\exists\ x\ \cdot\ has\_a\_partner(x)$$

is a proposition that is true if some x exists such as the predicate $has\_a\_patner(x)$ is true, i.e., if some x has a partner.

Universal Quantifier A statement of the form $\forall\ x\ \cdot\ p$ is true if, for *every* choice of x, the statement p is true when x is plugged into it.

For example :

$$\forall\ x\ \cdot\ has\_a\_partner(x)$$

is a proposition that is true if for all x the predicate $has\_a\_partner(x)$ is true, i.e., if all x has a partner.

Modal Logic (ML)

Modal logic (ML) [65] is another branch of logic that has its roots in philosophy. It was created to look at the use of the terms "necessarily" and "maybe" in reasoning. Modal formulas are composed of propositions, which are atomic arguments, and modal operators, which deal with the concepts discussed above. In this section, we introduce some fundamental concepts of modal logic.

Uni-modal logic. Given a set $\mathcal{P}$ of atomic propositions (i.e., propositions which cannot be broken down into other simpler proposition) the propositional modal language $\mathcal{ML}$ is defined as :

$$\varphi ::= p \mid \bot \mid \varphi\ \Rightarrow\ \varphi \mid \Box\varphi$$

where $\bot$ is the constant *false* proposition and $p \in P$ is an atomic proposition.
The classical boolean and $\Rightarrow$ operators are defined as follows.

$$\begin{aligned}
&\neg\varphi \equiv \varphi \Rightarrow \bot\\
&\varphi_1\ \vee\ \varphi_2 \equiv (\neg\varphi_1) \Rightarrow \varphi_2\\
&\bot \equiv \neg\top\\
&\varphi_1\ \wedge\ \varphi_2 \equiv \neg(\neg\varphi_1\ \vee\ \neg\varphi_2)
\end{aligned}$$

$\Box$ is called the necessity modal operator. Usually, but not necessarily, $\diamond$ is called the possibility operator and is defined as $\diamond \equiv \neg\Box\neg$, the dual of $\Box$.

Multi-modal logic. A multimodal logic is a modal logic that has more than one primitive modal operator. Given a set P of atomic propositions, the propositional multimodal language ML_n is defined as :

$$\varphi ::= p \mid \bot \mid \varphi\ \Rightarrow\ \varphi \mid \Box_1\ \varphi, ..., \Box_n\ \varphi$$

Normal modal logic. A normal logic is a set $\mathcal{L}$ of modal formulas such that $\mathcal{L}$ contains :

- all axioms of propositional logic
- all instance of the Kripke schema (**K**) : $\Box(A \Rightarrow B) \Rightarrow (\Box A \Rightarrow \Box B)$

and it is closed under the following inference rules

- Detachment rule (modus ponens): $A \Rightarrow B$, $A \vdash B$
- Necessitation rule: $\vdash A \Rightarrow \Box A$

The smallest logic normal logic is called **K**. Most commonly used modal logics are extensions of **K**. These modal logics are obtained by extending the system **K** with a set of new axioms. We note a model logic L extending system **K** with a set ω of new axioms as $L \equiv K \oplus \omega$

In the following, we list some a the most common modal logic obtained by supplement **K** with new axioms:

- $\mathbf{KD} \equiv \mathbf{K} \oplus \Box\varphi \Rightarrow \diamond\varphi$
- $\mathbf{KT} \equiv \mathbf{K} \oplus \Box\varphi \Rightarrow \varphi$
- $\mathbf{S4} \equiv \mathbf{KT} \oplus \Box\varphi \Rightarrow \diamond\Box\varphi$
- $\mathbf{S5} \equiv \mathbf{S4} \oplus \Box\varphi \Rightarrow \diamond \diamond \varphi$

Modal operator. A modal operator (also known as a modal connective) is a logical connective used in modal logic. It is an operator that creates propositions out of propositions. A modal operator is distinguished "intuitively" by expressing a modal attitude about the proposition to which the operator is applied. Knowledge [66], obligation [67], and temporal [68] are some examples of use modal operators showing that modal logic is used today as more general way for various types of concepts.

Examples of common modal operator

Epistemic logic. Epistemic logic is an example of modal logic applied for knowledge representation. The multi-modal operators reflect the level of knowledge, ignorance and belief in the possible world. The $\Box_x$ operator written as $\mathbb{K}_x()$ is translated as "x knows that ...".

For example, we express in epistemic logic "Bob knows that Alice has a partner" as:

$$\mathbb{K}_{bob}(has\_partner(alice))$$

Temporal logic. Temporal logic is an example of modal logic applied for propositions qualified in terms of time. For example, the sentential tense logic has four modal operators (in addition to all usual operators in first-order propositional logic):

- $\mathbb{P}$: "It was the case that..." (P stands for "past")
- $\mathbb{F}$: "It will be the case that..." (F stands for "future")
- $\mathbb{G}$: "It always will be the case that..."
- $\mathbb{H}$: "It always was the case that..."

For example, we express in temporal logic "Bob will eventually have a partner" as:

$$\mathbb{F}(\mathit{has\_partner}(\mathit{bob}))$$

2.6.2 Event-B

For the this research, we decided to use Event-B. We aimed to use a proof-based formal language for developing our tools, and Event-B met this criterion while also aligning with the expertise available within our research team.

In this section, we provide a brief overview of the Event-B formalism with the help of a small example. The example chosen is the traffic light system for a crossroad, as described in the Event-B Handbook [69]. This system models the state of the traffic lights at a crossroad, showcasing the main principles of Event-B.

Event-B language

Event-B is a formal modeling language grounded in set theory and first-order logic. It is both a simplification and an extension of the B formal method. Event-B adopts an event/state-based approach [70], allowing for the precise definition of system behavior and properties.

Formally, an Event-B model is defined by a tuple $(d; c; A; v; \Sigma; I; \mathit{Init}, E)$, where:

- d: carrier sets (data types),
- c: constants,
- $A(d; c)$: a conjunction of axioms defining properties of model data structures,
- v: a vector of the model variables,
- Σ: the model state space defined by all possible values of v,
- $I(d; c; v)$: a conjunction of invariants specifying the properties that must be preserved across all states,
- Init: a non-empty set of model initial states ($\mathit{Init} \subseteq R$) with R the set of all the possible states of the machine,
- E: a set of model events, where each event e is a relation of the form $e \subseteq R \times R$.

Event-B models are divided into two main components: the *MACHINE*, which describes the behavioral aspects, and the *CONTEXT*, which defines the static properties of the system, such as sets, constants, and axioms.

Variables and invariants are declared in the MACHINE. Variables are strongly typed using invariants, ensuring correctness. For example, in Listing 2.1, two variables (*cars_go* and *peds_go*) are declared along with three invariants, stating that the variables are Booleans and cannot both be true simultaneously.

```
1  variables
2      cars_go
3      peds_go
4
5  invariants
6      @inv1 cars_go ∈ ℙ
7      @inv2 peds_go ∈ ℙ
8      @inv3 ¬(cars_go = TRUE ∧ peds_go = TRUE)
```

Listing 2.1: Variables and invariants

Events in Event-B are defined in the general form:

$$e \triangleq \textbf{any } lv \textbf{ where } g \textbf{ then } act \textbf{ end}$$

where:

- lv: local variables of the event,
- g: guards (predicates) defining the conditions under which the event can occur,
- act: actions performed by the event.

The guards serve two purposes:

- They type the local variables, and
- They define the event's conditions.

An event can only be triggered if its guards are satisfied, based on the current state of the MACHINE.

For instance, in Listing 2.2, two events control the *peds_go* variable. The *set_peds_go* event requires that *cars_go* is false, ensuring no conflict at the crossroad.

```
1      event set_peds_go
2      when
3          @grd1 cars_go = FALSE
4      then
5          @act1 peds_go := TRUE
6      end
7
8      event set_peds_stop
9      begin
10         @act1 peds_go := FALSE
11     end
```

Listing 2.2: *peds_go* events

In contrast, the *set_cars* event (Listing 2.3) uses a parameter *new_value* to update *cars_go*. This event demonstrates the use of local variables, which are typed and checked against guards.

```
1  event set_cars
2  any
3      new_value
4  where
5      @grd1 new_value ∈ ℬ
6      @grd2 new_value = TRUE ⇒ peds_go = FALSE
7  then
8      @act1 cars_go := new_value
9  end
```

Listing 2.3: *set_cars* event

Event-B semantics

Event-B semantics is based on the concept of *before-after predicates* [71], which describe the relationship between the system's state before and after an event occurs. A before-after predicate is expressed as:

$$\exists t \cdot (P(t, v) \land x' = F(t, v)) \land y = y'$$

where:

- t: temporary variables,
- v: system variables that change,
- $P(t, v)$: predicates involving t and v,
- x': new values of the changed variables, determined by $F(t, v)$,
- y: variables that remain unchanged ($y = y'$).

In Event-B, these predicates are embedded within events, ensuring that the system's state transitions preserve all defined invariants. Proof obligations are generated to verify these transitions and ensure consistency.

Refinement

Event-B employs a top-down refinement methodology to incrementally develop system models. Refinement involves introducing additional details, such as new variables or events, while preserving the properties of the abstract model. This approach supports the concept of *correctness by construction*, where properties proven at an abstract level are maintained in subsequent refinements.

For example, in our traffic light system, we refine the variable *peds_go* into *peds_color*, representing traffic light colors (Red or Green). This refinement requires a *gluing invariant*, such as the one shown in Listing 2.4, to link the abstract and concrete variables.

```
1 variables
2   cars_go
3   peds_color
4
5 invariants
6   @inv1 peds_color ∈ {Red, Green}
7   @gluing peds_go = TRUE ⇔ peds_color = Green
```

Listing 2.4: Concrete variable and gluing invariant

2.7 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.

2.7.1 Eclipse Modeling Framework

There are several Domain Specific Modeling (DSM) environments, one of them being the open-source Eclipse Modeling Framework (EMF) [72]. Meta-Object Facility (MOF) provides an implementation of Essential MOF (EMOF), a subset of MOF, called Ecore2. MOF 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 EMF. 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 DSL 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 [72].

- MOF 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 MOF-based tools and applications [72].

- Rich Client Platform (RCP) plug-in allows developers to use the Eclipse platform to create flexible and extensible desktop applications upon a plug-in architecture [73, 74].

- Acceleo is a M2T transformation language that is part of the EMF ecosystem. It is specifically designed to generate textual artifacts (such as code, documentation, or configuration files) from EMF-based models. Acceleo adheres to the MDE approach and supports the OMG standard for Model-to-Text (M2T) transformations.

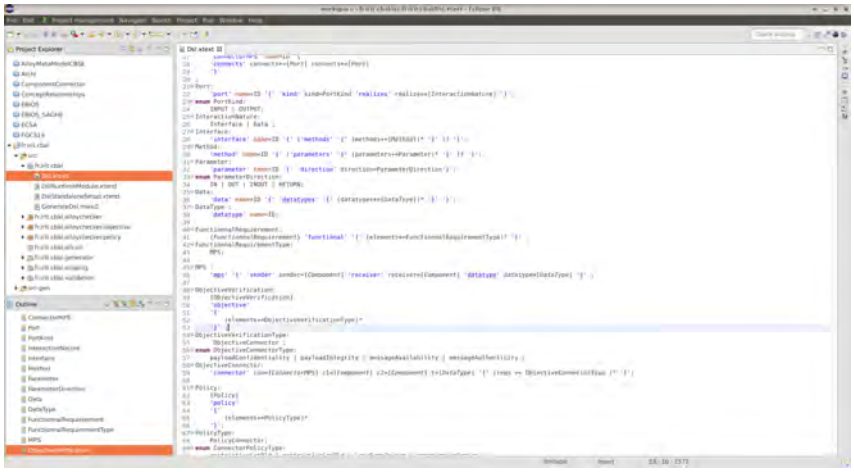

Figure 2.4: Example of the Eclipse Modeling Framework UI workspace

Sirius

Sirius is an Eclipse-based open-source framework that enables the specification and generation of custom graphical modeling workbenches, tightly integrated with the EMF. It allows users to define graphical editors (diagrams, tables, trees, etc.) for domain-specific models without having to write low-level graphical code.

Sirius operates on top of EMF metamodels and uses a declarative approach to define how models should be visualized and edited. The central concept in Sirius is the *Viewpoint Specification Model (VSM)*, which allows the designer to specify:

- **Graphical Representations**: including diagrams, node styles, edge mappings, and layout rules.
- **Model Navigation and Edition**: tools for direct editing, creation, and deletion of model elements through the graphical interface.
- **Conditional Styling**: dynamic adaptation of visual elements based on the underlying model state.

Sirius is particularly useful in MDE processes where visualization and interaction with complex models are essential. It promotes separation of concerns by allowing the graphical representation to evolve independently of the underlying model and iterative refinement of modeling environments.

In the context of this work, Sirius is used to build an interactive modeling environment aligned with the conceptual metamodel, facilitating model construction, consistency checking, and integration into broader MDE toolchains.

Figure 2.5: Example of a model diagram representation using Sirius

2.7.2 Event-B Tools

To support the modeling and verification process, a rich ecosystem of tools has been developed around Event-B, primarily centered on the *Rodin Platform*. These tools provide a comprehensive framework for the specification, validation, and verification of complex systems using Event-B. They enable rigorous development workflows where correctness can be ensured through both proof and simulation.

In this section, we present the main tools that are used in this work, namely the *Rodin platform* and *ProB*

Rodin Platform

The Rodin platform [75] is an integrated development environment (IDE) specifically designed to support Event-B. Developed alongside the formalism, Rodin provides a robust set of tools that facilitate the creation, verification, and validation of Event-B models. It includes features for defining machines and contexts, generating proof obligations, and interacting with automatic solvers.

One of Rodin's key components is its proof assistant, which plays a central role in verifying the correctness of models. Proof obligations are generated automatically to ensure consistency, invariant preservation, and refinement correctness. The toolkit integrates both automatic and interactive proof mechanisms. While most proof obligations can be discharged automatically using solvers, some require manual intervention. The interactive proof assistant allows users to apply custom proof strategies or simplify statements to a point where automatic solvers can handle them.

Rodin also supports multiple extensions that enhance its capabilities. One notable extension is the Camille text-based editor [76], which provides a flexible and user-friendly interface for creating Event-B models. Another crucial extension is ProB, which enables the animation and validation of Event-B models (See Section 2.7.2).

The combination of Rodin's core tools and its extensions creates a comprehensive environment for modeling and analysis. It bridges the gap between theoretical formalism and practical application, empowering users to create models that are both mathematically rigorous and aligned with real-world system requirements. By providing these capabilities, Rodin ensures that Event-B can be used effectively for the specification, design, and validation of complex systems, making it a vital tool for research and development in formal methods.

Figure 2.6: Event-B model representation using Rodin editor

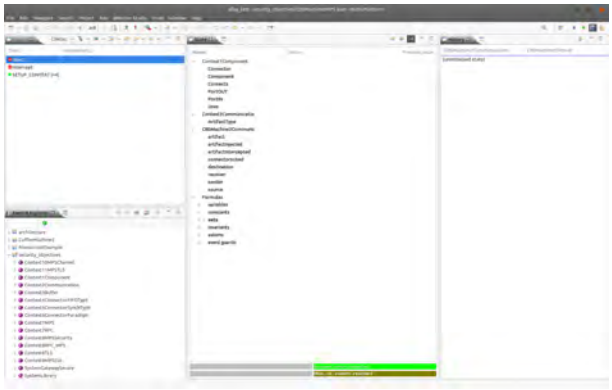

Figure 2.7: ProB UI for the simulation of an Event-B model in Rodin

ProB

ProB is an animator, model checker, and constraint solver for the B-Method, Event-B, and other formal notations. It is designed to support validation and verification of formal models through automatic and interactive exploration of their state spaces. ProB allows users to animate models, find counterexamples to invariants, check refinement relationships, and detect deadlocks or other inconsistencies.

In the context of Event-B, ProB is integrated into the Rodin platform. ProB complements it by enabling simulation and model checking capabilities that go beyond deductive proofs. Through this integration, users can seamlessly switch from proof-based verification in Rodin to animation and model checking with ProB, offering a more complete verification framework.

This combined usage enhances confidence in the correctness of models, especially in complex systems where interactive proofs may be insufficient or difficult to discharge. By using ProB alongside Rodin, developers can detect logical flaws early in the modeling phase, ensure consistency across refinement levels, and validate properties through automated techniques.

2.8 Introduction to the Case Studies

This section contains a brief presentation of the case studies used in this dissertation. First, we present a college library web application that will be used as an illustration of the presented concepts throughout the dissertation. Then, we present a smart meter gateway case study that we will employ to assess our proposed approach.

2.8.1 College Library Web Application

As an illustrative example, we will consider a college library web application system [77], such as that visualized in Figure 2.8. The web application provides online services for searching for and requesting books. The users are students, college staff, and librarians. Staff and students will be able to log in and search for books, and staff members can request books. Librarians will be able to log in, add books, add users, and search for books. Informally, among the set of requirements that the software system to build should fulfill, we will focus on three functional requirements as examples. We specify them as follows:

- *F_Req_1. It should be possible for a user to visualize a book page.*
- *F_Req_2. It should be possible for the database to transmit data to the webserver.*
- *F_Req_3. It should be possible for the administrator to write a shared configuration variable in the database.*
- *S_Req_1. The data transmitted to the database cannot be read by anyone else*
- *S_Req_2. The origin of the messages sent by the webserver is correctly identified*
- *S_Req_3. The data sent to the user by the webserver is the one being received*

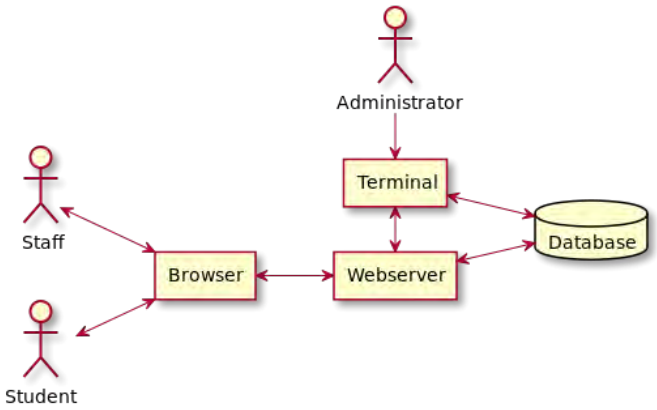

Figure 2.8: A college library web application example

We use a UML class diagram to describe the high-level architecture model of the web application, where software components are represented by classes, and relationships between these components are represented by associations. Figure 2.9 depicts the corresponding system architecture. For instance, there are implicit design decisions that links three entities (Webserver, Database, and Administrator) that collaborate in order to ensure that the application has clearly defined the user types and the rights of the said users. However, effective realizations of these connections are not modeled in the UML class diagram; they may be subject to certain changes and/or adaptations (e.g., new solutions, deletions, modifications of realization), verification (e.g., formal verification) and reuse (e.g., in the same domain or across domains) while the structure of the main software architecture can be maintained. Each connector represents a communication pattern which rigorous software developers, mainly architects, would like software modeling and analysis languages to easily express.

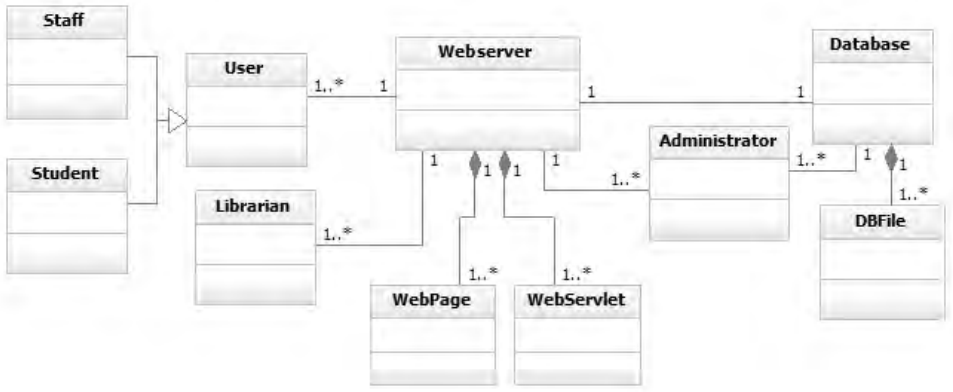

Figure 2.9: A UML description of the high-level architecture of the college library web application example

2.8.2 Smart Meter Gateway

As a case study, we use a smart meter gateway, which is a simplified version of a real Gateway Protection Profile [4] that enables connecting to several meters for different commodities, such as electricity, gas, water or heat, and communicating data with remote entities in a Wide Area Network (WAN) or Home Area Network (HAN). The system is depicted in Figure 2.10. The meter in the Local Metrological Network (LMN) is an electricity meter that is located in the same housing as the Gateway. The electricity meter communicates measurement information with the Gateway. The main function of this system is to ensure that the measurement information is processed in the gateway and exchanged with (1) the remote readout center (Rrc) in the WAN acting as an authorized external entity, and (2) with a customer in the HAN acting as a Consumer.

Below, we describe several selected cases of the smart meter system, presented as a set of functional and non-functional requirements.

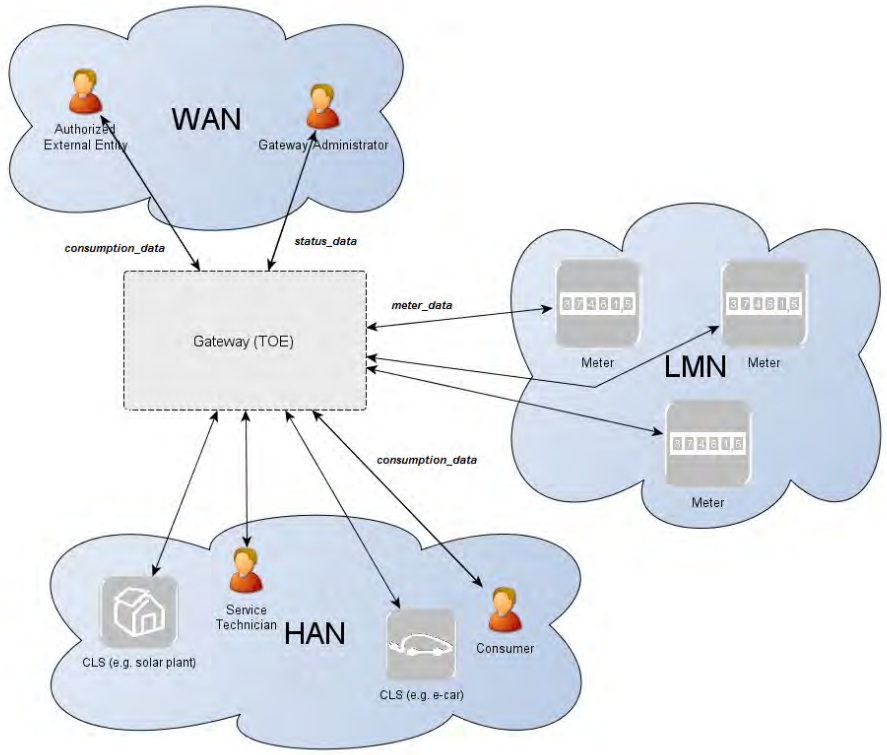

Figure 2.10: Actors and roles in the smart meter gateway scenario [4]

1. *F_Req_1.* Exchange measurement data on consumed commodities: (1) Gateway sends command to read data from the meter; (2) meter sends requested data to the Gateway; and (3) Gateway processes and stores data.

2. *F_Req_2.* Exchange measurement data and information on actual consumption: (1) Rrc sends command to read measurements from the Gateway and (2) Gateway sends measurement information to the Rrc.

3. *F_Req_3.* Exchange information on actual electricity consumption and total amount of energy consumed: (1) Customer sends command to read measurements from the Gateway and (2) Gateway sends measurement information to the Customer.

4. *S_Req_1. The data transmitted to the Gateway cannot be read by anyone else*

5. *S_Req_2. The origin of the messages sent by the metering components is correctly identified*

6. *S_Req_3. The data sent to the Gateway is the data received by the Gateway*

Chapter 3

State-of-the-Art

Contents

3.1 Introduction

The need for higher quality tools to support the development of secure systems is continuously increasing. Nowadays, secure practices must be implemented in more and more domains that did not traditionally deeply consider security concerns. New recommendations for the specification and verification of secure systems should be considered to ground security engineering on two pillars: solid theory and proven principles. We seek to focus these two pillars towards the development of a formal tool focused on the description and analysis of architecture models that can be incorporated in traditional system development lifecycles (SDLCs).

Considering the context stated in Section 1.1 and the research goals stated in Section 1.4, this chapter outlines the state-of-the-art of academic and industry practices in two domains of research, Software Engineering and Security Engineering. We give an overview of the existing work in these two domains. Section 3.2 details existing work on different approaches in software engineering and covers works on MDE and formal methods for developing software architectures. Section 3.3 focuses on the security aspect of software engineering describing SDLCs for developing secure software systems. It also covers MDE and formal methods applied to developing secure systems. Lastly, Section 3.4 positions our proposed contributions in the state-of-the-art.

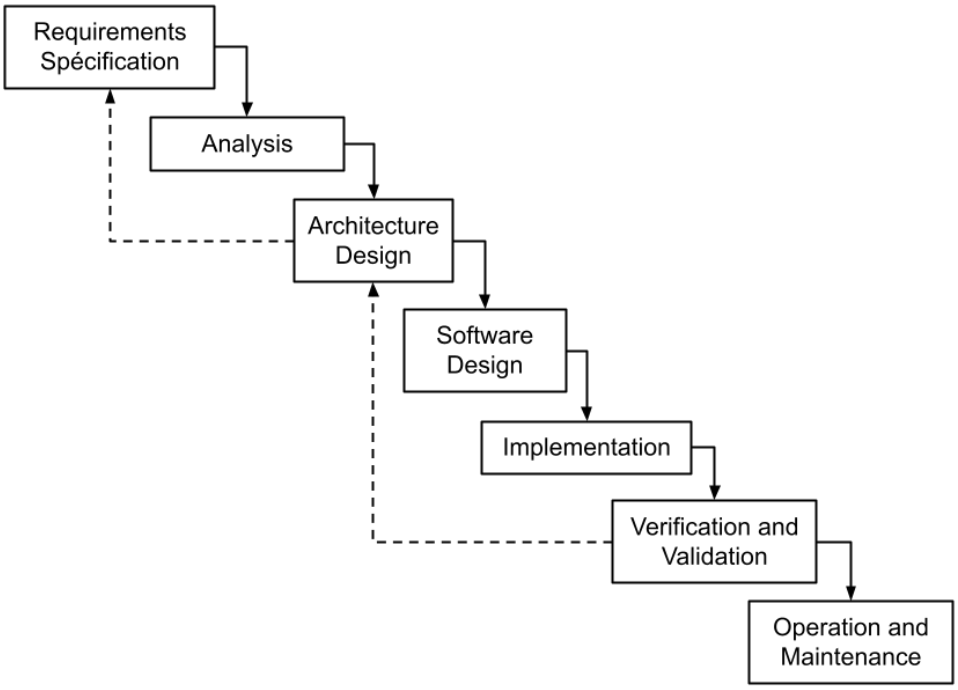

Figure 3.1: Royce iterative waterfall SDLC

3.2 Engineering Software Systems

Early on in the world of developing large and complex software systems, the need of clear and simple method to produce big infrastructure was apparent. At a NATO conference in 1968, MD McIlroy made an address that is considered the basis of component-based engineering [78]. It initiated what was called the *software crisis* [79] which arose because the capabilities of the computers grew faster than the means to produce software for them. As such the concept of components was created as a way to reduce the complexity of the productions. Components are used as independent architectural entities that communicate between them using interfaces.

Our focus is on the architectural design of computer systems. A well-known system development lifecycle (SDLC) is the Royce iterative waterfall. As shown in Figure 3.1, the architectural design of a system takes place at the early stages of production. It is our belief that security concerns should be addressed early in production. In our approach we aim to address part of the first three phases of this SDLC mainly focusing of the analysis and architecture design following the requirements.

3.2.1 Software Architecture Design

Recently there has been a shift when it comes to software architecture design [80]. This shift integrates multiple paradigms instead of separating them, namely, component-based

development [81], model-driven engineering [82] and formal methods [83]. Many description languages and formalisms have been proposed in the literature to model complex distributed systems by using this approach of multi-paradigms. These works aimed to capture the communication, concurrency, and non-functional properties of the components of given systems.

The literature around Component-Based Systems (CBS) focuses on the description of the interaction between components. Shin et al. [84] proposed a software product line approach by modeling the variability of secure software connectors and to promote connector reusability, while Bures and Plasil [85] modeled four basic component interaction types, namely message passing, remote procedure calls, streaming, and blackboard. This paper sets the basis for the theoretical aspect of our work, and we present a metamodel that is an interpretation, in part, based on these foundations. It shows the complexity of the interactions that exists when working on the communication in the context of CBS. It shows that clear and formal specifications of such behaviors is needed to study such systems.

The real-time aspect of embedded systems was studied by Neto et al. [86]. In this work, the authors provide a multi-platform framework (MARTe) that differentiates hardware, algorithm, and the system configuration. Czarnecki et al. [87] presented a method to represent ports using a UML notation. This approach reduces the possible semantic ambiguities when using composite ports and supports reuse. By using a description that associates connectors as simple links between ports, these papers differ from our interpretation which is closer to how components are represented. We describe component behavior independently which allows us to create libraries of connectors using different paradigms or protocols.

3.2.2 Model-Driven Engineering Approaches for Engineering Component-Based Systems

Models manifest in diverse forms across various contexts within the realm of software engineering. Their principal function is to tailor the representation of complex systems by presenting essential information that may be of interest to the user. MDE is a form of generative engineering, in which all or a part of an application is generated from models. MDE aims to reduce the overall complexity of describing system by using different levels of abstraction [7], thereby promising to improve their quality and reduce their development life cycles. By creating models of the desired system instead of directly working on the system, we create an approach that is reusable and easy to discuss and modify before extensive implementation. In this work, we aim to create a model-based approach that will allow for the description of CBS using formal methods.

In software engineering, UML stands as the industry standard for visualizing, specifying, constructing, and documenting software systems [88]. It provides a graphical language to describe systems interactions. It is a flexible language that can be updated or extended to follow the pace of change in software development.

Following the use of abstraction in computer engineering and more modern languages,

Schmidt [7] proposed a model-driven approach as a means to address platform complexity that was growing and might reach a ceiling at the time. Hutchinson et al. [89] worked on the deployment of model driven methods in the industry and the requirements this method needs for a successful use in a professional setting. These works show that the use of MDE is needed both in theory and practice. The growth of complexity of computer systems in recent years confirm that the abstract nature of MDE is an advantage.

Gossler and Siffakis [90] proposed a multi-level framework for the description of CBS. They used a very simple definition of component only describing what action they can do and the interaction they can have with other components. By using this definition, they described the requirements that the components must follow for a seamless integration despite the Heterogeneity of their behaviors. This paper aimed to reduce the complexity of designing CBS by creating models and their requirements. This abstract representation of components allows for an easier description, and as such reduces the complexity of the resulting model and architecture. However, this work only focuses on the structural aspect of CBS and does not touch upon the communication aspect.

Goos et al. [91] proposed a metamodel-based approach to describe models using UML. In doing so, they created an abstract and concrete syntax, as well as a semantic domain to allow for the creation of intelligent tools that can be generated based on the resulting specifications. Kounev et al. [92] introduced an MDE-based approach to model CBS using event-based communication to enable interactions between components. The approach is applied for the modeling and performance prediction of event-based system at the architecture level. These papers show that a separation of an abstract and concrete semantics, as well as an event-based approach can be used to model CBS. In this proposed work, we aim to use both abstract and concrete semantics applied to an event-based language to model architectures.

3.2.3 Formal Methods for Engineering Software Systems

In addition to the approaches presented in the previous sections, different formal languages have also been used in engineering software systems and in the architecture design stages. Process algebra has been used in different forms to describe systems. Milner [93] presented a comprehensive method to describe distributed systems using Calculus of Communicating System (CCS) and extended that language later as π-calculus [94] to describe changing systems, Hoare [95] used Communicating Sequential Processes (CSP) to show that the input, output and concurrency of components can be manipulated as primitive of CBS. Bergstra and Klop [96] studied synchronous system using Algebra of Communicating Processes (ACP) based on the previous work of Milner. Using process algebra as its basis, Ferrara [97]) created a framework dedicated to the design and validation of web services called LOTO. Aside from process algebra, automata are also used to describe systems. Alur and Dill [98] used timed (finite) automata to describe the behavior of real-systems. Kaynar et al. [99] described a complete theory of I/O automata for the behavior of distributed system. Beek et al. [100] extend the theory of I/O automata incorporating process algebra. This allowed to describe and validate properties both at the structural

and behavioral level.

A general methodology for the specification of CBS was presented by Allen and Garlan [101]. It uses formal semantics based on a subset of CSP to describe architectural connectors. This work focused on the interactions between components. While this paper showed a general and adaptable approach to CBA using custom notation and definitions, it lacks the tool support to verify the overall correctness of the approach which makes it difficult to use in practice. Ke et al. [102] proposed a new method (rCOS) that focuses on the integration of Model-Driven Architecture (MDA) in existing development processes and its integration at the different stages. It also describes the verification and validation process in this method. This paper introduced a formal tool that allow for the specification and validation of component-based systems, but the constraints put on the behaviors of the components fail to represent the variety of behaviors that exist.

Alloy has been used as a model checker formal language to study CBS. Khosrav et al. [103] used it to model and analyze Reo connectors. Alloy has also been used by Rouland et al. [16] to formally specify component-port-connector systems by representing connectors not only as links between components, but closer to components where they are able to simulate different behaviors and create a library of reusable connectors. In this paper, properties of the connectors and overall behaviors are verified using model checking tools. This approach is interesting to detect flaws in the system specifications, but lacks definitive answers that proof-based methods can provide.

In this proposed work, we choose to use Event-B as the formal language for the design and analysis of component-based software systems. Event-B has been used a lot in the research world over the years and we can find several works on CBS. Mammar et al. [104] presented an approach to describe CBS. Their work focuses on composites where they describe a set of components as one bigger component like a white box, and the correct construction of CBA using those composites, More specifically, the work focusses on the structural aspect of the construction. We can also find work focusing on more concrete component-based systems using Event-B. For example, Bodeveix et al. [105] focused on a variability-aware component model to test the structural correctness of systems obtained by the formal application of patterns. Babin et al. [106] studied the verification of compensation for web services to formally verify their behaviors. Graiet et al. [107] worked on the verification and deployment of elastic component-based applications, as well as the verification of dynamic composite services [108]. Regarding the deployment aspects of CBS, Attiogbé [109] used Event-B to check correctness of deployment of a CBS on a deployment environment modeled as a set of nodes hosting the components. These papers show that the specification and verification of CBS using Event-B is possible. They also cover both small and large scale models from a specific system specification to a whole process. These papers often focus on either structural or behavioral aspects but fail to present both in a single model.

3.3 Engineering Secure Software Systems

As critical systems play an increasingly central role in modern infrastructure, research efforts have intensified to identify processes that facilitate the development of secure software systems. In this section, we present an overview of general approaches to integrating security within the SDLC. Additionally, we review key research contributions that explore the integration of security concerns with various methodologies, including MDE and formal methods.

3.3.1 Secure Development Processes

Engineering secure software requires the use of reliable and trusted system development processes. We will limit ourselves to existing general purpose security engineering approaches that cover a broad spectrum of the SDLC. In this category, three representatives are widely acknowledged as the primary figures in the domain, namely Microsoft's Security Development Lifecycle (SDL), OWASP's Comprehensive, Lightweight Application Security Process (CLASP) and McGraw's Seven Touchpoints. Each process has been extensively validated, either by security specialized companies [45, 46] or by large industrial projects [110, 111].

Microsoft SDL. Michael Howard and Steve Lipner presented the Microsoft Security Development Lifecycle (SDL) [112]. This process is composed of seven stages: training, requirements, design, implementation, verification, release and response. It follows a more traditional SDLC (Waterfall, V-model) from the training of the organisation employees to the response in case of an attack on the deployed system. It follows a threat analysis process following the STRIDE (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege) approach [113].

CLASP. Open Worldwide Application Security Project (OWASP) and its Comprehensive, Lightweight Application Security Process (CLASP) [114] moves the security analysis of the system in the early stage of development. It divides the security process in activities and resources. It contains 24 main security activities that can be integrated to fit different projects. It focuses on security as one of the main aspects of the project. It ranges from, an awareness program that aim to train project member to consider security as a project goal, to the publishing of security guideline aimed at the product users. The activities span at every stage of a product lifecycle and their execution are left open to the different to the modeller's perspective. This SDLC have some drawbacks however, the activity description fails to give detailed methodological indications which is crucial for secure software development.

Seven Touchpoints. The Cigital's Seven Security Touchpoints approach [111] is a review of industrial experience and has been validated over time. The touchpoint approach includes both "negative" (i.e., attacks, exploits, software breaking, etc.) and "positive"

(i.e., design, defence, etc.) approaches to security and the activities associated with them. The seven touchpoints are (i) code review, (ii) risk analysis, (iii) penetration testing, (iv) risk-based security tests, (v) abuse cases, (vi) security requirements, and (vii) security operations. This SDLC provides activities focuses on risk management and flexibility, and provides a framework supporting to support these objectives. The touchpoint activities can be tailored to most existing projects.

These secure SDLCs answer a need for a clear process when developing a secure system. All three of them include activities that span from the first design to the operational aspects of secure system development. Activities introducing the architecture of such systems mark the foundation of projects and as such must be addressed with a high level of assurance. Our work aims to use proof-based formal methods in these activities such that security, architecture, and formal methods experts can work together in achieving a high level of assurance.

3.3.2 Model-Driven Engineering Approaches for Engineering Secure Software Systems

As mentioned in Section 3.2.2, MDE is extensively used in software development. This use does not only focus on functional requirements of a system but can be extended to non-functional requirements such as security requirements. For example, we can find two extensions of UML dedicated for security purpose SecureUML [115] and UMLsec [116]. These extensions aim to create a modeling support of systems intended for security concerns (i.e., access control...). Our proposed work aims to provide a modeling language to integrate security into system development to design secure systems.

In a recent systematic review of model-driven approaches for safety-critical and security-critical software systems, Mashkoor et al. [117] managed to extract 143 publications relevant in this domain. In this systematic mapping we can see that while significant work has already been published mainly proposing new approaches for describing risks, MDE-based approaches for dealing with security is still in its infancy. In their conclusion, the authors call for an advancement in tool and methodology development in which we aim to contribute with this proposed work.

Despite MDE-based approaches for dealing with security still in its infancy, several researchers have aimed to build secure architectures using MDE principles. Glässer et al. [118, 119] used Abstract State Machines (ASM) to help design architectures with safety and security in mind. ASM is a method that allow for the specification and analysis of real work problems by describing them as a set of entity and actions using a state-based approach. This description can then be transformed into usable code that can simulate the system. In this case, security requirements were introduced into this model and then tested using the resulting code. In this paper, the models represented the decision support for marine operations and were not focused on distributed systems.

Best et al.[120] used UMLsec for the analysis of a distributed information system of a car manufacturer. This work aimed to create a clear annotation of the critical area of the system and create a better understanding of its possible issues. This paper showed

an industrial application of modeling languages. However, despite providing a better understanding of the system, the approach did not propose any potential solutions to the issues detected.

Casola et al. [121] used threat modeling to develop and test a methodology to build secure DevOps pipelines. They focus their approach on a set of models to detect potential threats as well as the security controls that can help mitigate them using a reusable library. This paper showed the lack of security driven development for DevOps pipelines and provided a threat modeling based approach to solve this issue.

Bouhziz et al. [122] used a model-driven approach using UML to integrate security patterns into CBS. They focused their approach at the design phase and aimed for an approach that can be used by non-security expert. This paper uses a similar approach to our proposed work in order to offer non-security expert tools to implement security solutions. However, their approach only uses UML notations for the design and lacked the formal solution we target in the proposed work.

3.3.3 Formal Methods for Engineering Secure Software Systems

The formalization of security concerns aims to build systems with strong security guarantees and resiliency to security threats. Formal methods aim to achieve a high level of assurance by providing proofs of the compliance to certain security properties. Kulik et al. [123] performed a survey of the use of formal methods in aspects of cyber security by splitting them into three categories: theorem proving, model checking, and lightweight formal methods. This work concludes that while formal methods may not be able to detect new security concerns, it can formalize and study even complex attacks. It also shows that the steep learning curve and lack of complete tool support hinder the use of theorem proving formal methods. This review shows that while the need for formal methods in safety-critical systems outlined by Landwehr [124] has improved over the years, it still requires some additional research, especially in proof-based methods.

As outlined in the survey, more than 30 years ago Burrows et al. [125] pioneered the use of formal methods applied to security. It created some basis on the use of formal methods applied to security protocols by answering a set of simple questions such as “Does this protocol work?”, “Can it be made to work?”, etc. The authors introduced the BAN (Burrows–Abadi–Needham) logic as a set of rules to define and analyze communication protocols and applied it to authentication protocols as a way to answer those questions. In more recent work, Rodano et al. [83] presented an axiomatic approach to security properties using first-order logic. By integrating these axioms into the CORE tool filters and Innoslate (a cloud-native, model-based systems engineering solution) they were able to validate modeled architecture in different tools using the same formal representation. These papers show that the use of formal methods to specify and describe security aspect of communication has applications. Contrary to our proposed approach, these works focus mainly on the data transfer aspect of communication while we introduce a representation of the underlying behavior of connectors as part of our modeling.

Oheimb and Modersheim [126] introduced a formal specification tool named ASLan++

that specializes in the specification and analysis of distributed systems. This language uses an entity and agent specification close to what any object-oriented language may provide with classes. It uses model checking to verify security goals and security properties described with first-order logic in order to verify the validity of the system modeled. In their paper, Kindler et al. [127] proposed a composite approach to formal methods. Instead of relying on a single formal analysis they build a component tool designed to take input from different formal modeled in the form of a scenario. This approach allows for both a formal application of multiple formal methods but also to present the results hiding the underlying complexity to the user. These papers show that integrating formal methods in more complete tools and framework produce better results than a single formal model. In our work, we aim to place formal models in the center of complete framework; not simply as an input.

Following their architectural representation of component-based systems in Alloy, Rouland et al. [16] used the architectural model as basis to represent security threats in Alloy [15] as well as libraries of policies that mitigate those threats. A following work using the Coq proof assistant [128] aimed to represent the relationship between security policies and objectives in a component based system using message-passing communication. This work uses the mathematical foundation of Coq to represent those relationships as formulas and proves them in a semi-interactive manner. These works set the theoretical basis and practical possibility of our proposed approach. We aim to integrate Event-B as the language of choice to answer some of the limitation that appeared in their work (state explosions, etc.) and go further by integrating those results in a more complete theoretical and practical framework validated using recognized examples.

Among formal methods Event-B has also been used in the context of security engineering and for the modeling and analysis of security using the capabilities of its proof-based method. Laibinis et al. [129] used Event-B to simulate attacks on communication protocols using initial knowledge and protocol primitives to gain access to critical data and use the potential attacks to propose changes preventing them. They model the protocol primitives as constants in the Context. The knowledge of every actor is a function mapping every actor to the bit of knowledge it has at a point of time. Finally, the protocol is animated by events allowing to execute every primitive with its respective conditions stated in the Context as axioms. Benaissa and Mery [130] created guidelines to formally describe communication protocols to support cryptography key establishment using Event-B. They used refinement to introduce the attacker's action and to identify potential vulnerabilities. They applied these guidelines to the Needham-Schroeder public key protocol and the Blake-Wilson-Menezes key transport protocol. They use events to represent the transitions of the protocols as well as the actions that an attacker could perform on the system. Invariants are used to specify and verify the protocol's properties. Rivera et al. [131] worked on an open problem of the Tokeneer system funded by the National Security Agency (NSA). They use this problem to build a model in Event-B and generate Java code using EventB2Java plug-in to both verify security properties of the system and also demonstrate the possibility of creating a prototype of the system backed by the formal specification. Guards are used to model security "properties" (constraints)

constraining the system to ensure that the model respect the security objectives of the original project. Invariants are added to prove the validity according to the requirements. These papers focus on the specification of known vulnerabilities and protocols and not architectural aspects.

Hoang et al. [132] specified access control using an illustrative example. They described the functional and security aspect of their model separately and then used machine composition to merge both aspect into the same machine. Therefore, they verify the security hypothesis in the composed model. Security properties of the system are represented as invariant backed up by the model guards and events. Gawanmeh et al. [133] described an approach for group key secrecy and used refinement to verify forward secrecy as well. Secrecy properties are modeled as invariants in the model and backed up by the guards and events. These papers manage to represent security properties in component-based systems, but they did not apply any architectural concerns and only work in isolation.

3.4 Positioning

With the rise of multi-paradigms integration in software engineering and the rising concern for computer security, our work aims to provide support for specifying systems at various level of abstraction. By combining model-drivel engineering and formal methods we offer both a flexible and verifiable view of communication for component-based systems. In contrast to our work, other modeling and formal languages for capturing the communication and non-functional requirements of complex distributed systems do not directly provide such a simple and understandable view or a level of assurance reaching proof-based methods.

As far as we are aware, none of the approaches outlined in Section 3.3 incorporate the validation of security solutions into the MDE refinement process of the application. Unlike other formal security engineering methods [124, 134, 135], our work is designed around the desired security properties and associated assumptions and not the attacks nor threats.

In Table 3.1 we provide an overview of the relevant work outlining the themes studied and tools used relevant to our work. It outlines work presented throughout this chapter and includes whether the publication uses formal methods and what language was used, if it is designed to support reuse, if it covers security or architectural concerns, if it comes with some sort of tool support, and finally whether it uses a Domain-Specific Language (DSL). Table 3.2 provides an overview of the references using formal methods with the type of formal methods used and the specific language used. Table 3.3 shows the references with a focus on security and their focus.

In this work, we propose a metamodel for the description of security in the context of a component-based software architecture and message-passing communication, augmented with a formal interpretation in Event-B. Concretely, in our work, we provide a formal support for the specification and analysis of policies (i.e., security solutions), objectives (i.e., security problems) and the relationships between them, thereby supporting

the principles of security-by-design in a well-documented SDLC. The resulting Event-B model will describe the functional and security aspects (structural and behavioral) of a software architecture and enable these aspects to be specified and verified at different levels of abstraction.

Table 3.1: Related works relevance to this thesis and our positioning

Publication	FM	Language(s)	Reuse	Security	Architecture	Tool	DSL
Bures et al. (2004) [85]	✓	SOFA	✗	✗	✗	✗	✓
Kounev et al. (2013) [92]	✗	Diagram, QVT-O	✓	✗	✗	✗	✗
Neto et al. (2010) [86]	✗	BaseLib2, CINT C++	✓	✗	✓	✓	✓
Schimidt et al. (2006) [7]	✗	UML	✓	✗	✓	✗	✗
Goos et al. (2002) [91]	✗	UML	✓	✗	✗	✗	✗
Milner et al. (1980) [93]	✓	CCS	✓	✗	✗	✗	✗
Hoare et al. (1978) [95]	✓	CSP	✓	✗	✗	✓	✗
Bergstra et al. (1984) [96]	✓	ACP	✗	✗	✗	✗	✗
Milner et al. (1992) [94]	✓	Pi-calculus	✗	✗	✗	✗	✗
Ferrara et al. (2004) [97]	✓	LOTO	✓	✗	✗	✓	✗
Alur et al. (1994) [98]	✓	Timed automata	✗	✗	✗	✗	✗
Kaynar et al. (2011) [99]	✓	I/O automata	✗	✗	✗	✗	✗
Beek et al. (2003) [100]	✓	Team automata	✗	✗	✗	✗	✗
Allen et al. (1997) [101]	✓	WRIGHT, CSP	✓	✗	✓	✗	✗
Ke et al. (2012) [102]	✓	RCOS	✓	✗	✓	✓	✓
Rodano et al. (2013) [83]	✓	COREScript	✓	✗	✓	✓	✗
Khosrav et al. (2008) [103]	✓	Alloy	✓	✗	✓	✓	✗
Rouland et al. (2020) [16]	✓	Alloy	✓	✗	✓	✓	✗
Mammar et al. (2021) [104]	✓	Event-B	✗	✗	✗	✓	✗
Bodeveix et al. (2018) [105]	✓	Event-B	✓	✗	✓	✓	✗
Babin et al. (2015) [106]	✓	Event-B	✓	✗	✓	✗	✗
Graiet et al. (2017) [107]	✓	Event-B	✓	✗	✓	✓	✗
Attiogbe et al. (2016) [109]	✓	Event-B	✗	✗	✓	✗	✗
Graiet et al. (2013) [108]	✓	Event-B	✗	✗	✓	✗	✗
Shin et al. (2018) [84]	✗	Pseudo code		✓	✗	✗	✗
Landwehr et al. 1981() [124]	✓	Automata	✗	✓	✗	✓	✗
Burrows et al. (1990) [125]	✓	Automata/custom logic	✓	✓	✗	✗	✗
Oheimb et al. (2011) [126]	✓	ASLAN++	✓	✓	✗	✓	✓
Hutchison et al. (2006) [127]	✓	Petri Nets	✗	✓	✓	✓	✗
Howard et al. (2006) [112]	✓	Event-B	✗	✓	✓	✓	✗
Laibinis et al. (2016) [129]	✓	Event-B	✓	✓	✗	✓	✗
Benaissa et al. (2010) [130]	✓	Event-B	✗	✓	✗	✗	✗
Hoang et al. (2009) [132]	✓	Event-B	✗	✓	✗	✗	✗
Rivera et al. (2016) [131]	✓	Event-B	✗	✓	✗	✓	✗
Gawanmeh et al. (2012) [133]	✓	Event-B	✓	✓	✗	✗	✗
Proposed work	✓	Event-B, UML	✓	✓	✓	✓	✓

Table 3.2: Related works with focus on Formal Methods

Publication	Type	Language(s)
Milner et al. (1980) [93]	Process Algebra	CCS
Hoare et al. (1978) [95]	Process Algebra	CSP
Frappier et al. (2008) [136]	Process Algebra	ASTDs
Bergstra et al. (1984) [96]	Process Algebra	ACP
Milner et al. (1992) [94]	Process Algebra	Pi-calculus
Ferrara et al. (2004) [97]	Model Checking	LOTO
Alur et al. (1994) [98]	Timed Automata	Timed automata
Kaynar et al. (2011) [99]	I/O Automata	I/O automata
Beek et al. (2003) [100]	Team Automata	Team automata
Allen et al. (1997) [101]	Architecture Description	WRIGHT, CSP
Ke et al. (2012) [102]	Proof-based	RCOS
Rodano et al. (2013) [83]	Proof-based	COREScript
Khosrav et al. (2008) [103]	Model Checking	Alloy
Rouland et al. (2020) [16]	Model Checking	Alloy
Méry et Singh (2011) [137]	Proof-based	Event-B
Mammar et al. (2021) [104]	Proof-based	Event-B
Bodeveix et al. (2018) [105]	Proof-based	Event-B
Babin et al. (2015) [106]	Proof-based	Event-B
Graiet et al. (2017) [107]	Proof-based	Event-B
Attiogbe et al. (2016) [109]	Proof-based	Event-B
Landwehr et al. (1981) [124]	Model Checking	Automata
Burrows et al. (1990) [125]	Custom Logic	Automata/custom logic
Oheimb et al. (2011) [126]	Model Checking	ASLAN++
Hutchison et al. (2006) [127]	Petri Nets	Petri Nets
Graiet et al. (2013) [108]	Proof-based	Event-B
Howard et al. (2006) [112]	Proof-based	Event-B
Laibinis et al. (2016) [129]	Proof-based	Event-B
Benaissa et al. (2010) [130]	Proof-based	Event-B
Hoang et al. (2009) [132]	Proof-based	Event-B
Rivera et al. (2016) [131]	Proof-based	Event-B
Gawanmeh et al. (2012) [133]	Proof-based	Event-B
Proposed work	Proof-based	Event-B, UML

Table 3.3: Related works with focus on Security

Publication	Security Focus	Subject
Shin et al. (2018) [84]	Security patterns	SPL for secure connector
Landwehr et al. (1981) [124]	Threats	Information disclosure
Burrows et al. (1990) [125]	Objectives	Authentication
Howard et al. (2006) [112]	Threats	Threat analysis in SDLC
Laibinis et al. (2016) [129]	Vulnerabilities	Detect harmful scenarios
Benaissa and Méry (2010) [130]	Protocols	Cryptographic Protocols
Rivera et al. (2016) [131]	Vulnerabilities	Tokeneer Challenge
Gawanmeh et al. (2012) [133]	Protocols	Group Key Protocols
Proposed work	Objectives	CIAAA

Chapter 4

Approach

Contents

4.1 Introduction

In this chapter, we introduce our general conceptual vision to study security at the architectural design level from the positive view (i.e., objective) in the context of component-based software architecture development. To implement this vision, we propose a methodology for the creation of a design and analysis framework, and provide guidelines on how the resulting methodology can be integrated to develop secure systems within existing SDLC methodologies. This chapter is organized as follows: Section 4.2 presents the conceptual vision used by our research team and the basis for the methodology of this work. In this chapter we will present the application of our conceptual vision into a clear methodology applied to the security of CBS using formal methods. Section 4.3 described the overall methodology proposed for the creation of the design and analysis framework. Finally, Section 4.4 presents the integration of our work into an existing SDLC.

4.2 Approach Conceptual Vision

Security concerns are becoming increasingly important as critical software-intensive systems become more widespread. Our goal is to develop a methodological tool support for the design and analysis of such concerns and describe them as quality attributes of the

systems. With reuse as our priority, our hypothesis is that the creation of models and tools for analysis will help in the development of better, well-designed secure systems.

Our general conceptual vision shown in Figure 1.1 is refined into Figure 4.1 to reflect the security concerns we aim to address.

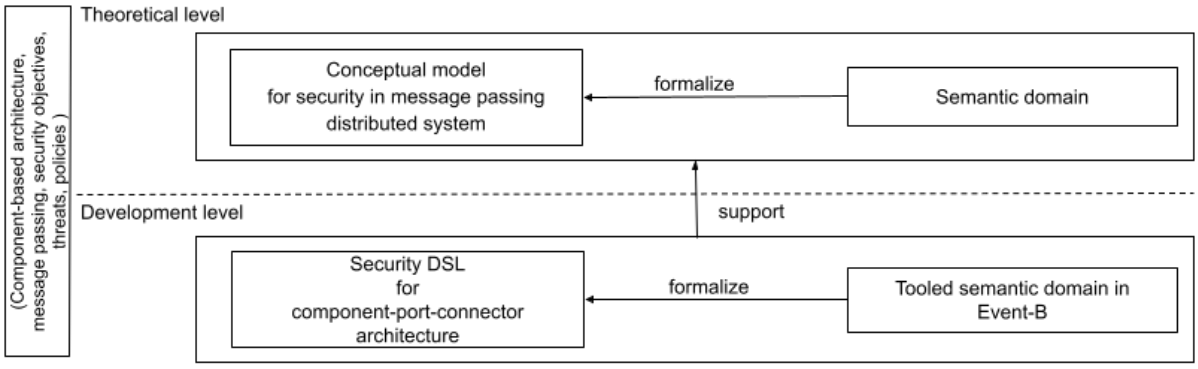

Figure 4.1: Conceptual vision applied to security

Our conceptual vision is based around four central aspects:

1. Separation of concerns

2. Model-driven approaches

3. Formal techniques

4. Component Based Software Engineering (CBSE)

The separation of concerns is based on the actors in a development process. Figure 4.2 show the relations between all the actors and the different components of the framework. The goal is to provide every actor with tools in which they are knowledgeable, as well as a complete framework that allows them to add their expertise into the project in a controlled and formal way.

In this vision, a separation is established between domain-independent modeling and system-specific development. Domain experts—such as distributed systems specialists, security analysts, and analysis experts—are responsible for creating reusable models that capture communication patterns, security properties, and analysis strategies. These reusable artefacts are then employed by system architects and software developers to construct software architecture models and associated security analysis artefacts. The resulting analysis provides feedback on potential vulnerabilities or security weaknesses, enabling iterative refinement of the software architecture to ensure the development of a secure system. The architect work is assisted by the different experts in order to produce the best architecture model.

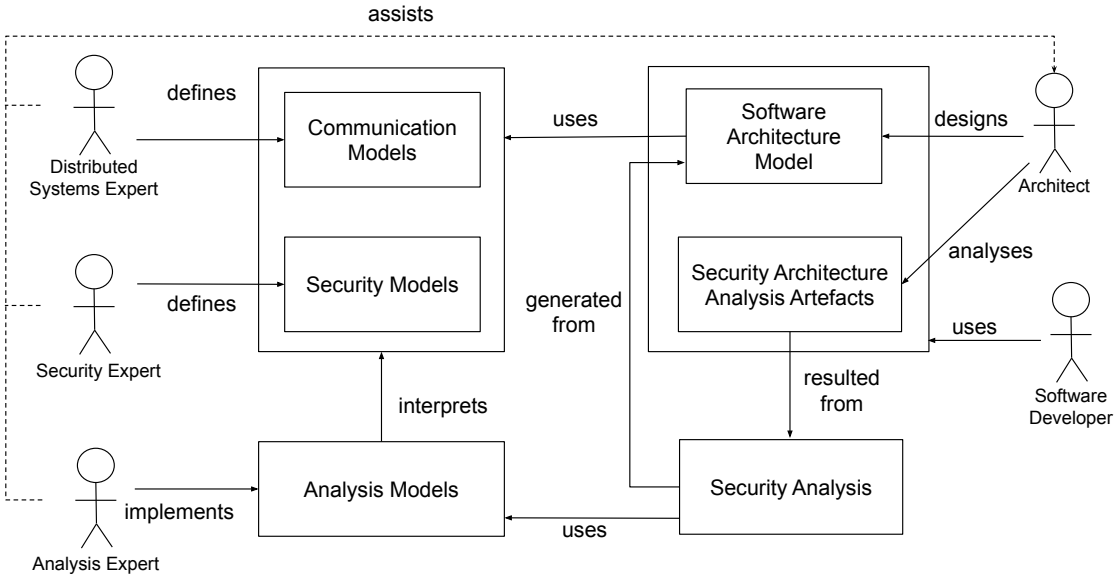

Figure 4.2: CBSE Process for Security engineering: An instance of the involved Stakeholders

4.3 Methodology for the Creation of a Design and Analysis Framework

In this section, we propose an implementation of the conceptual vision introduced in Figure 1.1 applied for the architecture and security concerns as described in Figure 4.1. Then, we describe a set of guidelines for the integration of the methodology in existing software production methodologies.

The proposed methodology supports the design and analysis of software architecture and concerns, as well as the introduction of security concerns in the form of objectives. The goal is to capture both functional and non-functional requirements of the system architecture and provide a reusable model in Event-B to study them. The use of formal methods enables the verification and analysis of potential security issues in the system. As depicted in Figure 4.3, the methodology is composed of several steps and activities, which are described below.

4.3.1 Modeling Framework Development Process

As depicted in Figure 4.3, we consider both the theoretical and development activities for the architecture and security viewpoints.

Theoretical Level. We begin by developing a software architecture metamodel capable of representing the functional requirements of the architecture. Then as an extension of this metamodel we construct a security objectives metamodel. This metamodel is built in a way that it allows for the creation of a semantics to describe formally security

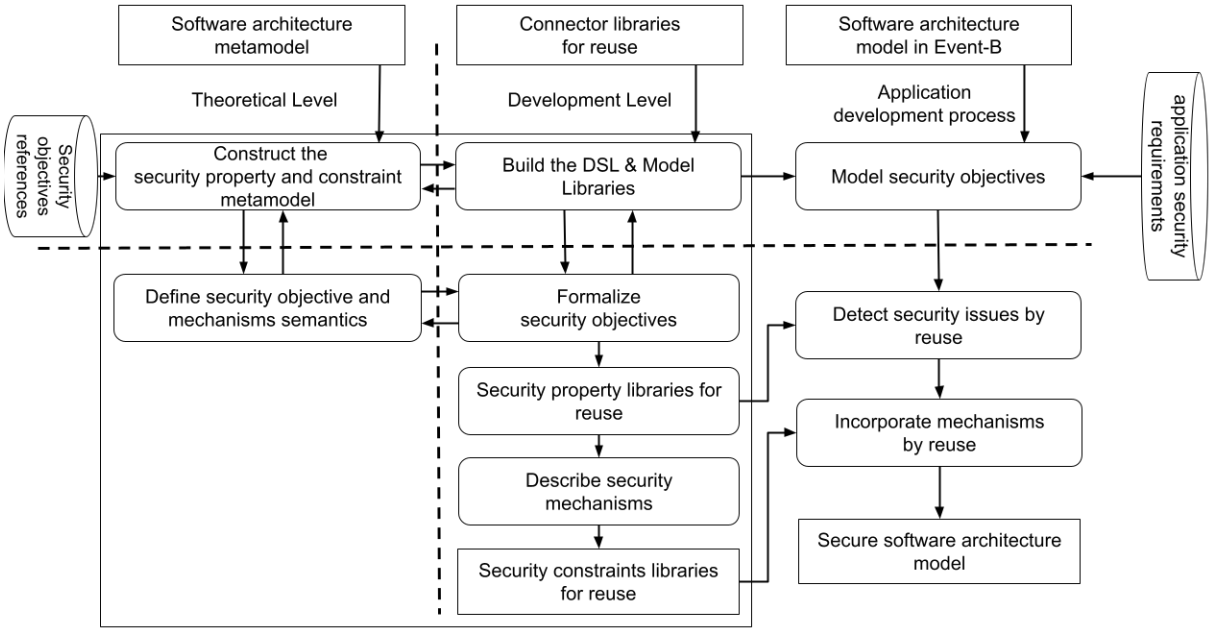

Figure 4.3: Methodology for the creation of a design and analysis framework

objectives and apply them to the architecture.

Development Level. After constructing the security objective metamodel and semantics, we formally specify a Domain-Specific Language (DSL) to describe the model. Then the metamodel is implemented in Event-B in a proof based fashion using its tools. By following the semantics defined in the theoretical level, we are able to formalize security objectives that are regrouped in libraries for reuse. Finally, security mechanisms are developed to answer those objectives and are grouped in the same libraries as the objectives.

4.3.2 Application Development Process

Once the modeling framework is developed it can be used in the context of an application. First, the architect must create the architecture of the application using the architecture metamodel. Then, the security requirements of the applications are modeled using the DSL alongside the functional architecture. This representation is then translated automatically in Event-B and the proof assistant of Rodin can detect possible security issues through undischarged proofs. These issues can be solved by incorporating mechanisms which at the end gives us a trustworthy secure system.

4.4 Supporting Security-by-Design within the SDLC

As shown in Figure 4.3, our approach is meant to be integrated into SDLC as an extension. In this section, we present one example of how to integrate our approach into a standard

process.

Our approach focuses on the relation between the requirements and the architecture. We believe that both functional and non-functional requirements should be studied as soon as possible to detect and correct potential requirements that are not satisfied by the architecture. We build a concrete architecture by instantiating the abstract meta-model for an application application-specific distributed software system. The functional requirements of the system are then verified by the primitive connectors of the model.

In Figure 4.4, we describe an example on how to integrate our approach into the Royce iterative waterfall SDLC. The activities defined in Section 4.3 come as a supplement to the existing phase as follows: (1) The requirements specification is extended with the "Formalize the (New) Libraries for reuse" activities. These activities concern the modeling framework development process and should only be done if a new interaction type is needed to be added in the framework, i.e., if a needed communication requirement is not already defined in the framework libraries. (2) The architecture design is extended with the "Model Software Architecture" and the "Analysis / Incorporate by Reuse" activities. The goal of these activities is to ensure that the software architecture model satisfies the desired properties for the designed system. (3) Finally, if we determine that the system design does not satisfy the desired properties, we take action to revisit the requirements or controls of the system. In this way, we can iterate over the requirements specification and architecture design phases of the SDLC to revise and improve the architecture design of the system.

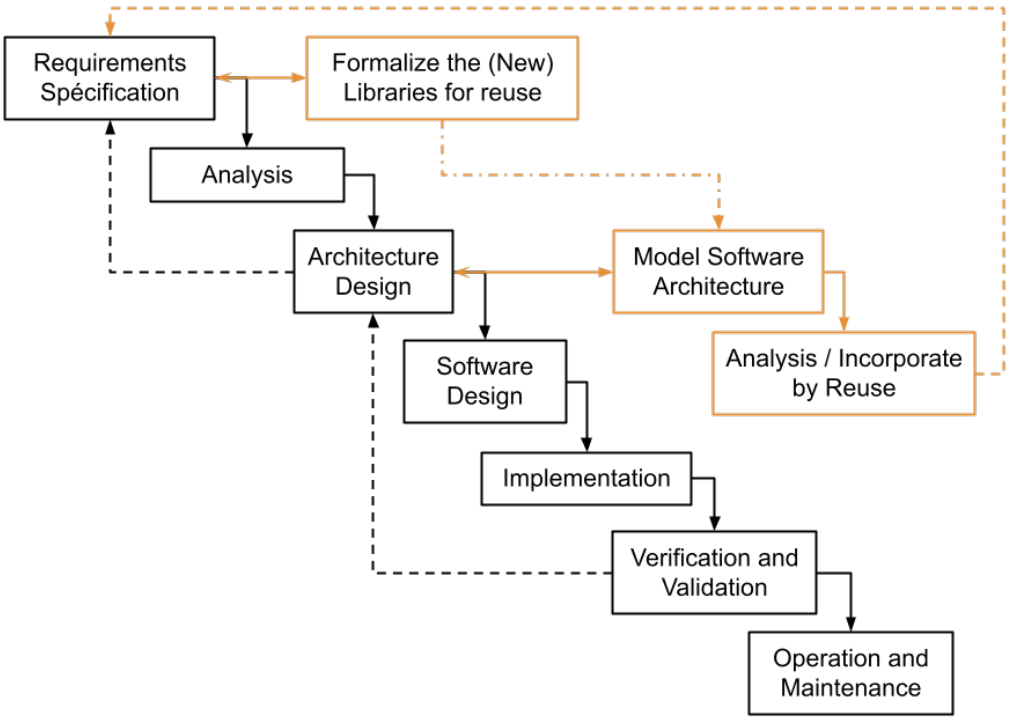

Figure 4.4: The proposed architecture design approach within the Royce iterative waterfall SDLC

Chapter 5

Model-Based Design of System Architecture and Security

Contents

5.1 Introduction

In this chapter, we introduce a model-based approach for the design of CBS. The proposed approach is built to define a metamodel capable of representing CBS for both functional and non-functional properties. The objective is to establish a comprehensive modeling framework that supports the specification of architectural structure, system behavior, and associated security properties. By integrating these aspects, the model representation facilitates a unified and extensible approach to modeling complex secure CBS. This chapter aims to answer the modeling aspects of our objectives namely **RO 1.1**, **RO 2.1** and **RO 3.1**.

The remainder of this chapter is organized around the metamodel packages illustrated in Figure5.1 as follows. The Architecture metamodel, presented in Section 5.2, provides a representation of CBS covering both structural (Logical) and behavioral (Scenario) aspects. Section 5.3 will present both the Property and Functional sub-package of the Analysis package. Finally, Section 5.4 will present the Security sub-package.

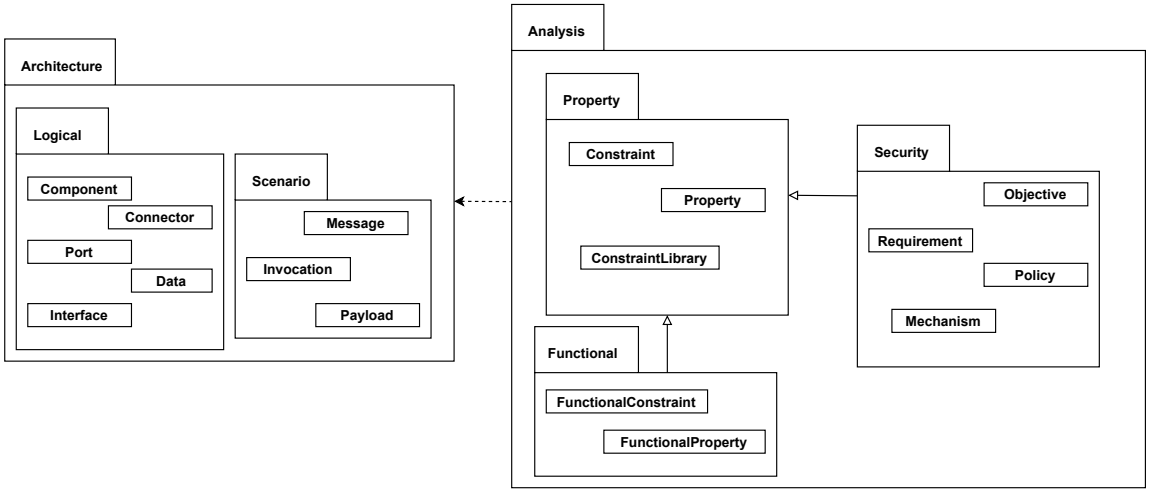

Figure 5.1: Metamodel packages overview of our modelization

5.2 Architecture Metamodel

The first step of our work as stated in Figure 4.3 is to create a metamodel to describe the different elements of CBA. Our goal is to: (1) provide a modeling language to specify the software architecture model from the user requirements and (2) specify the user requirements in the form of properties on the model to verify the fulfillment of the use requirements.

We use the software architecture metamodel defined by Rouland et al. [138] as a basis. The metamodel enables describing architectural models in a component-port-connector fashion. The metamodel concepts, as visualized in Figure 5.2, are conceptually close to the industrial practice (i.e., containing a UML-like [139], UCM-like [140, 141] and ADL-like [142] vocabulary). In addition, the metamodel concepts are structured in different packages [143], with a focus on the:

1. *Logical package* to capture the functional architecture of the system in terms of components. This view is concerned with the functionality that the system provides for the end user.

2. *Scenario package* which builds upon the logical view, describing the behavioral aspects of the system. This view is concerned with the representation of communication behavior between distributed components.

In what follows, we detail the principal classes of our metamodel, as described with UML notations in Figure 5.2.

- *Component.* A Component is a modeling artifact which represents a piece of software architecture. It has a set of Ports which realize InteractionNature defining how it can interact with other Components.

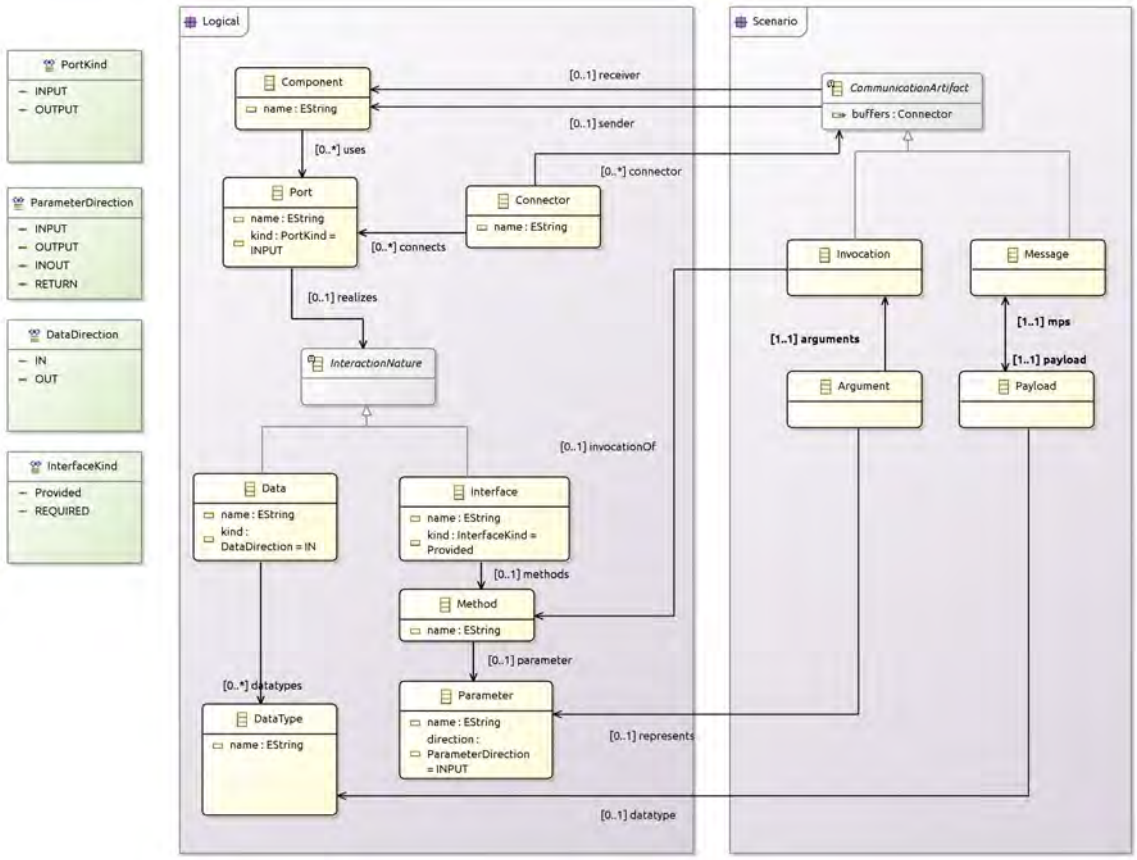

Figure 5.2: Component-port-connector metamodel

- *Port.* A Port is the interaction point through which a Component can communicate with its environment.

- *Connector.* A Connector specifies a link that enables communication between Ports by allowing the exchange of communication artifacts (cf. *Scenario view*, CommunicationStyle

- *InteractionNature.* An InteractionNature defines the nature of an interaction between two Components. This is an abstract concept. In our work, we have focused on defining two types:

  - *Data.* A Data defines the exchanged data between two components.
  - *Interface.* An Interface defines a set of public features (Methods). It specifies a kind of contract. Any Component that realizes (implements) this Interface must fulfill this contract. In this case, the Interface is PROVIDED by the component. For any Component that needs that interface (requires another component to fulfill this contract), the Interface is REQUIRED.

- *CommunicationArtifact.* A CommunicationArtifact represents a specific communication behavior between components (sender and receiver(s)). This is an abstract concept. In our work, we have focused on supporting the following communication paradigms:
  - *Message.* A Message is the representation of a message exchange from a sender Component that is producing it (i.e., a Component realizes an OUT Data with its corresponding DataType) and a receiver Component that is consuming it (i.e., a Component realizes an IN Data with its corresponding DataType). This interaction type is used for message passing interaction. At the time of sending (scenario), a *payload* is typed by the corresponding DataType.
  - *Invocation.* An Invocation, or commonly referred to in literature as *Remote Procedure Call*, is the representation of a call and the reply of the Method from a sender Component that requires it (i.e., a Component realizes a REQUIRED Interface that necessitates it) to a receiver Component providing it (i.e., a Component realizes a PROVIDED Interface that implements it). At the time of invocation (scenario), at set of *Argument* are used as a set of *Parameter* of the called *Method*.

For instance, interactions may be subject to specific characteristics (e.g., synchronous/asynchronous), to specific communication paradigms (e.g., message passing) and to specific technology implementations (e.g., HTTPS protocol). In our work, we embed the interaction semantics in a connector to represent a communication pattern. Therefore, rigorous software developers, mainly architects, would like software modeling and analysis languages to easily express connectors, providing operations to specify the functional (structural and behavioral) requirements they should fulfill. The basic idea of this extension is that the semantics of an interaction is defined by a certain port type and that one or more connectors can support this port type. The port types are already fixed at component design time, whereas the choice of a connector (and a specific interaction) is also constrained by the deployment characteristics. A connector has certain similarities with a component. The main difference is that it is dedicated for communication purposes. A connector is responsible for incoming, outgoing, intercepting, and blocking data and messages.

Example of our metamodel instantiation In the college library web application case study described in Section 2.8.1, the functional requirements are partitioned into structural and behavioral sub-requirements as follows. For illustration, we choose to use *Message* and *Invocation* interactions in the context of *Req_3* of the Library example.

- *Message*
  - *Str_Req_3.* Administrator and Database should be connected.
  - *Beh_Req_3.1.* It should be possible to send a message from Administrator to Database.

  - *Beh_Req_3.2.* The Database should be able to receive the message.

- *Invocation*
  - *Str_Req_3.* Administrator and Database should be connected.
  - *Beh_Req_3.1.* It should be possible to send an invocation from Administrator to Database.
  - *Beh_Req_3.2.* The Database should be able to receive the invocation.

Regarding the structural parts, we can identify the corresponding architectural concepts that will need to be instantiated in order to fulfill them, while the behavioral parts cannot be specified at this level of description.

- **Message.** We define two components, namely an *Administrator* and a *Database*. The Administrator needs to use a port that realizes a data producer. The Database needs to use a port that realizes a data consumer. Finally, it needs to use a connector to connect these ports in order to make possible the delivery of the messages between the Administrator and the Database.

- **Invocation.** We define two components, namely an *Administrator* and a *Database*. The Administrator needs to use a port that realizes a required interface containing a method *write*. The Database needs to use a Port that realizes a provided interface containing a method write. Finally, it needs to use a connector to connect these ports in order to make possible the delivery of write invocations between the Administrator and the Database

5.3 Analysis Metamodel

In this section, we present the *Analysis* package of the metamodel. Given the security-oriented focus of this work, we limit the scope here to the *Property* and *Functional* sub-packages. The *Security* sub-package, which addresses security-specific concerns, will be detailed separately in Section 5.4.

5.3.1 Property Metamodel

To integrate the properties described using the logical specification we need a model representation of those properties. In order to implement these properties we extend the metamodel presented previously with a new package named *Property*, as visualized in Figure 5.3. This extension is defined around two primary concepts: *Property* and *Constraint*.

- *Property:* A property characterizes an expected or desired behavior of the system, abstracted from the underlying mechanisms that ensure its enforcement. It defines

a high-level, declarative specification of system behavior, typically used to express correctness, security, or functional expectations independently of their implementation.

- *Constraint:* A constraint represents an intentional alteration of the system's behavior by imposing restrictions on its operation. These constraints may embody specific design patterns that are applied to guide or limit system functionality. Within the context of component-based systems, constraints can be applied to both *components* and *connectors*, serving to regulate interactions, enforce architectural styles, or fulfill non-functional requirements such as security or performance.

By making this distinction, we can see that the constraints contribute to the satisfaction of the properties. We also include a properties and constraints library structure to make reusable models.

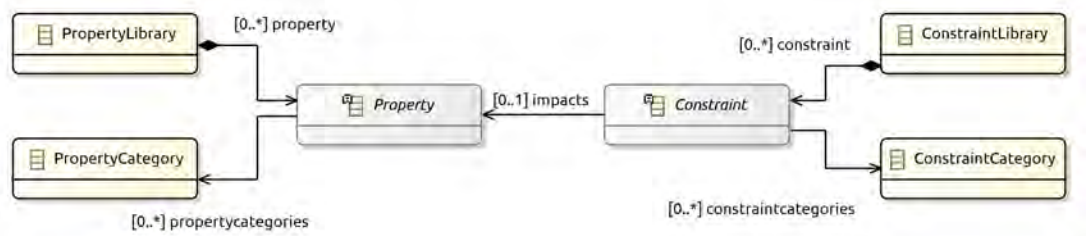

Figure 5.3: Property and constraint metamodel

5.3.2 Functional Metamodel

The Functional metamodel represents a simple refinement of the Property metamodel. It extends both concepts of *Property* and *Constraint* as respectively *FunctionalProperty* and *FunctionalConstraint*. Similarly to the previous section, the *FunctionalProperty* captures only a specific property of the system without implying any actual alteration of its behavior. In contrast, a *FunctionalConstraint* represents a modification of the system's behavior without necessarily guaranteeing any particular property.

Example of our metamodel instantiation

First-In First-Out (FIFO)

The FIFO property is a fundamental ordering principle applied to communication channels or data structures such as queues. In the context of system communication, particularly in distributed or component-based systems, the FIFO property ensures that messages or data units are delivered in the exact order in which they were sent.

In our models it can be interpreted as a functional property such that it guarantees that if the message m_i is sent before the message m_j ($i < j$), then m_i will be received before m_j at the destination.

FIFO is typically enforced by the underlying communication medium or protocol (e.g., TCP/IP), or it may be explicitly modeled and verified in system specifications. In this model, any *FunctionalConstraint* that alters the system's behavior in a way that ensures the satisfaction of the FIFO property can be considered applicable.

Example 5.3.1 *Ordering is defined as a category (i.e.,* PropertyCategory*) within the Communication library (i.e.,* PropertyCategoryLibrary*), and FIFO (the property selected for illustration) is defined as property (i.e.,* FunctionalProperty*) as part of this Ordering category. We also define* restrictiveReceive *as a category (i.e.,* ConstraintCategory*) within the access control library (i.e.,* ConstraintCategoryLibrary*). The* restrictiveReceiveFIFO *(the constraint selected for illustration) is defined as a constraint (i.e.,* FunctionalConstraint*).*

Synchronicity

Synchronous behavior in connectors refers to communication where the sender and the receiver are tightly coupled in time. A message or operation is only executed when both the sending and receiving components are ready to engage in the interaction. This implies that no buffering occurs; instead, both parties must synchronize for the data exchange to take place. In the context of CBS, synchronous behavior can be formalized as follows: a message m_j can be transmitted through a synchronous connector c only if all preceding messages m_i (where $i < j$) have been successfully received.

Example 5.3.2 Ordering *is defined as a category (i.e.,* PropertyCategory*) within the Communication library (i.e.,* PropertyCategoryLibrary*), and Synchronicity (the property selected for illustration) is defined as property (i.e.,* FunctionalProperty*) as part of this Ordering category. We also define* restrictiveSend *as a category (i.e.,* ConstraintCategory*) within the access control library (i.e.,* ConstraintCategoryLibrary*). The* restrictiveSendSynch *(the constraint selected for illustration) is defined as a constraint (i.e.,* FunctionalConstraint*).*

5.4 Security Metamodel

The properties and constraints described previously can describe security properties in a general manner. To differentiate between functional and non-functional properties and constraints we propose a refinement of the property constraint metamodel. In this section we are going to detail the Security package that refines the Property package (See figure 5.1).

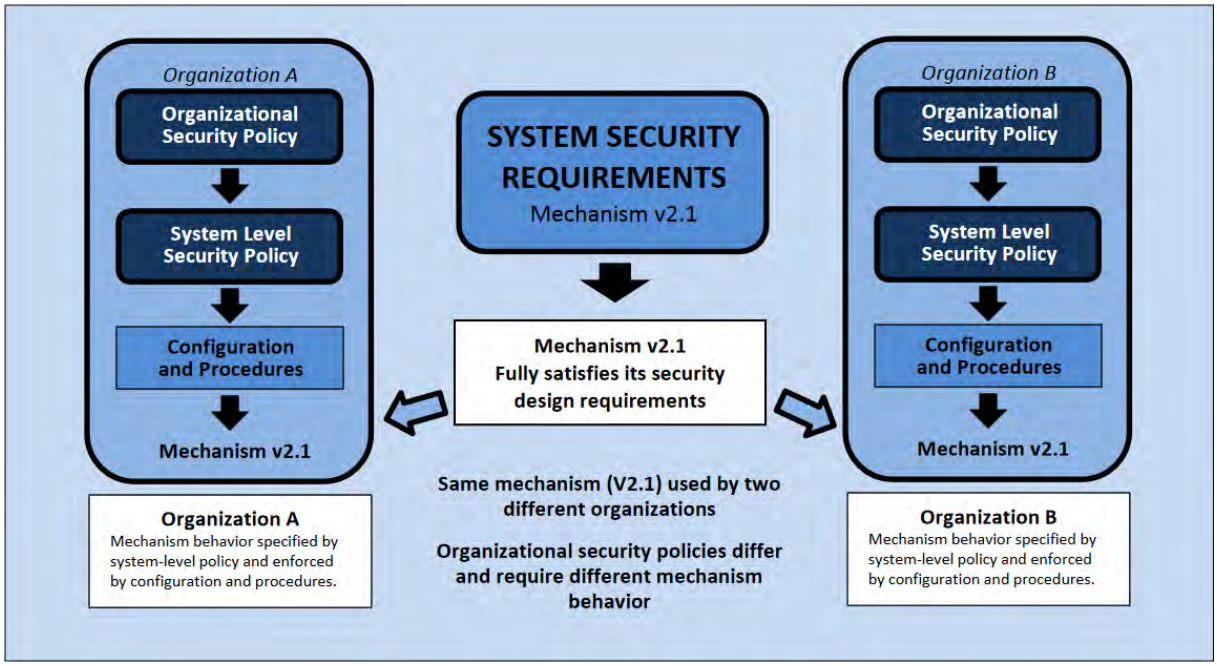

Figure 5.4: NIST SP 800-160 Vol. 1 figure G-8 [5]

5.4.1 NIST SP 800-160 Standard

NIST SP 800-160 Vol.1 standard [5] offers a perspective of security separating the security requirements of the security policies. As shown in Figure 5.4 taken from the Annex G of the standard, the security requirement describes the set of properties that a mechanism must satisfy. The requirements are independent from any organization and are described for a general description of a security need. As such mechanisms satisfying those requirements aim to be general and domain independent.

At the organization level, two levels of policies are described: the organizational security policy and the system-level security policy. In this case the system-level policy can be described as a refinement of the organizational policy applied to the system being developed. Finally, the mechanisms developed to satisfy the security requirements are configured and adapted to the procedures of the organization.

This security model explicitly separates the domain-independent and domain-specific aspects of security. The domain-independent layer is concerned with addressing generic security problems and identifying broadly applicable solutions. These problems and their corresponding solutions are intentionally abstract and system-agnostic, designed to be applicable across a wide range of systems and contexts. The goal is to define requirements and associated mechanisms in a reusable manner, ensuring that they can be adapted and configured for various domain-specific scenarios without being tied to the specifics of a particular system architecture or implementation. In contrast, the domain-specific aspect focuses on addressing the particular needs of a specific application domain. In this context, we first define the *domain policy*, which describes a specific requirement tailored to the

domain's operational context.

5.4.2 Security Objectives, Requirements, Policies and Mechanisms

In this section, we present our interpretation and extension of the idea presented by NIST in the form of a metamodel (see Figure 5.5). This metamodel maintains a clear separation between domain-independent requirements and mechanisms, and domain-specific policies and mechanisms. To facilitate the traceability and formal reasoning of security concerns, the concept of *Mechanism Capability* is introduced. It serves as an intermediate abstraction that captures the essential behavior of the constraint required to fulfill a given requirement, which is subsequently implemented by a concrete *Mechanism*. Furthermore, the notion of *Security Objective* is introduced as a high-level concept that supersede the domain-independent and domain-specific perspectives, allowing for a high-level articulation of security goals that guide the selection and configuration of appropriate requirements, policies, and mechanisms.

We provide a detailed description of the various concepts defined within both the conceptual and implementation metamodels. This includes an overview of the core elements that constitute these models, as well as the relationships that define their interactions. By systematically analyzing these elements and their interrelations, we establish a structured framework that ensures consistency, reusability, and traceability from abstract SecurityRequirement to their concrete enforcement in system architectures. The example presented will take place in the context of the web application example presented in Section 2.8.1

- *SecurityObjective*: A SecurityObjective is a high-level rule describing desirable properties of the system. For example "integrity of the origin of the information". This objective can be used both in the domain independent and the domain specific to describe SecurityRequirement and DomainSpecificPolicy that aim to achieve the objective. Additionally these objectives can be used in any implementation of the others concepts (component-based, object, events...)

- *SecurityRequirement*: A SecurityRequirement is a statement representing a general security need. We follow the recommendations outlined in the NIST SP 800-160 guidelines [5], which state that *"requirements specify the capability and behavior that a security mechanism is able to provide"*. This implies that a requirement defines a general property or capability that a system must possess, and the associated domain-independent mechanism is designed to fulfill this requirement. For example a SecurityRequirement can be instantiated as, the source of the message must be verifiable as the component sending the message. This statement does not represent a specific system.

- *DomainIndependentMechanismCapability*: The DomainIndependentMechanismCapability allow to describe the capabilities that a mechanism should offer in order to fulfill the SecurityRequirement. For example, (1) ensure the source of the message

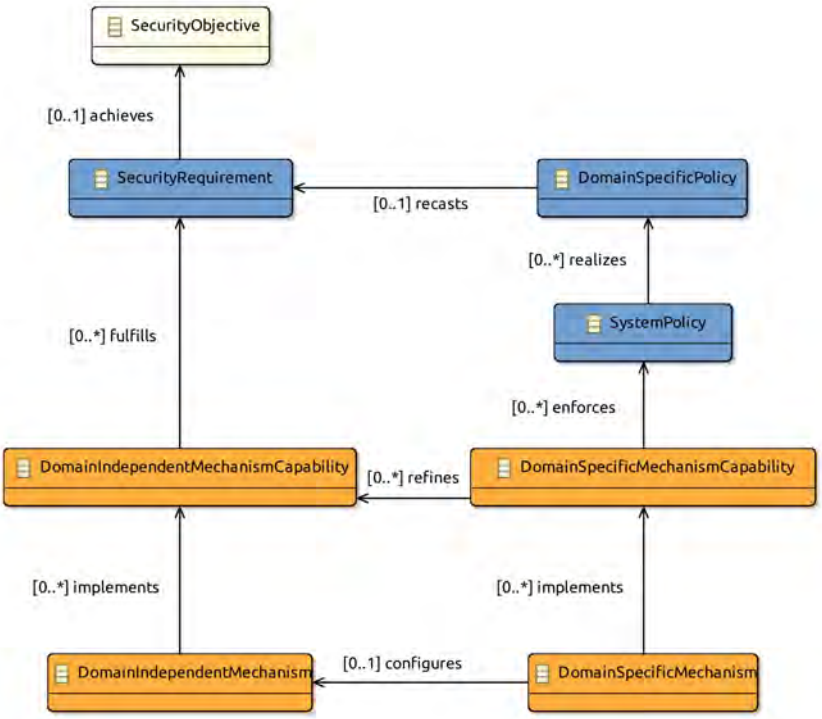

Figure 5.5: Security Conceptual Metamodel

is correctly initialized, and (2) ensure the source of the message cannot be changed after its initialization (before, during and after transit).

- *DomainIndependentMechanism*: The DomainIndependentMechanism allow to constrain the behavior of the systems as a concrete implementation of the DomainIndependentMechanismCapability. Any mechanism that implements the DomainIndependentMechanismCapability can be used to ensure by transitivity the SecurityRequirement. For example, a DomainIndependentMechanism can be instantiated as follows: the TLS protocol can ensure the authenticity of a message by implementing both capabilities (1) and (2), using digital certificates during the handshake.

- *DomainSpecificPolicy*: From NIST SP 800-160, "Security policy specifies the particular aspects that a security mechanism must enforce to achieve organizational objectives." The DomainSpecificPolicy describes a particular security need of the domain. For example a DomainSpecificPolicy can be instantiated as, web browser requests sender is correctly identified.

- *SystemPolicy*: SystemPolicy represents the desired properties of the under-development system. It contributes in the realization of the DomainSpecificPolicy. For example a SystemPolicy can be instantiated as, the source of the web browser request sent to the webserver is the original sender.

- *DomainSpecificMechanismCapability*: DomainSpecificMechanismCapability refines the DomainIndependentMechanismCapability enforcing the SystemPolicy. This refinement can take the form of configuration or methods to adapt the DomainIndependentMechanismCapability to the under-development system. For example two DomainIndependentMechanismCapability can be instantiated as, (1) the source of the web browser request sent to the webserver is correctly initialized to the appropriate web browser, and (2) the source of the web browser request sent to the web server cannot be modified after its initialization (before, during, and after transit).

- *DomainSpecificMechanism*: DomainSpecificMechanism configure the DomainIndependentMechanism implementing the DomainSpecificMechanismCapability. This implementation can take the form of configuration or methods to adapt the DomainIndependentMechanism to the under-development system. For example, a TLS communication channel with authentication must be opened between the web browser and the webserver.

In this approach, the S_Req presented for the case study in Section 2.8 would be represented as DomainSpecificPolicy as they represent security properties applied to the domain of the respective domain.

In this metamodel, the classes outlined in blue represent security properties (i.e., they extend the *Property* class of the *Property* sub-package) that can be described as predicates, essentially defining conditions that must hold true or false for the system to satisfy specific security requirements. These predicates serve as yes/no questions, such as verifying whether a certain access control policy is enforced or ensuring the confidentiality of data. On the other hand, the classes outlined in orange represent constraints (i.e., they extend the *Constraint* class of the *Property* sub-package) that actively modify the actual behavior of the system. These constraints impose restrictions or define specific operational rules that guide the system's execution, such as limiting resource access based on roles or enforcing timing constraints for critical operations. Together, these classes form a comprehensive framework for integrating security and functional considerations into the system's design and analysis.

The relations between all these concepts as depicted in the metamodel can be described as follows:

- *Achieves*: The *Achieves* relation establishes a direct connection between the objectives of the system and the SecurityRequirement. This relation signifies that an objective is effectively a conjunction of a set of requirements. In this context, achieving the objective necessitates that all associated requirements are met. This relationship highlights the critical interdependence between high-level goals and the granular specifications necessary to realize those goals, ensuring traceability and coherence across system design. For example, the SecurityRequirement "the source of the message must be verifiable as the component sending the message" achieves the SecurityObjective "Integrity of the origin of the information." In this example, the

SecurityRequirement alone is not sufficient to fully achieve the objective; however, it constitutes a part of the overall solution necessary for its achievement.

- *Fulfills*: The *Fulfills* relation maintains a similar multiplicity (N to 1 relation) to the *Achieves* relation, operating at the level of DomainIndependentMechanismCapability. For a SecurityRequirement to be deemed fulfilled, all related DomainIndependentMechanismCapability must be enforced. This relation underscores the essential role of DomainIndependentMechanismCapability as foundational building blocks in satisfying higher-level system requirements, ensuring a robust and comprehensive approach to requirement validation. For example, the conjunction of both DomainIndependentMechanismCapability (1) ensure the source of the message is correctly initialized, and (2) ensure the source of the message cannot be changed after its initialization (before, during and after transit) fulfill the Requirement that states that "the source of the message must be verifiable as the component sending the message." In this example, we can intuitively establish that both DomainIndependentMechanismCapability are necessary for fulfilling the SecurityRequirement; however, we have not formally proven that they are sufficient to guarantee its complete satisfaction. As such while being essential to the achievement of the SecurityRequirement the DomainIndependentMechanismCapability put as example do not guarantee that the SecurityRequirement has be achieved.

- *Recasts*: The *Recasts* relation establishes a connection between a SecurityRequirement and a DomainSpecificPolicy, indicating that the DomainSpecificPolicy is a domain-specific transformation of a more general security requirement. This relation serves as a bridge between domain-independent and domain-specific properties. For instance, the DomainSpecificPolicy "web browser request sender is correctly identified" recasts the generic SecurityRequirement "the source of the message must be verifiable as the component sending the message" within the context of the metering domain.

- *Implements*: The *Implements* relation defines the connection between the DomainIndependentMechanismCapability and their corresponding DomainIndependentMechanism. Unlike other relations, this one is exclusive, signifying that each mechanism in the set uniquely implements the capability it represents. This exclusivity ensures that each capability is implemented by a distinct mechanism, avoiding redundancy and fostering reuse by encouraging the design of diverse and alternative mechanisms. This approach provides flexibility, allowing system designers to choose the most suitable mechanism for a specific context or requirement, thereby enhancing adaptability and optimization in the system's design and deployment. For example, the TLS protocol [144] is one possible implementation of the DomainIndependentMechanismCapability "ensure the source of the message is correctly initialized." Additionally, it can also implement the DomainIndependentMechanismCapability "ensure the source of the message cannot be changed after its initialization (before, during, and after transit)," thereby implementing multiple capabilities within the

same mechanism.

- *Enforces*: The *Enforces* relation mirrors the purpose of the *Fulfill* relation but operates between the SystemPolicy and DomainSpecificMechanismCapability. This relation ensures that the enforcement of SystemPolicy is directly tied to the specific capabilities defined for the domain, aligning policy implementation with domain-specific needs and limitations. For example, the SystemPolicy"the source of the web browser request sent to the webserver is the original sender." in the context of the web application system enforces the DomainSpecificPolicy"the web browser is correctly identified"

- *Refines*: The *Refines* relation represents the process of tailoring DomainSpecificMechanismCapability to specific system requirements by applying DomainIndependentMechanismCapability in a manner that aligns with the system policy. This refinement bridges the gap between abstract capabilities and concrete implementations, ensuring that the application of DomainIndependentMechanismCapability is both contextually relevant and policy-compliant. This relation is critical for adapting general-purpose capabilities to address specific challenges within a given system. For example the DomainIndependentMechanismCapability"ensure the source of the message is correctly initialized" can be refined in the web application domain as "the source of the web browser request sent to the webserver is correctly initialized to the appropriate web browser".

- *Configures*: The *Configures* relation establishes the link between DomainIndependentMechanism and DomainSpecificMechanismCapability. It describes the process of configuring a DomainIndependentMechanism to suit the unique characteristics of a specific system. This configuration may involve the application of the DomainIndependentMechanism to specific system components, alterations to operational procedures, or adjustments to parameters. By capturing the configurability of mechanisms, this relation ensures that general-purpose solutions can be effectively adapted to diverse operational contexts, thereby enhancing system flexibility and resilience. For example the configuration of the TLS protocol to the web application domain as "a TLS communication channel with authentication must be opened between the web browser and the webserver".

5.4.3 Example of our Metamodel Instantiation

Example 5.4.1 *Confidentiality is defined as a category (i.e.,* PropertyCategory*) within the CIAAA library (i.e.,* PropertyCategoryLibrary*), and MessageConfidentiality (the property selected for illustration) is defined as property (i.e.,* ConnectorProperty*) as part of this Confidentiality category. We also define* restrictivePayload *as a category (i.e.,* ConstraintCategory*) within the access control library (i.e.,* ConstraintCategoryLibrary*). The* restrictiveSetSource *(the constraint selected for illustration) is defined as a constraint (i.e.,* ConnectorConstraint*).*

In this work we will describe the security objective using the CIAAA. Table 5.1 shows a few examples of SecurityObjective, SecurityRequirement, DomainIndependentMechanismCapability and DomainIndependentMechanism. While Table 5.2 shows a list of domain specific security elements applied to the web application presented in Section 2.8.1. Similarly, Table 5.3 shows the instantiation of the security elements applied to the smart meter gateway presented in Section 2.8.2. This case study will be presented in more detail in Chapter 8 but it is used here to illustrate that domain independent security elements can be applied to multiple domains.

Table 5.1: Example of Security Objectives, Requirements, and Mechanisms based on CIAAA

SecurityObjective	**SecurityRequirement**	**DomainIndependentMechanismCapability**	**DomainIndependentMechanism**
SO1. Authenticity of the source of information	SR1. The source of the message must be verifiable as the component sending the message	DIM1. Ensure the source of the message is correctly initialized	TLS Digital Certificate
		DIM2. Ensure the source of the message cannot be modified in transit	TLS Digital Certificate
SO2. Confidentiality of the information	SR2. The payload of the message must only be readable by the intended destination component	DIM3. Ensure the payload can only be read by the destination component	TLS Encrypt
	SR5. No sensitive data is transmitted in the clear, internally or externally.	DIM5. Ensure all sensitive data is encrypted before being transmitted	TLS Encrypt
SO3. Integrity of the information	SR3. The payload of the message must not be changed in transit	DIM3. Ensure the payload of the message cannot be changed in transit	TLS MAC
SO1	SR4. All connections must be authenticated	DIM4. Ensure all internal and external connections go through an appropriate and adequate form of authentication	TLS Digital Certificate
SO2	SR5. No sensitive data is transmitted in the clear, internally or externally.	DIM5. Ensure all sensitive data is encrypted before being transmitted	TLS Encrypt

Table 5.2: Domain-Specific Policies, System Policies, and Domain specific Mechanisms in the web application system context

DomainSpecificPolicy	**SystemPolicy**	**DomainSpecificMechanismCapability**	**DomainSpecificMechanism**
DP1. The web browser is correctly identified and authenticated	SP1. The source of the web browser request sent to the webserver is the original sender	DSM1.1. Ensure the source of the web browser request sent to the webserver is correctly initialized to the appropriate web browser	TLS Digital certificate for web application
		DSM1.2. Ensure the source of the web browser request sent to the web server cannot be modified in transit	TLS Digital certificate for web application
DP2. All connections to resources containing critical information must be authenticated	SP2. All connections to the webserver have been authenticated	DSM2. Ensure all connections to the webserver go through an appropriate and adequate form of authentication	TLS Digital certificate for web application
DP3. Authentication credentials do not traverse the wire in clear text form	SP3. Authentication credentials of the webserver's users do not traverse the wire in clear text form	DSM3. All authentication credentials sent by the webserver and its users are encrypted before being sent	TLS Encryption for web application

Table 5.3: Domain-Specific Policies, System Policies, and Domain specific Mechanisms in the smart-metering system context

DomainSpecificPolicy	**SystemPolicy**	**DomainSpecificMechanismCapability**	**DomainSpecificMechanism**
DP4. The sender of the metering data is correctly identified and authenticated	SP4. The source of metering data sent to the gateway is the original sender	DSM4.1. Ensure the source of the metering data sent to the gateway is correctly initialized to the appropriate sensors	TLS Digital Certificate for metering
		DSM4.2. Ensure the source of the metering data sent to the gateway cannot be modified in transit	TLS Digital Certificate for metering
DP5. The metering data is only readable by the intended destination	SP5. The metering data can only be read by the gateway	DSM5. Ensure the metering data sent by the sensors to the gateway can only be read by the gateway	TLS Encrypt for metering
DP6. The metering data doesn't change once produced	SP6. The metering data produced by the sensors doesn't change once produced	DSM6. Ensure the metering data cannot be changed once produced	TLS MAC for metering

Authenticity

According to the ISO/IEC 27000:2018 standard [145], *authenticity* denotes the property that an entity is what it claims to be. In the context of message passing communications within software architectures, message authenticity ensures that the source written in the message is the actual sender. If a message m has a component $c1$ as its source then it was sent by $c1$.

Confidentiality

According to the ISO/IEC 27000:2018 standard [145], *confidentiality* ensures that information is not made available or disclosed to unauthorized individuals, entities, or processes. In the context of message passing communications within software architectures, payload confidentiality allows a transmitted message to be received by other components, but only specific components can get the actual content of the payload.

Integrity

According to the ISO/IEC 27000:2018 standard [145], *integrity* ensures that transmitted information is accurate and complete. In the context of message passing communications within software architectures, message integrity ensures that the content of the message received is the same as when it was sent. If a message m, sent by the component c, has a payload p then c originally sent m with the same payload p.

Availability and Accountability

According to the ISO/IEC 27000:2018 standard [145], *availability* denotes the property of being accessible and usable on demand by an authorized entity, while *accountability* denotes the property that ensures that the actions of an entity can be traced uniquely to the entity. These two security objectives complete the CIAAA pentagon, however they will not be explored in this thesis due to time constraint and technical difficulties (See Section 8.4.4.

According to the ISO/IEC 27000:2018 standard [145], *confidentiality* ensures that information is not made available or disclosed to unauthorized individuals, entities, or processes. In the context of message passing communications within software architectures, payload confidentiality allows a transmitted message to be received by other components, but only specific components can get the actual content of the payload.

Authenticity Example in Web Application Context

In this section, we take the example of authenticity and apply it in the web application case study to illustrate the examples shown previously.

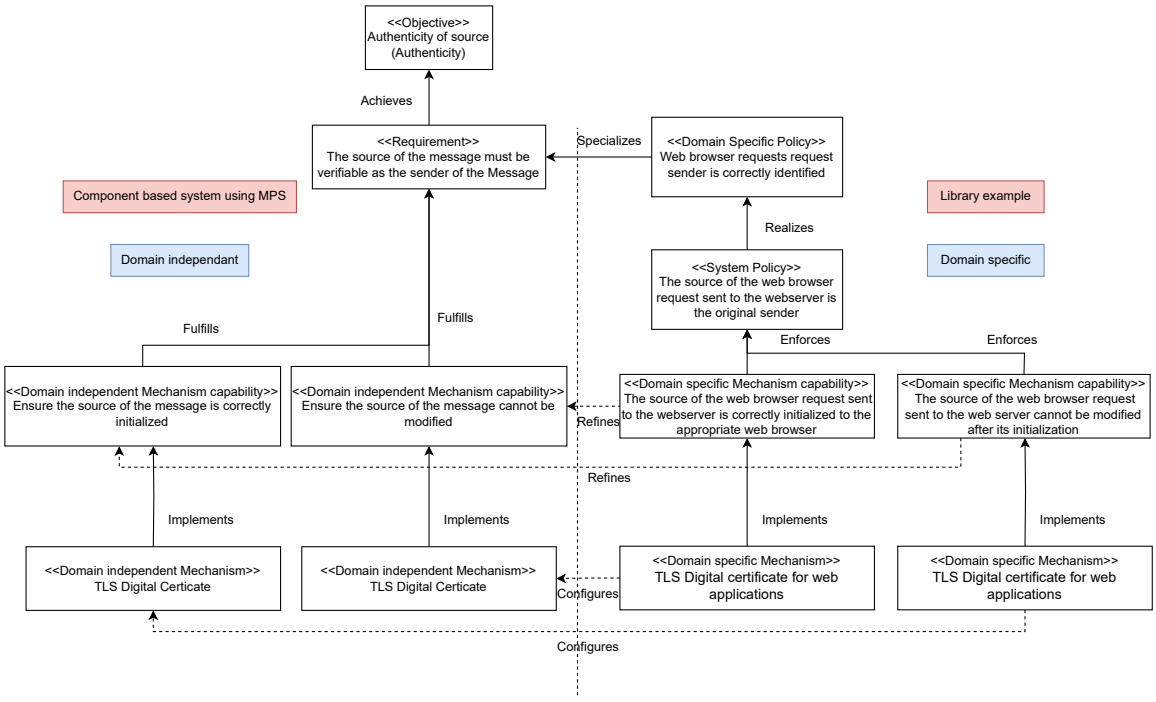

Figure 5.6: Instantiation of NIST SP 800-160 standard in the Web application case study with authenticity objective

In Figure 5.6, we present an instantiation of our interpretation of the relationships between requirements, mechanisms, policies, and objectives. This example illustrates the

application of the authenticity objective (Origin integrity) to a web application scenario within a component-based system, utilizing MPS (Message Passing System). In this example we focus on the Web application case study (See Section 2.8.1. More specifically, our goal is to ensure the authenticity of the source for messages transmitted from the web browser to the web server. This guarantees that the data originates from the web browser it claims to be from, thereby ensuring the reliability of the collected requests.

Our primary focus is the *authenticity objective*, specifically the authenticity of the message source. This objective is intentionally generic, allowing it to be applied across a wide range of systems that involve machine-to-machine communication.

Based on this objective, we define a domain-independent requirement aimed at achieving it. This requirement can be summarized as: **SecurityRequirement: The source of the message must be verifiable as the sender of the message**.

This domain-independent requirement is fulfilled by the corresponding DomainIndependentMechanismCapability, which can be described as: **DomainIndependentMechanismCapability: Ensure the source of the message cannot be modified**. One implementation of this capability is the TLS protocol through the use of its digital certificates.

On the domain-specific aspect of our design, exemplified by the web application case study, the domain policy aimed at achieving the objective is defined as *"The source of the web browser request sent to the webserver is the original sender"*. This domain policy can be realized through various approaches. In this particular example, we adopt one specific realization: *"The source of the web browser request sent to the webserver is correctly initialized to the appropriate web browser"*. This system policy focuses on verifying source authenticity at the webserver level. However, it is important to note that additional system policies can be incorporated at different points within the system to comprehensively achieve the domain policy. To enforce this system policy, we refine the domain-independent mechanism with the objective of ensuring that *"The source of the web browser request sent to the webserver is correctly initialized to the appropriate web browser."* This refinement involves configuring the domain-independent mechanism to align with and support the specified system policy, thereby achieving the desired behavioral enforcement within the system. Finally, the TLS protocol can be implemented into the final system to enforce the policy.

Chapter 6

Formal Analysis of System Architecture and Security

Contents

6.1 Introduction

In this chapter, we introduce the formal representation of the model described in Chapter 5, including component-based software architecture, security objectives and security policies. This formalization is based on Event-B, a proof-based formal method, to rigorously define the previously introduced concepts for their formal analysis. By leveraging Event-B's proof mechanisms, we ensure the correctness and consistency of the model through formal verification, providing a mathematically sound interpretation of its properties and constraints. This chapter aims to answer the formal aspects of our objectives namely **RO 1.2**, **RO 2.2** and **RO 3.3**.

The rest of this chapter is organized as follows. Section 6.2 introduces a technology-independent specification language to describe the behavior of CBS, enabling the specification of properties that can later be formalized. Section 6.3 then presents the formalization

of CBS architectures using Event-B, addressing both structural and behavioral aspects. Section 6.4 illustrates the interpretation of properties and constraints in Event-B through functional examples, such as FIFO. Section 6.5 demonstrates the creation of a system instance using the refinement process inherent to Event-B. Finally, Section 6.6 presents our Event-B-based interpretation of security properties and constraints using the concepts introduced in the previous chapter, while Section 6.7 provides concrete instantiations of those concepts.

6.2 Logical Specification

In this section, we present the precise definition of a set of representative properties for the communication between components using first-order logic and modal logic [146] as a formalism that is abstract and technology-independent. This formalism used in our research team combines modal logic and first order logic in order to represent temporal properties of CBS. This provides a more generic and understandable approach with mapping support to different existing property specification languages used in modeling and software development.

6.2.1 Abstract System Computing Model

A distributed software system is modeled by a set of components and a set of connectors. Components communicate and synchronize by sending and receiving messages through existing connectors. A computation program encodes the local actions that components may perform. The actions of the program include modifying local variables, sending messages, and receiving messages to/from each component in the corresponding system architecture. In the context of engineering software systems, we define the following domain to capture the notion of both legitimate and illegitimate message providers (e.g., *send*) and message consumers (e.g., *receive*) of the system messages. These concepts are defined on top of the basic communication primitives (e.g., *inject* and *intercept*):

Sets

- $\mathcal{C}$ is the set of components
- $\mathcal{CA}$ is the set of communication artifacts
  - $\mathcal{M}$ is the set of messages is extended with the following predicates:
    - has_src(m, s) indicates that the source of $m \in \mathcal{M}$ is $s \in \mathcal{C}$, where s may not be the origin of m
    - has_rcv(m, r) indicates that the recipient of $m \in \mathcal{M}$ is $r \in \mathcal{C}$, where r may not be the intended receiver of m
    - has_pld(m, d) indicates that $m \in \mathcal{M}$ contains a payload $d \in \mathcal{D}$
  - $\mathcal{D}$ is the set of payloads

  - $\mathcal{I}$ is the set of invocations
  - $\mathcal{R}$ is the set of arguments
- $\mathcal{T}$ is the set of data types
- $\mathcal{A}$ is the set of actions
  - *inject*(c, a) denotes that component c $\in \mathcal{C}$ adds communication artefact a $\in \mathcal{CA}$ into the system
  - *intercept*(c, a) denotes that component c $\in \mathcal{C}$ gets communication artefact a $\in \mathcal{CA}$ from the system
  - get_src(m, s) denotes that component $c \in \mathcal{C}$ gets the declared source $s \in \mathcal{C}$ from message $m \in \mathcal{M}$, where s is not necessarily the true sender

Modalities

- $\mathbb{E}_c(a)$ is a predicate indicating that action $a \in \mathcal{A}$ is $\mathbb{E}$**nabled** for component $c \in \mathcal{C}$
- $\mathbb{H}_c(a)$ is a predicate indicating that action $a \in \mathcal{A}$ $\mathbb{H}$**appens** for component $c \in \mathcal{C}$
- *term1* $<$ *term2* : is predicate that indicating that all possible sequences of actions where the term *term2* is true also contain a true term *term1* that **happens before** *term2*

6.2.2 Property Specification

We specify some representative properties for the message passing communication style. These examples illustrate a subset of the possible properties that can be specified. Additional properties can be defined following the same underlying logical principles.

- Once the server $c1$ receives a message, it must already have been sent by a certain client $c2$. $\forall\, c1, c2 \in \mathcal{C}, d \in \mathcal{D}, typ \in \mathcal{T} \cdot \mathbb{H}_{c_2}(\text{inject}(d, typ)) \;<\; \mathbb{H}_{c_1}(\text{intercept}(d, typ))$
- Messages sent from the client $c1$ to the server $c2$ reach the server $c2$ in the same order as they were sent from $c1$ if the connector is FIFO.
  $\forall\, c1, c2 \in \mathcal{C}, d1, d2 \in \mathcal{D}, typ1, typ2 \in \mathcal{T} \cdot \mathbb{H}_{c_1}(\text{inject}(d1, typ1)) \;<\; \mathbb{H}_{c_1}(\text{inject}(d2, typ2))$
  $\implies \mathbb{H}_{c_2}(\text{intercept}(d1, typ1)) \;<\; \mathbb{H}_{c_1}(\text{intercept}(d2, typ2))$

At this level of specification, we are able to: (1) specify the behavioral semantics of a set of communication styles, and (2) provide an intermediate high-level representation (using first-order logic and modal logic) which is more understandable by system architects and that can be interpreted in appropriate formal tooled languages. This satisfies ***RO 1.2****.*

6.3 Formalization of the System Architecture in Event-B

In this section, we present the formalization of our software architecture metamodel using Event-B. The metamodel incorporates the concepts of a component–port–connector architecture and involves new concepts to capture the behavioral aspects of a specific message-passing communication model. We then provide the specification and verification of a representative set of structural and behavioral properties.

6.3.1 Interpretation of our Metamodel

A software architecture metamodel as described in Section 5.2 is mapped to an Event-B model as follows. We use the separation of the static (context/logical) and the dynamic (machine/scenario) views present in both Event-B and the metamodel. Therefore, elements in the contexts represent the structural aspects of the metamodel and the elements in the machines represent the behavioural aspects. The constructs of Event-B such as carrier sets and axioms will be used to specify the structural aspect. Events and variables will be used to define the behavioral aspects. Finally, invariants and theorems will be used to prove the properties of the model at each level of abstraction. These properties will be used as the basis to validate the requirements of the concrete system, such as illustrated in Section 6.5.

Figure 6.1 represents the general refinement tree of the Event-B model described in detail below. Each Machine and Context are annotated with an id that will be used to link all the following listings to their corresponding level of refinement in this figure. Recall that our goal is to provide reusable connectors in the form of reusable formal model libraries. *A complete presentation of our Event-B metamodel, the full specification of the other constructs (e.g., communication primitives, etc.), and the reusable formal model libraries and properties are available online via* `https://gitlab.com/semcoproject/toc`.

6.3.1.1 Structural Model

To specify the software system architecture in Event-B using the component-port-connector fashion, we first define the following domain to introduce the meaning of the principal concepts of the metamodel. These will be the sets ($\mathcal{C}, \mathcal{P}, \mathcal{O}$, ...) described in Section 6.2.1.

We used a similar modeling strategy as Mammar et al. [147]. The mapping of these structural elements to Event-B with respect to the semantics given in the definition of the metamodel, as shown in figure 5.2, is straightforward. Carrier sets are used to capture model elements' type (i.e., COMPONENT to represent the set of components $\mathcal{C}$). The architecture model of a system is then described through constant subsets of the corresponding carrier set. Before two (or more) components can interact, we assume that a connector must be present between them. A component is connected to a connector

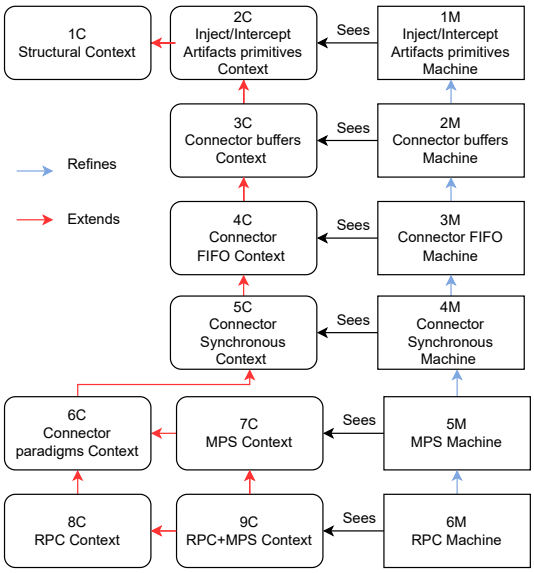

Figure 6.1: Refinement tree of the Event-B model for CBS

through a number of ports. This is represented in the model as relations between the components, the ports, and connectors (Uses, Connects). The multiplicity of the associations in the metamodel (see Figure 5.2) defines the type of relations. For example, *Uses* has a full function and *Connects* has a simple relation as the multiplicities of the associations are respectively $1..n$ and $n..n$ in the metamodel. These statements are specified using axioms, as depicted in Listing 6.1.

6.3.1.2 Behavioural Model

To model the behaviour of the system, we take advantage of the event-based nature of Event-B and the logical specifications described previously. An execution of a system is a sequence of steps (instants), where a step is determined by the execution of an event such as Inject or Intercept. To specify the communication paradigms under the connector, we extend the previous domain defining a new context and machine. Mainly, we introduce two basic communication primitives *inject* and *intercept* on top of which we capture the notions message producers (e.g., *send* in the MPS context) and consumers (e.g., *receive*) of the system messages. As with the structural model, we use the abstract specification of Section 6.2.1 but we focus on the behavioral elements ($\mathcal{CA}, \mathcal{M}, \mathcal{D}$...).

To define these new concepts, we use carrier sets and the set constructs of Event-B. For instance, COMMUNICATION_ARTIFACT type is defined as a carrier set to represent the set of communication artifact $\mathcal{CA}$. In the following, we present the modeling of the proposed domain. Furthermore, we introduce some characteristics to describe the connectors.

```
context Context1Component

sets COMPONENT // Set of components
     PORT // Set of ports
     CONNECTOR // Set of connectors
     PORTKIND // Set of the types of ports

constants Component // Finite set of system components
          Port // Finite set of system ports
          Connector // Finite set of system connectors
          Uses // Connection of the port to the components
          Connects // Connection of the connectors to the ports
          ...

axioms
  // Defines constants sets types as subsets of the carrier sets
  @axmComp1 Component ⊆ COMPONENT
  @axmPort1 Port ⊆ PORT
  @axmConn1 Connector ⊆ CONNECTOR
  // Defines constants relations
  @axmCompPort1 Uses ∈ Port → Component
  @axmConnCon1 Connects ∈ Port ↔ Connector
  @axmPortKind PortKinds ∈ Port → PortKindsType
  ...
end
```

Listing 6.1: Context of the structural model (1C)

Inject/Intercept Primitives

We introduce in our model two events to describe the Inject/Intercept primitives, as depicted in Listing 6.2. Without loss of generality, since each port is attached to a unique component, in our modeling we use port instead of component in the Inject and Intercept modeling. For instance, the *Inject* event is defined using four parameters (communication artifact, producer, consumer, and connector represented by the parameters a, p, c, and $conn$, respectively), nine guards and five actions. As examples of guards, a is a Communication Artifact ($grd1$) that is not contained in the set $artifact$ ($grd9$) which implies that this artifact is a new one that has neither been injected nor intercepted before. We also describe that the producer port p and consumer port c are connected by the connector $conn$ ($grd7$ and $grd8$), and they are respectively an out and in port ($grd10$ and $grd11$). If there is a communication artifact, producer, consumer, and connector represented by the parameters a, p, c, and $conn$, respectively, where all the *Inject* event's guards are satisfied we can trigger the *Inject* event, executing its actions which adds a into the $artifactSent$ ($act1$). This results in the adding of the communication artifact into the system.

Similarly, in the *Intercept* event, the artifact is transferred from the $artifactSent$ set to the $artifactReceived$ set, indicating that it has been successfully received.

Buffering

From the metamodel, a connector connects components through ports and specifies a connector to support communication between components by allowing the exchange of

```
machine CBSMachine2Communication sees Context2Communication

variables artifactsSent artifactsReceived source destination connectorsUsed artifact receiver

invariants
  ...
  @ allArtifactsInv artifact = artifactInjected ∪ artifactIntercepted
  // No artifact already received are in the artifact sent
  @ artifactInjectedAndInterceptedDisjoint artifactsSent ∩ artifactsReceived = ∅
  // All the artifacts sent are sent between components that are connected by a connector
  @componentsCanInteract ∀ a · a ∈ artifact ⇒ {destination(a) ↦ connectorsUsed(a), source(a)
    ↦ connectorsUsed(a)} ⊆ Connects
  // The destination of the artifact is an input
  @destinationPortInTypeInv ∀ c · c ∈ ran(destination) ⇒ c ↦ PortIN ∈ PortKinds
  // The destination of the artifact is an output
  @sourcePortOUTTypeInv ∀ p · p ∈ ran(source) ⇒ p ↦ PortOUT ∈ PortKinds

events
  event Inject
    any a // artifact
        s // source
        d // destination
        conn // connector
    where
      @grd1 a ∈ COMMUNICATION_ARTIFACT // New artifact
      @grd4 s ∈ Port // source port of the artifact
      @grd5 d ∈ Port // destination port of the artifact
      @grd6 conn ∈ Connector // connector used to
      @grd7 s ↦ conn ∈ Connects // source is connected to the connector
      @grd8 d ↦ conn ∈ Connects // destination is connected to the connector
      @grd9 a ∉ artifact // New artifact only
      @grd10 PortKinds(s) = PortOUT // Check source port kind to be out
      @grd11 PortKinds(d) = PortIN // Check destination port to be in
    then
      // Set the relations of the newly sent artifact
      @act1 artifactInjected := artifactInjected ∪ {a}
      @act2 source(a) := s
      @act3 destination(a) := d
      @act4 artifact := artifact ∪ {a}
      @act5 connectorsUsed(a) := conn
  end

  event Intercept
    any a r
    where
      @grd1 a ∈ artifactsSent
      @grd2 r ∈ Port
    then
      // Add the artifact to the received artifact set
      @act1 artifactsReceived := artifactsReceived ∪ {a}
      // Remove the artifact to the sent artifact set
      @act2 artifactsSent := artifactsSent \ {a}
      // Set the receiver of the artifact
      @act3 receiver(a) := r
  end
end
```

Listing 6.2: Inject and intercept machine (1M)

communication artefacts. Here, we consider a buffering characteristic of the connector to support communication artefacts storage during the transit. The connector is modeled as a buffer of communication artefact associated with the two operations Inject and Intercept to support the high-level communication primitives (e.g., send and receive in the MPS context): (i) Inject to add a communication artefact in the corresponding buffer and (ii) Intercept to remove a communication artefact from the corresponding buffer. As depicted in Listing 6.3, we proceed with a refinement of the previous machine (*CBSMachine2Communication*). *artifactsSent* is redefined as multiple connector buffers associated with the corresponding connectors using a *gluing invariant* refinement (Listing 6.3). Events are also modified accordingly. In the *Intercept* event, the artifact is now removed from the corresponding buffer instead of *artifactsSent* set.

For the *Inject* event, several new guards are needed to make sure the buffer in which we want to inject the artifact is not full ($grd12$) and we also need to make sure that the artifact is not already in a buffer ($grd13$). Finally, the artifact is added into the relevant buffer ($act6$).

6.3.2 Communication Paradigms

In this section, we experiment with modeling of two well-known communication paradigms, namely message passing (MPS) and remote procedure call (RPC). Using the Event-B refinement facilities, we provide a reusable library of connectors to support these two communication paradigms, including different configurations associated with the previous connector characteristics. Note, however, that the refinement of the machine (behavior) is only vertical while some extensions of the context can be horizontal, as seen in Figure 6.1. For instance, it does not make sense to say that RPC is a refinement of MPS. Overall, the shape of the model only represents the implementation with the Event-B limitations and not the real semantics of a system.

The idea behind our formalisation is to consider that at the time of sending a message (resp. invocation), the message (resp. invocation), as a communication artifact, is created and placed in the buffer of a connector connecting the sender and receiver (resp. caller and called) components. Similarly, the message (resp. invocation) would be removed from the connector buffer at the time of message reception (resp. method execution).

Message Passing

In the message passing communication style (MPS), a connector is used for sending a message from a sender to a receiver. The message is simply transmitted without any acknowledgment. The behavior of the MPS connector is modeled in Listing 6.4 through a refinement of the previous machine. We add a new event *CreateMessage* to create a new communication artifact with attributes defining the MPS paradigm (e.g., Payload, DataType) and extend the Inject and Intercept events, accordingly. For instance, restrictions in the form of guards are added to the Inject to ensure that the injected artefact is a message (guard @messageParadigm) and the ports to be used by the con-

```
1  invariants
2    // Relation connecting all the connectors to their buffered artifacts
3    @buffersTypeInv buffer ∈ Connector → ℙ(artifact)
4    // All the buffers combined make the artifact sent
5    @gluingBufferInv (⋃ conn · conn ∈ Connector | buffer(conn)) = artifactsSent
6    // The buffers contain a finite amount of artifact (Necessary for the use of card)
7    @ finiteBufferInv ∀ conn · conn ∈ Connector ⇒ finite(buffer(conn))
8    // The artifacts sent can only be in a single buffer
9    @artifactInOneBufferOnlyInv ∀ a, b1, b2 · a ∈ artifact ∧ b1 ∈ dom(buffer) ∧ b2 ∈ dom(buffer)
       ∧ a ∈ buffer(b1) ∧ a ∈ buffer(b2) ⇒ b1=b2
10
11 events
12   event Inject refines Inject
13     any a // artifact
14           p // source
15           c // destination
16           conn // connector
17     where
18       ...
19       // Can¬inject a new artifact into a full connector
20       @grd12 card(buffer(conn)) < ConnectorsBufferSize(conn)
21       // The artifact is ¬in a connectr buffer
22       @grd13 a ∉ (⋃ conn1 · conn1 ∈ Connector | buffer(conn1))
23     then
24       ...
25       @act2 source(a) := p
26       @act3 destination(a) := c
27   end
28
29   event Intercept refines Intercept
30     any a r conn
31     where
32       ...
33       @grd2 conn ∈ Connector
34       @grd3 a ∈ buffer(conn)
35       @grd4 r ∈ Port
36     then
37       @act1 artifactsReceived := artifactsReceived ∪ {a}
38       @act2 buffer(conn) := buffer(conn) \ {a}
39       @act3 receiver(a) := r
40   end
41 end
```

Listing 6.3: Event-B model introducing buffers (2M)

nector support the DataType of the message (see guards @payloadDatatypeSource and @payloadDatatypeDestination).

Remote Procedure Call

In the typical remote procedure call (RPC) communication style, a connector is used for sending invocation (request) messages from a caller to a called and for receiving acknowledgment (reply) messages from a called to a caller. The behavior of the RPC connector is modeled using the same process as MPS. We add a new event *CreateInvocation* to create an invocation out of an artifact with attributes defining the RPC paradigm (e.g., Method, Argument, etc.). Finally, we add the relevant guards to the inject event to ensure that the injected artifact is an invocation and the corresponding method is present at the interface of receiving component before an invocation is done.

```
1  machine CBSMachine6MPS refines CBSMachine5ConnectorSync
2  sees Context7MPS
3
4  variables
5      artifactsPayloads  payloadsType message
6
7  invariants
8    ...
9    // paylaod set type invariant
10   @payloadsTypeInv payload ⊆ PAYLOAD
11   // message set typeinvariant
12   @messageTypeInv message ⊆ COMMUNICATION_ARTIFACT
13   // message is a finite set
14   @messageFiniteInv finite (message)
15   // artifactsPayloads relation type invariant
16   @artifactsPayloadsTypeInv artifactsPayloads ∈ message → payload
17   // Every messages have a payload
18   @messagesHavePayload ∀ m · m ∈ message ⇒ m ∈ dom(artifactsPayloads)
19   // The source and the destination of the message realize some data (used for WD proof)
20   @messagesSourceAndDestinationPortRealizeData ∀ m · m ∈ message ∧ m ∈ artifact ⇒ source(m)
       ∈ dom(Realizes) ∧ destination(m) ∈ dom(Realizes)
21   // Every messages is sent in a MPS connector
22   @messageSentInMPSConnectors ∀ m · m ∈ message ∧ m ∈ artifact ⇒ ConnectorParadigm(
       connectorsUsed(m)) = MPS
23   // payloadsType relation type invariant
24   @payloadsDataTypeInv payloadsType ∈ payload → DataType
25   // Every destination of a message has a is defined as an input
26   @destinationPortInTypeInv ∀ c, a · a ↦ c ∈ destination ∧ ConnectorParadigm(connectorsUsed(a)
       ) = MPS ⇒ Realizes(c) ↦ DataIN ∈ DataDataDirection
27   // Every source of a message has a is defined as an output
28   @sourcePortOUTTypeInv ∀ p, a · a ↦ p ∈ source ∧ ConnectorParadigm(connectorsUsed(a)) = MPS
       ⇒ Realizes(p) ↦ DataOUT ∈ DataDataDirection
29   // The source and destination of a message realize a data containing the datatype of the
       message
30   @MPSTypeCheckDatatypeInv ∀ m · m ∈ message ∧ m ∈ artifact ⇒ payloadsType(artifactsPayloads(
       m)) ∈ DatasDatatypes(Realizes(source(m))) ∧ payloadsType(artifactsPayloads(m)) ∈
       DatasDatatypes(Realizes(destination(m)))
31   // Every artifact in an MPs connector is a message
32   @OnlyMessagesInMPSConnectors ∀ a, conn · a ∈ artifact ∧ conn ∈ Connector ∧ connectorsUsed(a)
       = conn ∧ ConnectorParadigm(conn) = MPS ⇒ a ∈ message
33
34 events
35   event Inject extends Inject
```

```
36      where
37        // If the connector is an MPs connnector then the artifacts needs to be a message
38        @messageParadigm ConnectorParadigm(conn) = MPS ⇔ a ∈ message
39        // If the connector is an MPs connnector then the source and destination port need to
          realize a Data
40        @portsRealizeData ConnectorParadigm(conn) = MPS ⇔ (Realizes(p) ∈ Data ∧ Realizes(c) ∈
          Data)
41        // If the connector is an MPs connnector then the source port data needs to contain the
          dataype of the message
42        @payloadDatatypeSource ConnectorParadigm(conn) = MPS ⇒ payloadsType(artifactsPayloads(a
          )) ∈ DatasDatatypes(Realizes(p))
43        // If the connector is an MPs connnector then the destination port data needs to contain
          the dataype of the message
44        @payloadDatatypeDestination ConnectorParadigm(conn) = MPS ⇒ payloadsType(
          artifactsPayloads(a)) ∈ DatasDatatypes(Realizes(c))
45        // If the connector is an MPs connnector then the source port must be an output
46        @dataDestincationSource ConnectorParadigm(conn) = MPS ⇔ Realizes(p) ↦ DataOUT ∈
          DataDataDirection
47        // If the connector is an MPs connnector then the destination port must be an input
48        @dataDestincationDestination ConnectorParadigm(conn) = MPS ⇔ Realizes(c) ↦ DataIN ∈
          DataDataDirection
49    end
50
51    event Intercept extends Intercept
52    end
53    ...
54
55    event CreateMessage // Factory creating a message that can be send afterward in an MPS
        connector
56      any m // message
57          messagePayload // payload
58          payloadType // payload type
59      where
60        @grd1 m ∈ COMMUNICATION_ARTIFACT
61        @grd2 messagePayload ∈ PAYLOAD
62        @grd3 payloadType ∈ DataType
63        @grd4 m ∉ artifact
64        @grd5 m ∉ message
65      then
66        @act1 payload := payload ∪ {messagePayload}
67        @act2 artifactsPayloads (m) := messagePayload
68        @act3 payloadsType(messagePayload) := payloadType
69        @act4 message := message ∪ {m}
70    end
71  end
```

Listing 6.4: Refinement introducing MPS (5M)

6.3.3 System Property Specification and Verification

Through Event-B modelling, certain properties emerge from the basic properties of connectors and their characteristics, which were previously identified and verified.

Among the possible properties, we present three below. As introduced in Section 2.6.2, in Event-B the verification process consists of proving the proof obligations that have been automatically generated by the Rodin tool.

(a) "For two components $c1$ and $c2$ to interact, a connector *conn* must be present between them". To specify this property, we use the invariant shown in Listing 6.5. This

property ensures that components cannot interact without the presence of a connector between them. This constrains the way in which the structural elements that are required to enable communication between two components are defined. The proof of this property using Rodin generates three automatic proof obligations that are then discharged by the automatic provers.

```
1 @componentsCanInteract ∀ a · a ∈ artifact ⇒ {destination(a) ↦ connectorsUsed(a), source(a) ↦
      connectorsUsed(a)} ⊆ Connects
```

Listing 6.5: Invariant representation of property (a)

(b) "A communication artifact is either in transit or intercepted". Every intercepted communication artifact is not in any buffer anymore and vice versa. To model this property, we use the invariant depicted in Listing 6.6. This property ensures that the sets *artifactSent* (all the artifacts that are currently in transit) and *artifactReceived* (all the artifacts that have been intercepted) have no elements in common. The proof of this property using Rodin generates three automatic proof obligations.

```
1 @ artifactInjectedAndInterceptedDisjoint artifactsSent ∩ artifactsReceived = ∅ // No artifact
      already received are in the artifact sent
```

Listing 6.6: Invariant representation of property (b)

(c) "For two components $c1$ and $c2$ to interact using message passing, each must use a port that realizes the correct Data offering the same Datatype dt". To model this property, we define three invariants, as depicted in Listing 6.7. This property ensures that only well-formed message interactions are possible by constraining the structural properties of the components involved in such interactions. The proof of this property using Rodin generates ten automatic proof obligations.

```
1 @destinationPortInTypeInv ∀ c, a · a ↦ c ∈ destination ∧ ConnectorParadigm(connectorsUsed(a))
      = MPS ⇒ Realizes(c) ↦ DataIN ∈ DataDataDirection
2 @sourcePortOUTTypeInv ∀ p, a · a ↦ p ∈ source ∧ ConnectorParadigm(connectorsUsed(a)) = MPS
      ⇒ Realizes(p) ↦ DataOUT ∈ DataDataDirection
3 @MPSTypeCheckDatatypeInv ∀ m · m ∈ message ∧ m ∈ artifact ⇒ payloadsType(artifactsPayloads(m
      )) ∈ DatasDatatypes(Realizes(source(m))) ∧ payloadsType(artifactsPayloads(m)) ∈
      DatasDatatypes(Realizes(destination(m)))
```

Listing 6.7: Invariant representation of property (c)

6.4 Functional Properties and Constraints

In this section, we present our interpretation in Event-B of the functional properties and constraints presented in Section 5.3.2.

6.4.1 Constraint Specification in Event-B

Event-B is fundamentally structured around the notion of refinement. Within this paradigm, the incorporation of constraints into a model is typically achieved by augmenting events

with additional guards. To clearly distinguish between a generic architectural model and its specific instantiations, it becomes essential to introduce reusable abstractions that can be systematically instantiated during model refinement.

As Event-B is grounded in set theory, we explored two principal strategies to semantically differentiate architectural elements: *set membership* and *tag-based relations*.

- **Set Membership:** Elements are associated with specific sets that activate certain behaviors or constraints. For instance, if a connector c belongs to a set *FIFO*, a guard condition can be appended to relevant events to enforce the corresponding constraint:
$$\forall c \in FIFO \Rightarrow \text{condition}$$
- **Tag-Based Relation:** Constraints are modeled as explicit entities (tags), and a relation is established between architectural elements and their corresponding constraints. For example, a connector c can be associated with a tag *FIFO* through a relation captured in a set such as:
$$ConnectorConstraint = \{c_1 \mapsto FIFO, \ldots\}$$
A guard can then be formulated to activate only when a specific relation is present:
$$\forall c \in Connector \land (c \mapsto FIFO) \in ConnectorConstraint \Rightarrow \text{condition}$$

Both approaches offer equivalent expressiveness for constraint specification, but each comes with distinct trade-offs. In this work, we adopt the tag-based relation approach due to its enhanced flexibility and its ability to reduce the complexity of the static context. Whereas the set membership approach requires a dedicated set for each constraint, the tag-based approach centralizes all associations in a single relation, thereby simplifying model initialization and management.

The following sections demonstrate the implementation of selected constraints using this relational tagging strategy, leveraging the separation of concerns between domain-independent abstractions and their domain-specific refinements that represent concrete system instances.

6.4.2 Formal Interpretation of Functional Properties and Constraints

In Event-B, **properties** of the system are formally specified using *invariants*, while **constraints** on the system behavior are interpreted through *guards* on events. This distinction supports a clear separation between declarative requirements and operational conditions.

The invariants define desirable system-wide conditions that must always hold during execution. Constraints, on the other hand, restrict the permissible transitions by conditioning event execution on certain guards.

Formally, if some constraints C are enforced by the guards of an event, the associated assumptions G_C become part of the proof obligations. If the constraints are correctly designed to ensure the property P, the invariant I_P representing the property should be provable under the assumptions induced by the guards.

Thus, the following logical implication must hold:

$$G_C \Rightarrow I_P$$

Where:

- G_C represents the assumptions introduced by the guard enforcing the constraint,
- I_P denotes the invariant expressing the property.

This formal approach ensures that constraints implemented via guards contribute to the overall system correctness by supporting the proof of properties encoded as invariants. The methodology is particularly effective when constraints are reusable across models and can be composed to imply more complex properties.

FIFO

In this section, we detail the formal interpretation of the *FIFO* (First-In, First-Out) property, as introduced in Section 5.3.2. This behavioral property is captured through a functional constraint, which is formally represented by additional guards on the *Intercept* event, as illustrated in Listing 6.8. These guards enforce a specific ordering on message processing to reflect the FIFO semantics. The functional constraint is introduced during the refinement of the *Buffer* machine.

To enable the constraint in a reusable and modular fashion, we associate it with the *FIFO* tag within the *ConnectorFIFOType* tagging relation. This relation allows the constraint to be selectively activated on any connector that is explicitly marked as adhering to FIFO semantics. This mechanism provides both flexibility and scalability by decoupling the constraint definition from specific connector instances.

Formally, the guard enabling the constraint can be expressed as:

$$(c \mapsto \text{FIFO}) \in \mathit{ConnectorFIFOType} \Rightarrow \text{FIFO\_Guard}(c)$$

where *FIFO_Guard*(c) represents the condition enforcing the FIFO order on connector c.

This approach ensures that the FIFO constraint is only enforced for the relevant connectors while maintaining the reusability of the architectural elements.

Synchronicity

Similarly, we represent the *Synchronicity* property and its corresponding functional constraint within the refinement of the *FIFO* machine. This follows the same modeling strategy applied in the interpretation of the *FIFO* property.

```
1  invariants
2    // Relation of every artifact to their order through natural number id
3    @ bufferArtifactListTypeInv  bufferArtifactList ∈ Connector → ℙ(ℕ × artifact)
4    // ID of the last artifact that was received per connector
5    @ bufferCurrentArtifactTypeInv  bufferCurrentArtifact ∈ Connector → ℕ
6    // Size of the set containing the ids
7    @ bufferArtifactListLenghtTypeInv  bufferArtifactListLenght ∈ Connector → ℕ
8
9  events
10   event Inject extends Inject
11     any i
12     where
13         @grd14 i ∈ ℕ
14       @grd15 i = bufferArtifactListLenght (conn)
15     then
16       // Set the id to the next available natural number
17       @act8 bufferArtifactList (conn) := bufferArtifactList (conn) ∪ {i ↦ a}
18       // Increment the connector id list
19       @act7 bufferArtifactListLenght (conn) := i+1
20   end
21
22   event Intercept extends Intercept
23     where
24       // Last id received can¬be equal to the size of the id list
25       @bufferCurrentIndexNotIndexSize ConnectorFIFOType(conn) = FIFO ⇒ bufferCurrentArtifact(
       conn) ≠ bufferArtifactListLenght (conn)
26       // Take the first sent artifact
27       @pickFirstSendGrd ConnectorFIFOType(conn) = FIFO ⇒ a = (bufferArtifactList(conn))(
       bufferCurrentArtifact (conn))
28     then
29       // Increment the last id received
30       @act5 bufferCurrentArtifact (conn) := bufferCurrentArtifact (conn) + 1
31   end
32 end
```

Listing 6.8: Refinement introducing connectors' FIFO property (3M)

```
1  invariants
2    // Set containing the state of the connector (TRUE = locked, FALSE = unlocked)
3    @inv1 connectorsLock ∈ Connector → ℙ
4
5  events
6    event INITIALISATION extends INITIALISATION
7      then
8        @act14 connectorsLock := Connector × {FALSE}
9    end
10
11   event Inject extends Inject
12     where
13       // make sure that the component does ¬have a artifact in transit
14       @NoMessageInTransitInv ConnectorSynchronousType(conn) = Synchronous ⇒ connectorsLock(
       conn) = FALSE
15     then
16       // Lock the connector when injecting in it
17       @lockConnector connectorsLock(conn) := TRUE
18   end
19
20   event Intercept extends Intercept
21     then
22       // Unlock the connector when intercept
23       @unlockConnector connectorsLock(conn) := FALSE
24   end
25 end
```

Listing 6.9: Refinement introducing connectors' synchronicity property (4M)

Listing 6.9 illustrates the formal encoding of this constraint, where synchronization behavior is enforced through guards on specific events. These guards ensure that a message is only transmitted through a connector if all required synchronization conditions are met, effectively capturing the essence of synchronous communication.

As with the FIFO constraint, this functional constraint is activated via a tagging relation. Specifically, connectors marked with the *Synchronous* tag in the *ConnectorSyncType* set are subject to the synchronization behavior constraints. The condition is thus enabled only for those connectors explicitly tagged, as expressed by the following guard condition:

$$(c \mapsto \text{Synchronous}) \in \mathit{ConnectorSyncType} \Rightarrow \text{Sync\_Guard}(c)$$

where *Sync_Guard*(c) defines the guard logic enforcing synchronous communication semantics for connector c.

6.5 Instantiation and Verification of Software Architecture

In this section, we will consider the college library web application system introduced in Section 5.2 to show how our contribution may be used for the modeling and analysis of software architecture. As we shall see, functional requirements are described in the form of properties of the modeled systems at different level of abstractions. Their fulfillment is then checked through model verification.

6.5.1 Expressing the Software Architecture

We start by defining the component types and the interfaces and connectors using the software architecture metamodel concepts and their interpretations in Event-B. Listing 6.10 shows the Event-B model specification of the architecture of the college library web application example described in Section 2.8.1, with respect to the functional requirements. Lines 3-4 specify the components, data types, data, and connectors for the web application according to the component-port-connector metamodel. Lines 7-10 specify the structural constraints in the form of axioms. For instance, the Database and Administrator components are defined as elements of the *Component* carrier set. Lines 13-18 specify the *Uses* relation between the ports and the components.

```
1  context SystemLibrary extends Context7MPS
2
3  constants Database Webserver Browser Administrator PortDatabaseIN PortDatabaseOUT
       PortWebserver PortBrowser PortAdministrator ConnectorBrowserWebserver...
4  ConnectorWebserverDatabase ConnectorAdministratorDataBase DataDatabaseIN DataDatabaseOUT
       DataWebserver DataBrowser DataAdministrator DataType1 DataType2
5
6  axioms
7      // Fill the Component set
8      @axmFillSystemComponents partition(Component, {Database}, {Webserver}, {Browser}, {
       Administrator})
9      // Fill the Port set
10     @axmFillSystemPorts partition(Port, {PortDatabaseIN}, {PortDatabaseOUT}, {PortBrowser}, {
       PortWebserver}, {PortAdministrator})
11     ...
12     // Linking the ports to their component
13     @axmFillSystemUses Uses =
14         {PortDatabaseIN ↦ Database,
15         PortDatabaseOUT ↦ Database,
16         PortWebserver ↦ Webserver,
17         PortBrowser ↦ Browser,
18         PortAdministrator ↦ Administrator}
19     ...
20     @axmFillConnectorParadigm partition(ConnectorParadigm,
21         {ConnectorAdministratorDataBase ↦ MPS},
22         {ConnectorBrowserWebsite ↦ MPS},
23         {ConnectorWebsiteDatabase ↦ MPS})
24     ...
25 end
```

Listing 6.10: Library context constant declaration and typing

6.5.2 Incorporating Connectors

At this step, we specify the structural and behavior of the connectors by reusing the developed and verified models. Connector types are defined as instances of the defined connectors in previous refinements according to the desired communication paradigms (e.g., MPS) and characteristics (e.g., FIFO) (See Lines 20-23 of Listing 6.10).

6.5.3 Specifying and Verifying the Requirements

As presented in Section 5.2, we can identify the corresponding communication semantics that will be needed to be used in order to fulfill the desired requirements. For instance, both the message passing communication and the message passing with FIFO ordering allow us to express *Req_3*. As identified in Section 5.2, two components (Administrator and Database) and a connector are involved in *Req_3*. To fulfill this requirement, the Administrator must respect the semantics of an MPS sender, and the Database must respect the semantics of an MPS receiver as described. Finally, the connector must respect the semantics of an MPS connector.

```
1  invariants
2    theorem @CanAlwaysCreateMessage ∃ m · m ∈ COMMUNICATION_ARTIFACT ∧
3        m ∉ artifact ∧
4        m ∉ message
5      theorem @CanSendMessageFromAdministratorToDatabase ∀ a · a ∈ message ∧ a ∉ artifact ∧ a ∉
       artifactIntercepted ∧ payloadsType(messagePayloads(a)) = DataType1
6          ⇒ (∃ conn · conn ∈ Connector
7          ∧ PortAdministrator ↦ conn ∈ Connects
8          ∧ PortDatabaseOUT ↦ conn ∈ Connects
9          ∧ PortAdministrator ∈ Port
10         ∧ PortDatabaseOUT ∈ Port
11         ∧ PortKinds(PortAdministrator) = PortOUT
12         ∧ PortKinds(PortDatabaseOUT) = PortIN
13         ∧ (ConnectorParadigm(conn) = MPS ⇔ a ∈ message)
14         ∧ (ConnectorParadigm(conn) = MPS ⇔ (Realizes(PortAdministrator) ∈ Data ∧ Realizes(
       PortDatabaseOUT) ∈ Data))
15         ∧ (ConnectorParadigm(conn) = MPS ⇒ payloadsType(messagePayloads(a)) ∈
       DatasDatatypes(Realizes(PortDatabaseOUT)))
16         ∧ (ConnectorParadigm(conn) = MPS ⇒ payloadsType(messagePayloads(a)) ∈
       DatasDatatypes(Realizes(PortAdministrator)))
17         ∧ (ConnectorParadigm(conn) = MPS ⇔ Realizes(PortDatabaseOUT) ↦ DataOUT ∈
       DataDataDirection)
18         ∧ (ConnectorParadigm(conn) = MPS ⇔ Realizes(PortAdministrator) ↦ DataIN ∈
       DataDataDirection) )
19     theorem @CanReceiveMessageWhenOnBuffer ∀ a · a ∈ message ∧ a ∈ buffer(
       ConnectorTerminalDataBase) ⇒
20       ConnectorTerminalDataBase ∈ Connector ∧
21       PortDatabaseOUT ∈ Port ∧
22       ConnectorFIFOType(ConnectorTerminalDataBase) ≠ FIFO
```

Listing 6.11: Library machine theorems

When discussing the satisfaction of requirements, we start with constructive aspects reusing abstract levels of description and then the specific aspects of the concrete system. We use some of the previously specified and verified structural and communication behavior properties for connectors to specify and verify the application-specific functional requirements on the concrete architecture, and we add new ones when required. For instance, the structural part of *Req_3* is fulfilled by construction, reusing the context of an MPS-based system, as depicted in Listing 6.10. Similar to the structural part, in the behavioral part of *Req_3*, we reuse the invariant "OnlyMessagesInMPSConnectors" introduced in Listing 6.4 and we add new properties defined as theorems (Listing 6.11). These theorems prove the validity/soundness of a succession of events (CreateMessage, Inject, and Intercept) and the properties of these events have been proven as invariants. In this

Table 6.1: Summary of the security elements and their interpretation in Event-B

Security element	Event-B interpretation
SecurityObjective	Comment at the start of the domain-independent machine
SecurityRequirement	Invariant in domain-independent machine
DomainSpecificPolicy	Comment at the start of the domain-specific machine
SystemPolicy	Theorem in domain-specific machine
DomainSpecificMechanismCapability	Application of the tag in the domain-specific context
DomainSpecificMechanism	Configuration of the system to the refinement made for the DomainIndependentMechanism
DomainIndependentMechanismCapability	Guard applied to the tag in domain-independent machine
DomainIndependentMechanism	Full refinement of the domain-independent machine to describe more concrete implementation

Table 6.2: Summary of the security relations and their interpretation in Event-B

Security relation	Event-B interpretation
Fulfills	Guards representing the DomainIndependentMechanismCapability appear in the SecurityRequirement proof
Implements (Domain independent)	Gluing invariant describing the SecurityRequirement in a more concrete manner in the refinement
Implements (Domain specific)	DomainIndependentMechanism configuration to the specific system holds the SystemPolicy invariant
Enforces	Tags necessary to prove the DomainSpecificPolicy theorem
Refines	Application of the tag in the domain-specific context
Configures	Configuration of the DomainIndependentMechanism refinement machine to a system
Achieves	Represented as the conjunction of requirement in the SecurityObjective comment

machine, the events and variables defined in the previous refinement are extended without any change. The same process may be applied to deal with *Req_1* and *Req_2*. In a more general cases, it is possible to modify the events (i.e., adding new guards, renaming them) to fulfill the target application use case. This process is currently manual, but the use of Event-B theories [148] could enable the automatic generation of such theorems.

At this level of specification, we are able to: (1) formally represent an interpretation of the software architecture model and properties in Event-B to support property specification and analysis, (2) formalize the behavioral semantics of the connectors described in Section 6.2, specifically by writing and proving invariants, (3) verify properties of the connectors to establish a set of reusable connector libraries that can be instantiated with respect the specific requirements in a concrete application, and (4) verify the satisfaction of a set of functional requirements for a concrete application through reuse of the previously specified and verified connectors. This satisfies ***RO 1.2****.*

6.6 Security Interpretation and Proof Engineering

In this section, we present a formal specification of the security metamodel, as illustrated in Figure 5.5, using the Event-B formal method.

The Event-B model developed to formally validate the modified version of NIST standard interprets aspects of the diagram with specific concepts. Tables 6.1 and 6.2 show a summary of the transformation rules between the metamodel elements and realtions and their Event-B representation.

- *SecurityObjective*: While the SecurityObjective is not explicitly represented in the Event-B model, it can be interpreted through abstract specifications that employ

formal semantics. For example, the "integrity of the origin of the information" is represented in the CIAAA machine as a comment. It has no formal specification in itself.

- *SecurityRequirement*: SecurityRequirement are formally represented as invariants within the Event-B machine. These invariants encode the requirements as predicates, leveraging set theory and first-order logic. The correctness of these invariants is ensured through proof by induction, guaranteeing their validity in all reachable states of the state machine. For example, "the source of the message must be verifiable as the component sending the message" is represented in the CIAAA machine as the *messageAuthenticityInv* invariant in the CIAAA machine.

- *DomainIndependentMechanismCapability*: The DomainIndependentMechanismCapability are modeled as guards in the Event-B machine. These guards encapsulate the necessary capabilities required to fulfill the associated requirements. The relationship between guards and requirements is formally verified, as the proof of the invariant representing the requirement is contingent on the proper application of these guards. For example, "ensure the source of the message is correctly initialized" is interpreted by the guard *restrictiveSetSourceGrd* in the Inject event of the CIAAA machine.

- *DomainSpecificPolicy*: The DomainSpecificPolicy is not explicitly formalized within the Event-B model. However, it can be abstractly represented and interpreted using formal semantics. For example, the DomainSpecificPolicy "web browser requests sender is correctly identified" can be represented in the domain-specific Event-B machine as a comment describing the steps needed to achieve the objective.

- *SystemPolicy*: The SystemPolicy is encoded as an invariant within the Event-B model. This invariant represents the SystemPolicy as predicates, which can be rigorously proven using first-order logic and set theory. For example, "the source of the web browser request sent to the webserver is the original sender" is represented in the domain-specific machine as a theorem *messageAuthenticitySystemPolicyInv* formally describing the behavior that we expect.

- *DomainSpecificMechanismCapability*: The DomainSpecificMechanismCapability are modeled by introducing behavioral modifications to system components. This is achieved by applying the DomainIndependentMechanismCapability to the relevant architectural elements. Within the Event-B framework, this is implemented by tagging the elements in the system model with the corresponding DomainIndependentMechanismCapability. The tagged elements activate the guards defined by the DomainIndependentMechanismCapability, thereby modifying their behavior in accordance with the specified capabilities. For example, "the source of the web browser request sent to the webserver is correctly initialized to the appropriate web browser" is represented in our Event-B model by adding the relation between the tag *ConnectorSenderAuthenticity* and the connector *ConnectorBrowserWebsite* inside the

set *ConnectorConstraint*. This enable the guards representing the DomainSpecificMechanismCapability for the connector enabling the proof of the theorem for it.

- *Mechanisms*: The formal implementation of mechanisms (both domain independent and specific) in the Event-B model would be done through refinement. The model aims to refine the simple capabilities into a more concrete Event-B representation changing the behavior of the system while keeping the invariant true. For example, for the payload confidentiality requirement is represented in the Event-B model as a refinement of the CIAAA machine that refines the already formalized DomainSpecificMechanismCapability. This refinement is made by the pair of context and machine and in this case represent the encrypt decrypt protocol in more details. This example demonstrates the feasibility of formally describing concrete mechanisms. This machine can in turn be refined to introduce the system and configure the DomainIndependentMechanism into the DomainSpecificMechanism. An additional verification of the concrete solutions or protocols could be done outside this approach using verification tools such as Tamarin [149] and ProVerif [150].

In addition to the main concepts, the relations are represented as followed:

- *Achieves*: The *Achieves* relation is not represented formally in the Event-B model as the objective is not either. This relation in the model is interpreted as the conjunction of the security requirements and domain specific policies in the Event-B model being proven.

- *Fulfills*: The *Fulfills* relation is shown in the proof of the Security requirement. In the proof, the guards representing the DomainIndependentMechanismCapability appear in the step making the proof possible (see Figure 6.4). This exemplifies that the guards used to represent the DomainIndependentMechanismCapability do in fact fulfill the SecurityRequirement.

- *Recasts*: The *Recasts* relation is can be found in the system machine. When looking at the theorem representing the DomainSpecificPolicy it is a recast of the SecurityRequirement invariant represented in the CIAAA machine. This recast takes the form on replacing the general approach of the invariant (for example: $\forall c \in Connector$) by the specific architectural element.

- *Implements*: The *Implements* relation is represented in multiple ways. For example a gluing invariant is introduced in the refinement showing that the guards introduced for the DomainIndependentMechanismCapability in the abstract machine is refined into a more concrete behavior in the concrete machine representing the Mechanism.

- *Enforces*: The *Enforces* relation is demonstrated when the introduction of the mechanism capabilities within the formal context enables the proof of the System Policy theorem. This implies that the defined capabilities provide the necessary conditions

to establish the correctness of the policy within the domain-specific formal model, ensuring that the intended security properties hold under all specified conditions.

- *Refines*: The *Refines* relation is represented when the tag representing the DomainIndependentMechanismCapability is applied to the relevant architectural element to apply the mechanism capabilities at the system level. The application of the tag enable the behavior of the mechanism capabilities in the element.
- *Configures*: The *Configures* relation is the application of the Mechanism concrete machine to the specific system by its configuration specific to the system it is applied to.

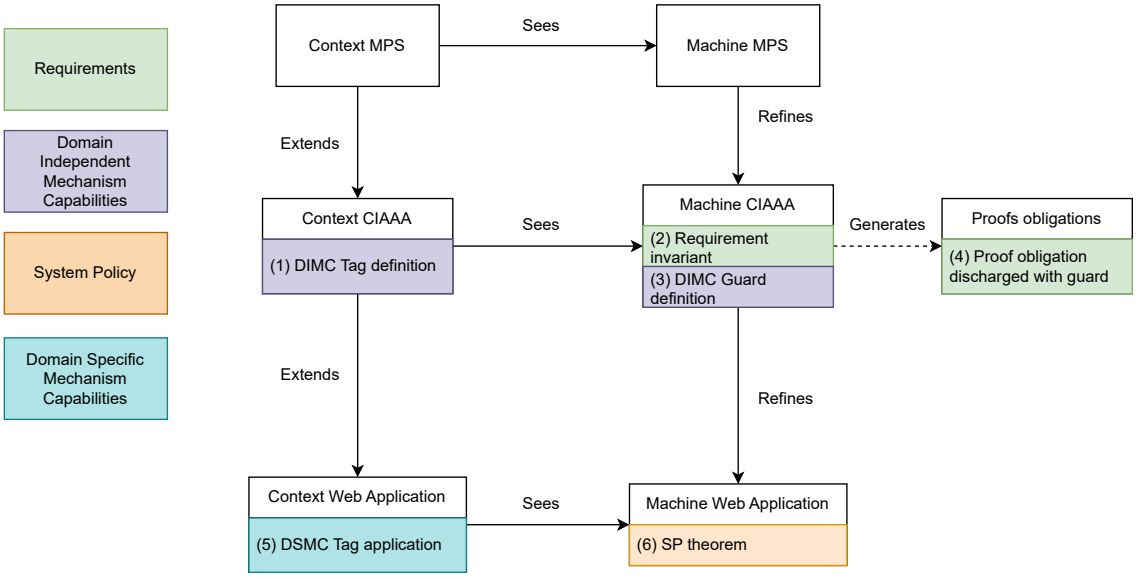

Figure 6.2: Event-B refinement tree of the interpretation of the our metamodel

The refinement tree illustrated in Figure 6.2 provides a formal depiction of the relationships between the concepts introduced, utilizing the refinement process inherent to Event-B. This figure is structured across three levels of abstraction: *MPS*, *CIAAA*, and *Web Application*. The first level, *MPS*, represents the functional machine that serves as the foundation for subsequent security refinements. It formalizes the MPS communication paradigm, as introduced in Section 6.3.2, and excludes any security-related constructs. The second level, *CIAAA*, corresponds to a domain-independent refinement that introduces a general formalization of security concerns. Finally, the *Web Application* level exemplifies a domain-specific instantiation of the abstract security model. This tier demonstrates how the abstract security framework can be tailored to a real-world application.

In the figure, the domain-specific elements are shown as refinements of the domain-independent components. This layering signifies a systematic progression, where the

general capabilities of domain-independent mechanisms are adapted and specialized to address the specific needs of the domain. Through this formal representation, the refinement process establishes a clear and traceable connection between the generic mechanisms capabilities and their context-specific refinement.

6.6.1 Mechanisms

In the Event-B formalism, a security mechanism (DIM) is represented as a refinement of the machine encapsulating the corresponding security mechanism capability (DIMC). This refinement process allows for the formalization of concrete mechanisms, such as TLS, while preserving the core security properties defined at the capability level. The refinement ensures that the invariants established in the abstract representation of the capability remain valid in the more concrete implementation. By progressively refining the abstract model, we introduce implementation details that align with real-world mechanisms while guaranteeing that the fundamental security properties, such as authenticity or integrity, hold throughout the refinement chain. This structured approach facilitates rigorous verification and validation of security mechanisms within a formal framework.

At this level of specification, we are able to: (1) specify the security properties (i.e. SecurityRequirement, DomainSpecificPolicy and SystemPolicy) as theorem and invariants in the Event-B machines, (2) specify the security constraints as guards on the events (i.e. DomainIndependentMechanismCapability) or by assigning tags to the corresponding architectural elements in the context (i.e DomainSpecificMechanismCapability), constraining the model behavior, (3) generate relevant POs corresponding to the invariants and the theorems, and (4) discharge the POs using hypothesis derived from the guards.

6.7 Specification and Verification of the System Architecture Security

In this section, we present examples of the formalization of the security metamodel using Event-B by extending the model presented in Section 6.3.

6.7.1 Instantiation of the Security Metamodel

In this section, we are going to present an example of instantiation of the Security metamodel using the Authenticity objective applied to the smart meter gateway case study as presented in Figure 6.3. This instantiation requires the combined work of the security expert and the analysis expert to produce a formal representation of the security concerns.

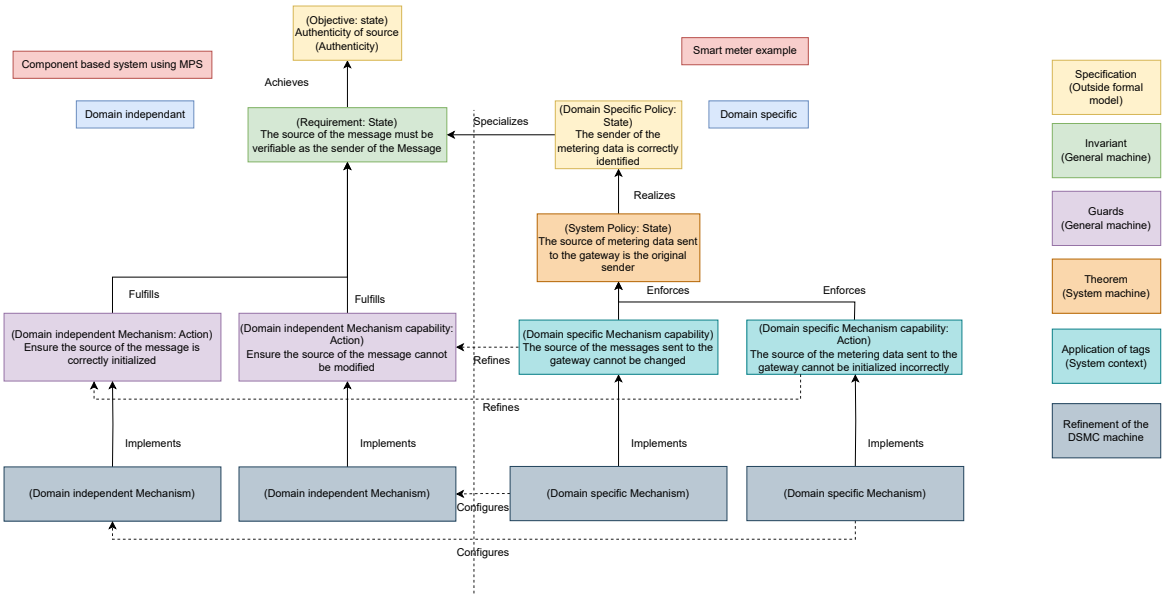

Figure 6.3: Instantiation of NIST SP 800-160 with authenticity example with every security elements associated with their respective Event-B formal Interpretation

Authenticity Formalization in Event-B

According to the ISO/IEC 27000:2018 standard [145], *authenticity* ensures that an entity is what it claims to be. In the context of message passing communications within software architectures, message authenticity guarantees that the source written in the message is the actual sender. If a message m has a component c_1 as its source then it was sent by c_1. In other words, for any message m, if a component c_2 is able to get source c_1 of a message m, then c_1 is the accurate sender of m. For components $s, r \in \mathcal{C}$, we denote this representative property as $MessageAuthenticity(s, r, m)$ which is specified for all messages $m \in \mathcal{M}$ as:

$$\mathbb{H}_{c_1}(inject(m)) \ < \ \mathbb{E}_{c_2}(get\_src(m, c_1))$$

Here, $\mathbb{E}_{c_2}(get\_src(m, c_1))$ is a predicate indicating that component $c_2 \in \mathcal{C}$ is able to get that the source of m is $c1$ and $\mathbb{H}_{c_1}(inject(m))$ is a predicate indicating that $c1$ has injected the message. Therefore, the validity of the property $MessageAuthenticity(s, r, m)$ would show that the $MessageAuthenticity$ objective is fulfilled between c_1 and c_2 in the system.

```
1 ========== (1) ==========
2 @AuthenticityConstraintsAxiom partition ( AuthenticityConstraint , {ConnectorSenderAuthenticity}, {
      ConnectorSendAuthorization})
3
4 ========== (2) ==========
5 @messageAuthenticityRequirementInv ∀ conn, m ·
6     conn ∈ Connector
7     ∧ conn ↦ ConnectorSenderAuthenticity ∈ ConnectorConstraint
8     ∧ m ∈ artifact
9     ∧ connectorsUsed(m) = conn
```

```
10 |     ⇒ source(m) = sender(m)
11 |
12 | ========== (3) ==========
13 | @ restrictiveSetSourceGrd  conn ↦ ConnectorSenderAuthenticity ∈ ConnectorConstraint ⇒ s =
   |     aSender
14 |
15 | ========== (5) ==========
16 | @axmFillConnectorProperty  partition (ConnectorConstraint,
17 |     {ConnectorGatewayRrcCD ↦ ConnectorSenderAuthenticity})
18 |
19 | ========== (6) ==========
20 | theorem @messageAuthenticitySystemPolicyInv ∀ m ·
21 |     m ∈ artifact
22 |     ∧ connectorsUsed(m) = ConnectorGatewayRrcCD
23 |     ⇒ source(m) = sender(m)
```

Listing 6.12: Event-B interpretation of the *Message Authenticity Requirement*

Listing 6.12 and Figure 6.4 show the six steps of this interpretation. Listing 6.12 shows a extract of 4 files as shown into Figure 6.2 while Figure6.4 show the final logic statement proving invariant (2) using automatic prover CVC4.

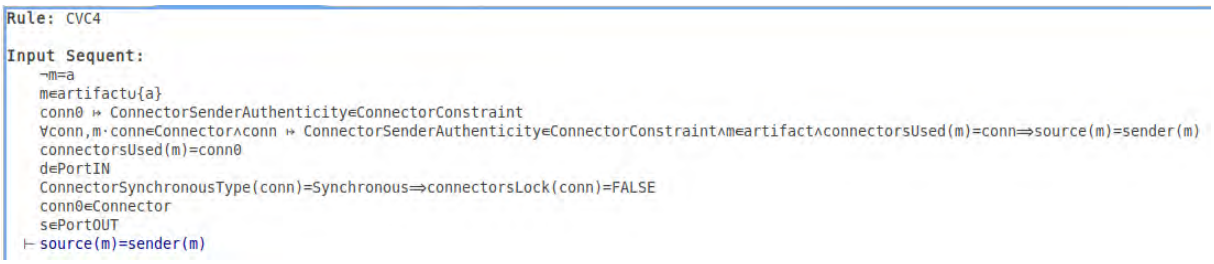

Figure 6.4: (4) Invariant proved using automatic tactic CVC4 and **ConnectorSender-Authenticity** tag

In the figure, the domain-specific elements are shown as refinements of the domain-independent components. This layering signifies a systematic progression, where the general capabilities of domain-independent mechanisms are adapted and specialized to address the specific needs of the domain. Through this formal representation, the refinement process establishes a clear and traceable connection between the generic mechanisms capabilities and their context-specific refinement.

In the context of the smart meter example, this refinement highlights how domain-independent mechanism capabilities are configured and tailored to meet the specific requirements of the domain. The refinement process thus ensures that the system maintains coherence and consistency while allowing for the customization necessary to achieve domain-specific objectives. This approach not only underscores the flexibility of the refinement process but also emphasizes its utility in bridging abstract concepts with their practical applications in diverse domains.

This example demonstrates the hierarchical interpretation of these concepts, culminating in the representation of mechanism capabilities for both domain-independent and domain-specific components.

Confidentiality Security Objective

According to the ISO/IEC 27000:2018 standard [145], *confidentiality* ensures that information is not made available or disclosed to unauthorized individuals, entities, or processes. In the context of message passing communications within software architectures, payload confidentiality allows a transmitted message to be received by other components, but only specific components can get the actual content of the payload. In other words, a payload is deemed confidential if a component c_3 receives a message m with a payload d sent by c_1 with c_2 as its intended destination, then c_3 will not be able to get the content of the payload d. Formally, this is expressed as:

$$\mathbb{E}_{c_3}(\text{get\_pld}(m,d)) \Rightarrow \neg(\mathbb{H}_{c_1}(\text{inject}(m)) \ \wedge \ \text{has\_rcv}(m,c_2) \ < \ \mathbb{E}_{c_3}(\text{get\_pld}(m,d)))$$

Here, $\mathbb{E}_{c_3}(\text{get\_pld}(m,d))$ is a predicate indicating that component $c_3 \in \mathcal{C}$ is able to get the payload $d \in \mathcal{D}$ from the message $m \in \mathcal{M}$ and $\neg(\mathbb{H}_{c_1}(\text{inject}(m)) \ \wedge \ \text{has\_rcv}(m,c_2) < \mathbb{E}_{c_3}(\text{get\_pld}(m,d)))$ is a predicate indicating that it is never the case at some past before that the component $c_1 \in \mathcal{C}$ sent message $m \in \mathcal{M}$ with a component $c_2 \in \mathcal{C}$ as the declared receiver. Therefore, the validity of the property *PayloadConfidentiality*(c_1, c_2) would show that the *PayloadConfidentiality* objective is fulfilled between c_1 and c_2 in the system.

Formal specification and analysis in Event-B. Following the same method shown in the previous Section for Authenticity we build the Event-B model for confidentiality. We build a new Event-B model named *CBSE MPS CIAAA Event-B model* as a refinement of the *CBSE MPS Event-B model*. Assuming a mechanism capability called *restrictiveGetPayload* is deployed, then we show that the security requirement *PayloadConfidentiality* holds. The payload confidentiality requirement is represented in Event-B as an *Invariant* (see Listing 6.13) that states that if a payload has been read and the message containing this payload has been sent into a connector enforcing *restrictiveGetPld* then the component that reads the payload is the intended destination.

```
1 invariants
2     @ payloadConfidentialityInv  ∀ p, conn, m ·
3         conn ∈ Connector
4         ∧ conn ↦ ConnectorPayloadConfidentiality  ∈ ConnectorProperty
5         ∧ p ∈ ran(readPayload)
6         ∧ m ↦ p ∈ messagePayloads
7         ∧ connectorsUsed(m) = conn
8         ⇒ Uses( receiver (m)) = allowedGetPayload(m)
```

Listing 6.13: PayloadConfidentiality property invariant

For the invariant encoding the security objective to hold, we add a constraint in the model (security mechanism), as shown in Listing 6.14). This constraint is modeled in Event-B as a guard (line 4) in the *GetPayload* event: when the considered message exchange is realized through a secure connector, the event can only be triggered if the message has been received by the intended destination.

```
1 events
2   event GetPayload extends GetPayload
```

```
3     where
4       @restrictiveGetPldGrd  connectorsUsed(m) ↦ ConnectorPayloadConfidentiality ∈
        ConnectorProperty ⇒ allowedGetPayload(m) = Uses(receiver(m)) // Only the inteded receiver
        can receive the message
5   end
```

Listing 6.14: restrictiveGetPld constraint guard

Integrity Security Objective

According to the ISO/IEC 27000:2018 standard [145], *integrity* ensures that transmitted information is accurate and complete. In the context of message passing communications within software architectures, message integrity ensures that the content of the message received is the same as when it was sent. If a message m, sent by the component c, has a payload p then c originally sent m with the same payload p. In other words, for any message m, if a component c_2 is able to get the payload p of a message m, then p is the initial payload of m. For components $s, r \in \mathcal{C}$, we denote this representative property as $PayloadIntegrity(s, r, m)$ which is specified for all messages $m \in \mathcal{M}$ as:

$$\mathbb{H}_s(set\_pld(m, p)) \ < \ \mathbb{H}_r(get\_pld(m, p))$$

Here, $\mathbb{H}_r(get\_pld(m, p))$ is a predicate indicating that component $r \in \mathcal{C}$ has gotten the payload $p \in \mathcal{D}$ of $m \in \mathcal{M}$ and $\mathbb{H}_s(set_pld(m, p))$ is a predicate indicating that $s \in \mathcal{C}$ has set the payload of m to p. Therefore, the validity of the property $PayloadIntegrity(s, r, m))$ would show that the $PayloadIntegrity$ objective is fulfilled between s and r in the system.

Formal specification and analysis in Event-B. Assuming a security mechanism capability called *restrictiveSetPld* is deployed, we show that the security requirement *PayloadIntegrity* holds. The payload integrity Requirement is represented in Event-B as an *Invariant* (see Listing 6.15) that states if a message has been injected into a connector enforcing *restrictiveSetPld* then the payload has not been changed.

```
1  invariants
2      @messageIntegrityInv  ∀ conn,  m ·
3          conn ∈ Connector
4          ∧ conn ↦ ConnectorMessageInTransitIntegrity  ∈ ConnectorProperty
5          ∧ m ∈ artifact
6          ∧ m ∈ message
7          ∧ connectorsUsed(m) = conn
8          ⇒    initialPayload (m) = messagePayloads(m) // every message m send in connector conn
       have its original payloads
```

Listing 6.15: PayloadIntegrity property invariant

For the invariant encoding the security objective to hold, we incorporate a constraint in the model (security mechanism capability), as depicted in Listing 6.16. This constraint is modeled in Event-B as a guard (line 4) in the event *ChangePayload*: when the considered message is sent through a secure connector (the connector has the property *ConnectorMessageInTransitIntegrity*), the event cannot be triggered (the guard *restrictiveSetPldGrd* cannot be true).

```
1 event ChangePayload
2   where
3     @SetPldGrd connectorsUsed(m) ↦ ConnectorMessageInTransitIntegrity ∉ ConnectorProperty
```

Listing 6.16: restrictiveSetPld constraint guard

6.7.2 Analysis Results and Discussion

Table 6.3 presents the count of *Events* and *Guards* used to specify both the properties and behavior of the system. It also shows the number of *proof obligations (PO)* to prove the correctness of the model (behavior and properties) at each refinement level. All of the generated proof obligations have been discharged. Some have been discharged automatically. Others required interactive proof through explicit case analysis and/or hypothesis instantiation and rewriting.

Table 6.3: Analysis results of the Event-B models

Machine	**Events**	**Guards**	**Proof Obligations**
2 Communication	3	11	27
3 Buffers	3	14	28
4 FIFO	3	4	26
5 Synchro	3	1	4
6 MPS	4	12	42
7 RPC	5	13	35
Authenticity	5	1	14
Payload confidentiality	5	1	24
Message confidentiality	5	1	16
Integrity	5	1	5
Library	4	0	5

6.8 Verification Informs Design

Once the architecture is translated into an Event-B model, the associated POs, particularly those derived from property theorems, are automatically generated. The status of these POs serves as a primary indicator of the validity of the specified properties. A *proven* status indicates that a given property holds under the current specification and set of assumptions. Section 6.5 illustrated an example of a system instantiation and the corresponding proof process. For example, in the Webserver use case, the PO generated from the authenticity theorem serves as a formal indicator of whether the authenticity property is upheld in the modeled system.

The status of each PO provides critical feedback for the architectural model design. However, the formal analysis process does not inherently differentiate between POs that

are unprovable due to a flaw in the model and those that remain unproven due to an incomplete or inadequate proof strategy. In such situations, the intervention of a formal methods expert may be required to diagnose whether the issue arises from the system specification itself or the reasoning process used by the proof assistant. In the Webserver case study, if the *restrictiveSetSource* constraint is omitted from the connector, the PO associated with the authenticity property will not be discharged. This outcome highlights, through formal analysis, that the current system design fails to satisfy the required authenticity guarantees, thereby indicating a security flaw.

When certain proofs remain undischarged, this outcome serves as a signal that additional constraints are necessary to satisfy the stated properties. Within our framework, this often translates to the application of additional mechanisms, guided by the *Enforces* relation between DomainSpecificMechanismCapability and DomainSpecificPolicy. In the Webserver case study, the inclusion of the *restrictiveSetSource* constraint in the system design enables the successful proof of the authenticity property, as demonstrated in Section 6.7. This illustrates the feedback loop between formal analysis and model design, where the results of the analysis directly inform necessary design adjustments to ensure security properties are satisfied.

This process will be discussed in more details including its tool support in Section 7.6.2 of the following Chapter.

At this level of specification, we are able to: (1) formally represent an interpretation of the security objectives in Event-B to support property specification and analysis, (2) use the semantics of the connectors described in Section 6.3, to interpret security objective, (3) verify properties of the connectors to establish a set of reusable objective libraries that can be instantiated with respect the specific requirements in a concrete application, and (4) verify the satisfaction of a set of non-functional security requirements for a concrete application through reuse of the previously specified and verified objectives. This satisfies ***RO 2.1*** *and* ***RO 2.2****.*

Chapter 7

Tool support

Contents

7.1 Introduction

In this chapter, we present the creation of the Domain-Specific Language and the tool support associated with it that we call *SecCBSA*. The implementation leverages the Eclipse Modeling Framework (EMF) ecosystem, particularly focusing on technologies such as Sirius for graphical modeling and Acceleo for Model-to-Text (M2T) transformation. The objective of this tooling environment is to provide users with an integrated platform to design, visualize, and generate artifacts from models conforming to the proposed metamodel. Sirius enables the creation of intuitive and customizable graphical editors, while Acceleo facilitates the automatic generation of formal specifications and implementation code from high-level models. Together, these tools put together in an Eclipse plugin called *SecCBSA*, contribute to a Model-Driven Engineering (MDE) workflow that supports the end-to-end development of CBS with embedded security properties and constraints.

This chapter present the creation and usage of the tool-chain answering **RO 4**. This chapter is organized as follows. Section 7.2 presents the requirements for the toolchain. Section 7.3 describes the implementation approach for the framework. Section 7.4 in-

troduces the abstract syntax of our DSL, including the concept of views and details the views provided by our framework. Section 7.5 presents the implementation of the abstract syntax as a concrete syntax using Sirius. Section 7.6 demonstrates the tool support for Event-B models through the Rodin platform. Finally, Section 7.7 describes the Model-to-Text transformation from Ecore models to Event-B models using Acceleo.

7.2 Tool Support Requirements

In this section, we outline a set of functional requirements for the toolchain (*SecCBSA*) implementing the tool support that supports the proposed approach. This toolchain is designed to operationalize the security metamodel introduced in our work and to integrate with the Event-B formal verification environment. The toolchain must fulfill the following core requirements:

- Support the creation and editing of UML class diagrams use to describe CBS.
- Enable the creation and editing of UML class diagrams used to describe the security metamodel.
- Provide support for defining a user-friendly concrete syntax for domain-specific languages used in the modeling process.
- Support the implementation of reusable security library models, including domain-independent and domain-specific components.
- Enable the creation and manipulation of system architecture models, including both functional and security-related views.
- Provide mechanisms to transform models from the UML format (e.g., Ecore) into the formal language format compatible with Event-B (e.g., Event-B machines and contexts).
- Allow integration with the Rodin platform to perform formal verification of system models using proof obligations generated from Event-B specifications.
- Allow the validation of the formal model through simulation using ProB.
- Support an iterative process to create a secure architecture with formal verification.

To address the above requirements, we have developed a MDE toolchain that supports all stages of our methodology—from architecture modeling to formal verification. This integrated tool environment aims to bridge the gap between high-level architectural design and rigorous formal validation, while preserving a separation of concerns across different modeling views and stakeholders.

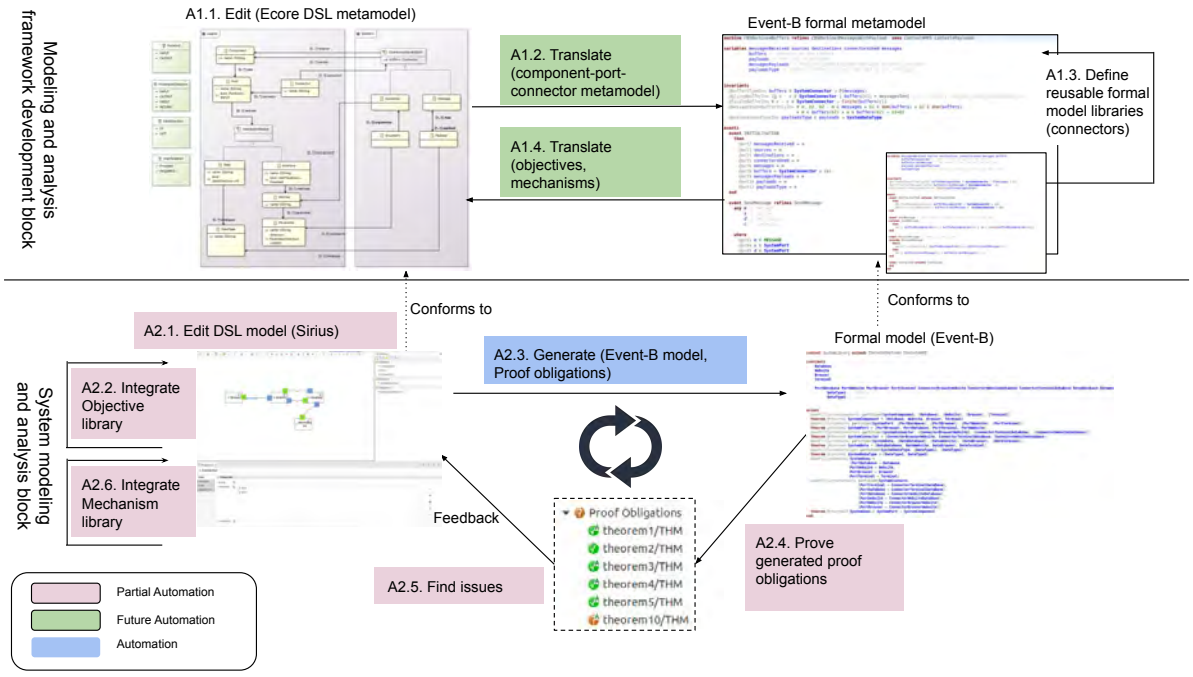

Figure 7.1: Architecture of the tool support

7.3 Architecture of *SecCBSA*

This section presents *SecCBSA* implemented to support the approach as an Eclipse plug-in[1]. Our starting point is the software architecture metamodel presented in Section 5.2 as the metamodel for a software architecture domain specific language (DSL) in a model-driven approach. The metamodel describes the abstract syntax of the DSL, by capturing the concepts of the component-port-connector software architecture domain and how the concepts are related.

The architecture of the tool, as shown in Figure 7.1, consists of two main blocks composed of a set of modules to support the corresponding activities. In the sections below, the numbers correspond to the activity numbers in Figure 7.1.

7.3.1 Modeling and Analysis Framework Development Block

The first block supports four activities. A1.1 is responsible for creating the DSL metamodel. The resulting metamodel is used in A1.2 to formally define (in Event-B) the static and behavioral semantics, based upon the DSL metamodel according to the procedure discussed in Chapter 6. We name this definition the formal metamodel. Furthermore, reusable security model libraries are also defined in A2.3 as Event-B models according to the procedure discussed in Section 6.7. We name this definition the formal model libraries. Lastly, A1.4 defines a set of DSL model libraries from the Event-B specifica-

[1]Also available in our public git repositories

tion of properties and constraints (formal model libraries), using the property view of the DSL metamodel (Figure 5.3). Each security property is associated with a set of security mechanisms to enforce it, during A1.3. As an example, we use *payloadConfidentiality* as an instance of *ConnectorProperty*, *restrictiveGetPld* as an instance of the *ConnectorConstraint* within the CIAAA library. Finally, we establish that the *restrictiveGetPld* requirement enforces the *payloadConfidentiality* objective. To support these four activities, we used EMF and Xtext to develop the DSL metamodel, and Rodin to encode the corresponding Event-B formal metamodel. No further implementation was required.

7.3.2 System Modeling and Analysis Block

The second block supports six activities. A2.1 allows the designer to model a software architecture (DSL model) conforming to the DSL metamodel. A2.2 allows the designer to integrate the security specification reusing the already developed formal model libraries (A1.4). Then, A2.3 generates a formal model (in Event-B) from a DSL model, using a transformation engine. This transformation creates a refinement of both the corresponding machine and context of the Event-B model. In A2.4, Rodin is invoked, and the generated model creates a number of proof obligations that may be automatically discharged. A2.5 analyzes the status of the proof obligation and A2.6 allows the designer to add the corresponding security requirements to the DSL model reusing the DSL model libraries and the mitigation relationship between properties (A1.4). The resulting DSL model is then transformed to a formal Event-B model (A2.3), where the formal definition of the corresponding security requirements is automatically added in the produced Event-B software architecture model from the DSL definitions. At the end of mitigation (i.e., all proof obligations are discharged), the architecture satisfies the requirements.

To support these six activities, we developed a graphical editor to model the architecture of the system using EMF and Sirius. The graphical editor provides model import for the manual integration of the objective library (A2.2), proposing the possible objective categories and properties for a software architecture element, when the designer is typing the elements. We also developed two transformation engines using Acceleo to generate the formal Event-B machine and context (A2.3). The transformation process consists of a set of transformation rules that are applied for each concept on a DSL model to generate an Event-B formal model. The generation process automatically incorporates the formal definition of the targeted objective from the DSL definition in the produced Event-B software architecture model. Finally, Rodin is used to verify the resulting models and to prove the different proof obligations. Additionally, ProB may be used to simulate and validate the architecture model.

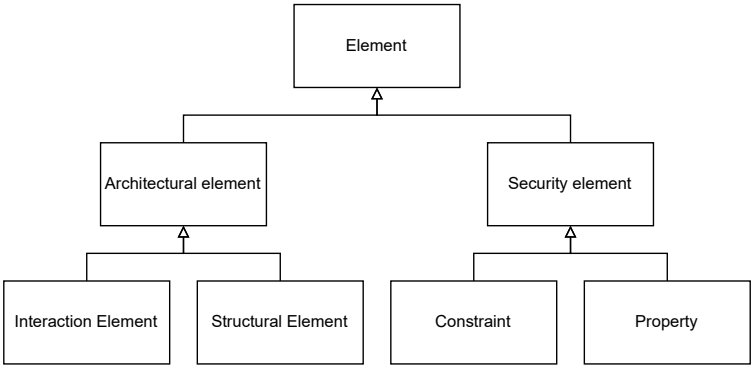

Figure 7.2: Abstract syntax elements categories

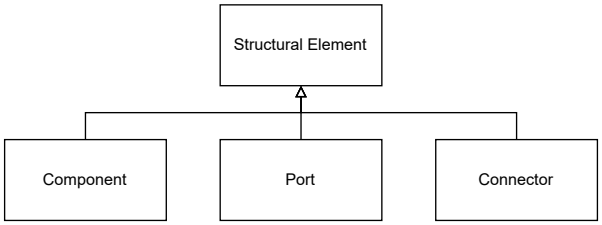

Figure 7.3: Abstract syntax Structural elements

7.4 Abstract Syntax

The abstract syntax of a language defines its underlying structural representation. It serves as a foundational layer for developing concrete syntaxes and acts as the basis for subsequent tool development. While there is no universally accepted standard for defining abstract syntax, in this work, we adopt a hierarchical graphical representation to describe the elements and relations that compose our language.

To organize the components of the language, we propose a hierarchical classification of elements into distinct conceptual categories, as illustrated in Figure 7.2. Each language element is modeled as a refinement or specialization of one of these core concepts.

Figures 7.3 and 7.4 present the architectural elements of the language, which are categorized into two main groups: *Structural* elements and *Interaction* elements. Similarly, Figures 7.5 and 7.6 illustrate the classification of the security-specific elements, grouped into *Properties* and *Constraints* as presented in Chapter 5.

This hierarchical abstraction facilitates modularity, reuse, and clarity in the language definition, supporting both tool implementation and future language extensions.

All elements are represented as classes within the metamodel and are reorganized independently of their relations to provide a clear and comprehensive view of all the concepts that a view can encompass. This isolation facilitates a better understanding of the individual components without the complexity of their interactions.

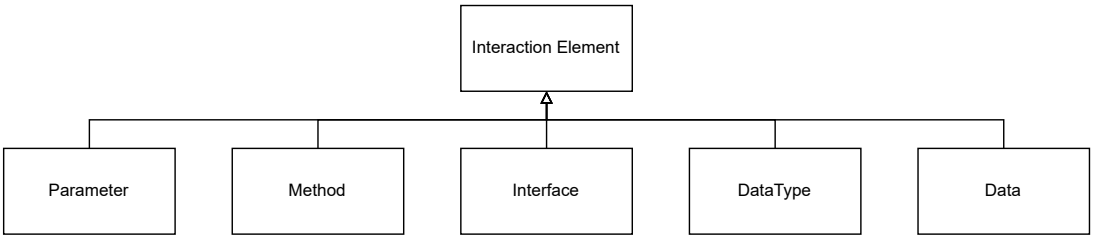

Figure 7.4: Abstract syntax Interaction elements

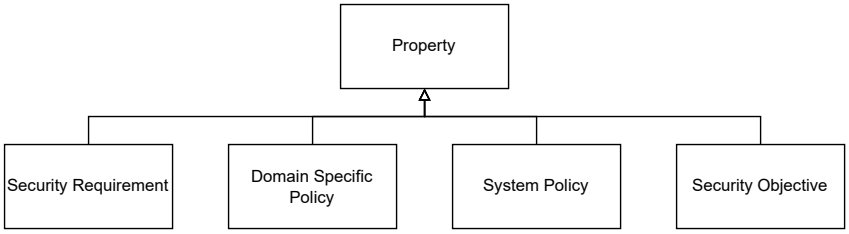

Figure 7.5: Abstract syntax Property elements

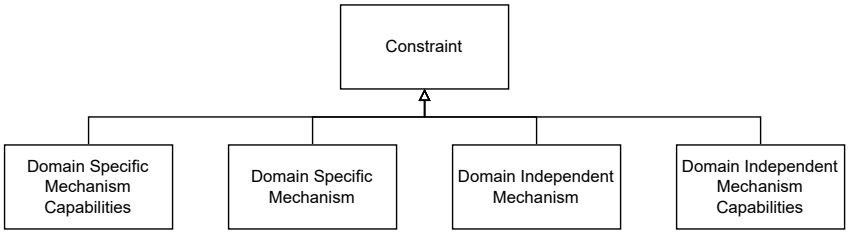

Figure 7.6: Abstract syntax Constraint elements

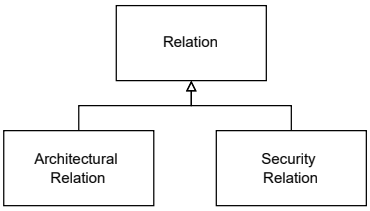

Figure 7.7: Abstract syntax relations categories

Additionally, Figure 7.7 presents the relations categories of the language in a similar fashion between the *Architectural* and *Security* relations. These relation are depicted respectively in Figure 7.8 and 7.9 highlighting both the source and destination elements associated with each relation. This separation enables a more structured representation of the interactions between elements.

By creating a separation between elements and relations, the syntax can be more easily leveraged for implementing tools that represent and manipulate the metamodel. This modular approach enhances the maintainability, reusability, and adaptability of the toolchain, making it easier to extend or modify the implementation according to evolving requirements.

7.4.1 Views

To transform implementation diagrams into usable components that can be developed into functional toolchains, we introduce the concept of *views*. These views are designed to uphold the principle of *separation of concerns*, enabling a clear division of responsibilities across the system's design and development process.

By organizing the model into distinct views, we facilitate the development of specialized tools tailored to each specific concern. This modular structure allows domain experts to focus exclusively on their area of expertise—be it architecture, behavior, constraints, or security—without being encumbered by unrelated details as shown in Figure 4.2. When integrated, these tools collectively form a cohesive and comprehensive toolchain, supporting the end-to-end design, analysis, and verification of complex systems.

A view-based representation of architectural and security component is not new. For example the 4+1 view model of software architecture present a method to describe the architectural aspect of a system using 5 different views [151]. Figure 7.10 illustrates our interpretation of the concept of *views* in the context of implementation diagrams. Each view is characterized as an aggregation of both *elements* and *relations* that are relevant to a specific concern within the system architecture.

This representation allows for a modular decomposition of the system, enabling focused reasoning and tool support per view. By isolating concerns into well-defined views, the overall complexity of the model is reduced, and the design process becomes more traceable and scalable.

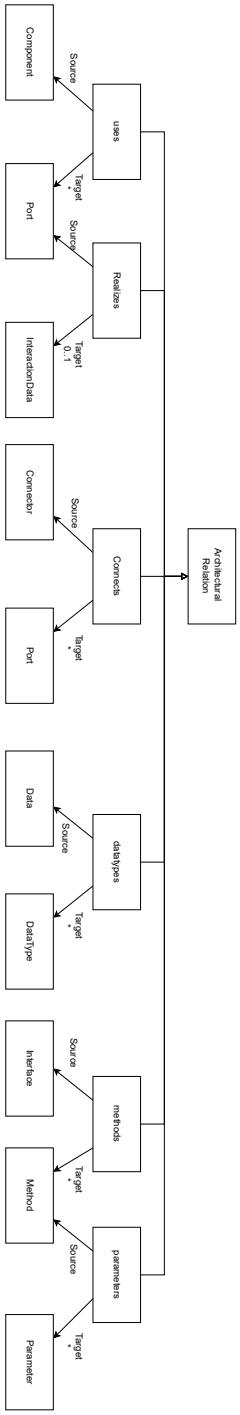

Figure 7.8: Abstract syntax Architectural relations

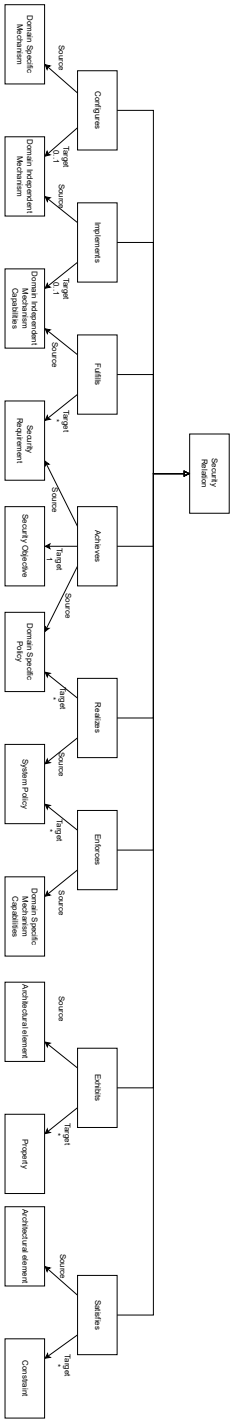

Figure 7.9: Abstract syntax Security relations

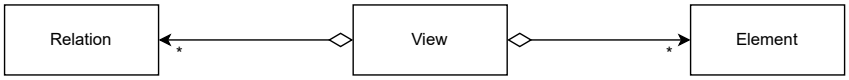

Figure 7.10: Implementation diagram: Views

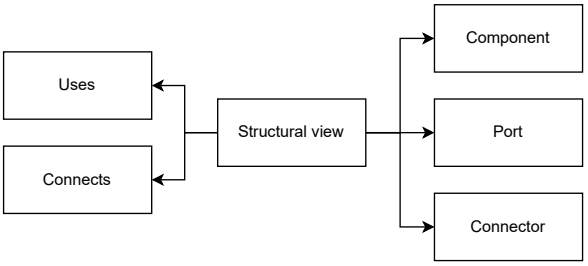

Figure 7.11: Instantiation of the Structural view

In this work, we present a set of illustrative views that span both the functional and security aspects of a component-based software engineering (CBSE) approach. These views serve to demonstrate how distinct concerns can be formally modeled and analyzed within a unified framework.

Structural view

The structural view represents the architectural elements that compose the system's overall architecture. This view is a representation of the metamodel presented in Section 5.2. This view provides a static representation of all system elements, illustrating how they are interconnected to form a cohesive structure. Each element in this view defines a fundamental building block of the system, specifying its role and relationships with other components. By capturing the static aspects of the architecture, the structural view serves as a foundation for understanding the system's organization, facilitating analysis, design, and integration of various system elements while ensuring consistency with architectural constraints and design principles.

For example, the representation of the smart metering system as a composition of interconnected components, such as the gateway and sensors, along with the ports and connectors connecting them, constitutes an instance of the Structural view.

Elements: Component, Port, Connector

Relations: uses, connects

Data view

The data view represents the various ways in which the elements of the structural view can interact. This includes defining the communication protocols they use, the data types exchanged, and the interaction rules governing their behavior. These interaction elements are instantiated alongside the structural elements and remain static throughout

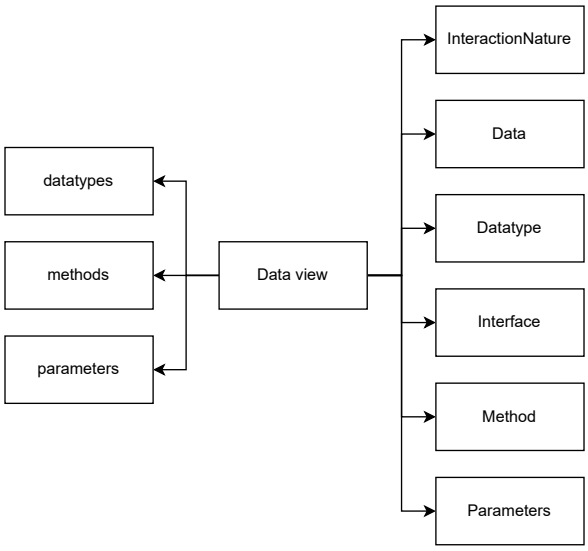

Figure 7.12: Instantiation of the Data view

the system's lifetime, ensuring consistency in communication and data exchange. By explicitly modeling interactions, this view provides a clear and structured representation of how different system components collaborate to achieve the desired functionality while adhering to predefined operational properties.

This view represents the possible configuration of the ports and connector for the communication. For example, in the smart meter example we can describe multiple Datatypes for the information that the sensor produces and package them into a Data bundle that represent the possible data that a sensor can send via message passing communication. Similarly interfaces and methods could be described if RPC is used.

This view represents the possible configurations of ports and connectors for communication within the system. For instance, in the smart metering example, various Datatypes can be defined to represent the information produced by the sensors. These datatypes can then be encapsulated into a data bundle, which characterizes the possible data that a sensor can transmit using message-passing communication. Similarly, if remote procedure calls (RPC) are employed, the view would include the definition of interfaces and methods that facilitate interaction between components.

Elements: InteractionNature, Data, Datatype, Interface, Method, Parameters

Relations: datatypes, methods, parameters

Behavioral view

The behavioral view represents the elements used for the behavior of the system. These elements aim to show the evolution of the system in a dynamic way.

This view emphasizes the simulation aspect of the model, capturing the dynamic

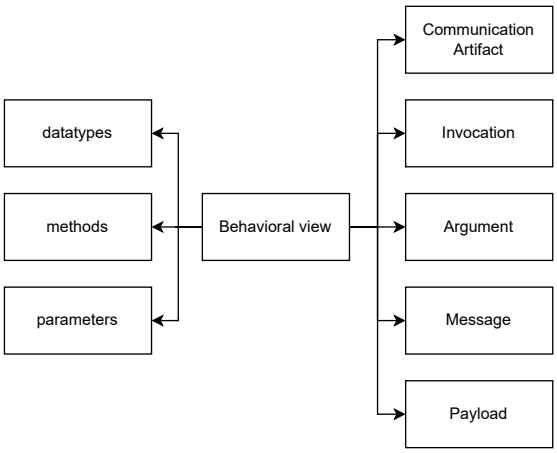

Figure 7.13: Instantiation of the Behavioral view

behavior of the system during execution. It encapsulates the artifacts exchanged between various components, along with their content. For instance, when a sensor generates data, it must encapsulate this data within a message payload before transmitting it to the gateway. The gateway, in turn, processes the received message and distributes it accordingly, following the defined communication protocols.

Elements: CommunicationArtifact, Invocation, Argument, Message, Payload

Relations: arguments, payload

Security Requirement view

The security requirement view is a security-focused perspective designed to represent domain-independent solutions to security objectives. It defines both security properties and constraints that are intended for general use and can later be adapted to specific systems through configuration or refinement. This approach ensures that high-level security principles remain consistent across different systems while allowing for their tailored application to meet the unique policies of a given implementation.

This view can incorporate certain requirements and mechanisms that have been designed independently of any specific domain. For example, a general security requirement such as the confidentiality of payloads, along with its associated mechanisms like TLS, would be represented within this view. These domain-independent elements provide reusable security solutions that can later be instantiated and configured for specific system implementations, ensuring consistency and facilitating the integration of well-established security practices across different domains.

Elements: SecurityRequirement, DomainIndependentMechanismCapability, DomainIndependentMechanism

Relations: fulfills, implements

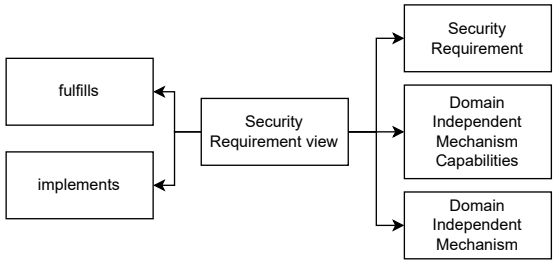

Figure 7.14: Instantiation of the Requirement view

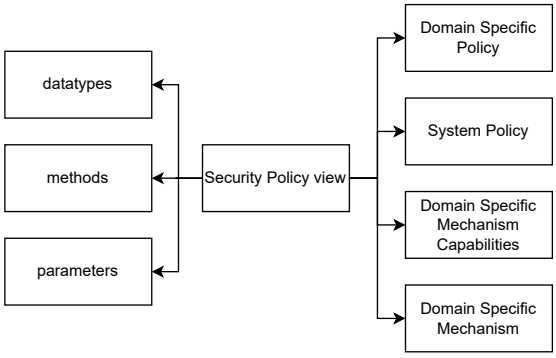

Figure 7.15: Instantiation of the Policy view

Security Policy view

The policy view is a security-focused perspective designed to represent domain-specific solutions at both the domain and system levels. This view is centered on a particular system and illustrates how it utilizes domain-independent elements, as defined in the requirement view, to fulfill its SecurityObjective. By bridging high-level security goals with concrete system-level implementations, the requirement view ensures that security policies and mechanisms are aligned with the system's specific needs while maintaining consistency with broader domain-independent security principles.

Elements: DomainSpecificPolicy, SystemPolicy, DomainSpecificMechanismCapability, DomainSpecificMechanism

Relations: realizes, enforces, implements

7.5 Concrete Syntax

The DSL editor is built upon the Eclipse Modeling Framework (EMF). Consequently, all models are defined using the *Ecore* metamodeling language, enabling seamless manipulation and integration within the custom tools developed as part of our work.

To enable a reusable modeling process that adheres to the principle of separation of

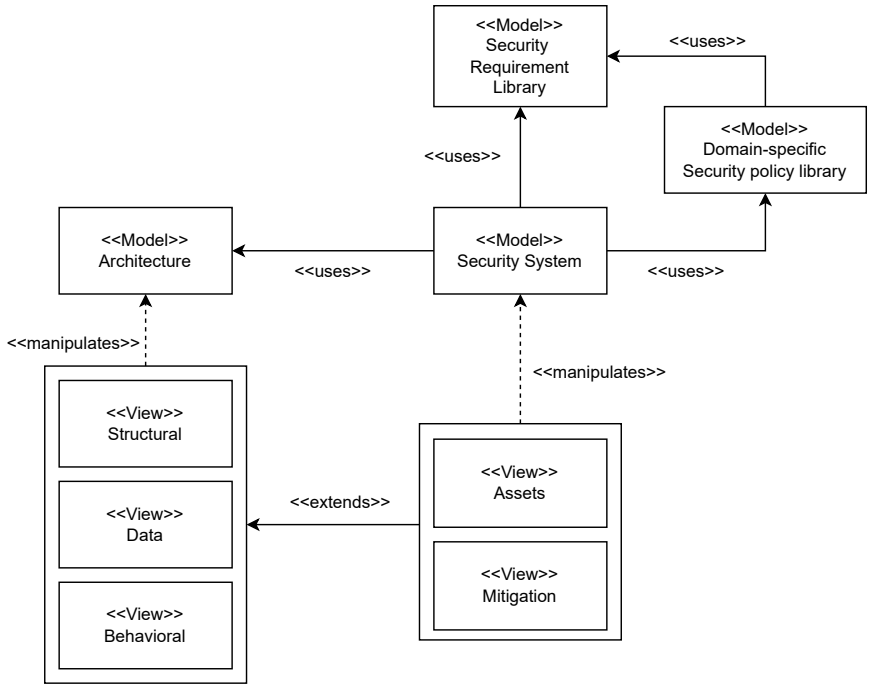

Figure 7.16: Ecore models of a secure system and views associated with them

concerns, our toolchain is structured around multiple *Ecore* models (See Figure 7.16). These models are designed to be accessed and manipulated through distinct views, each targeting specific aspects of the system architecture for both functional and non-functional concerns.

For the implementation of the concrete syntax of our DSL, we utilized Sirius, which is fully integrated into the Eclipse Modeling Framework (EMF). To develop this graphical interface, Sirius enables the creation of various views to manipulate models, facilitating the visualization and interaction with different components of the system. Furthermore, it allows for the seamless establishment of relationships between models, thereby promoting the reuse of existing models, such as the functional architecture models and the security models. This approach ensures a user-friendly environment for modeling complex systems while maintaining consistency and traceability across different domains. The creation of the architecture, interaction and security models supports activities *A2.1*, *A2.2* and *A2.6* of the complete toolchain as visualized in Figure 7.1.

7.5.1 Functional Architecture

The functional architecture is organized into three distinct views: *Structural*, *Data*, and *Behavioral*. As detailed in Section 7.4.1, these views collectively capture both the structural and behavioral dimensions of the architecture. All elements manipulated across these

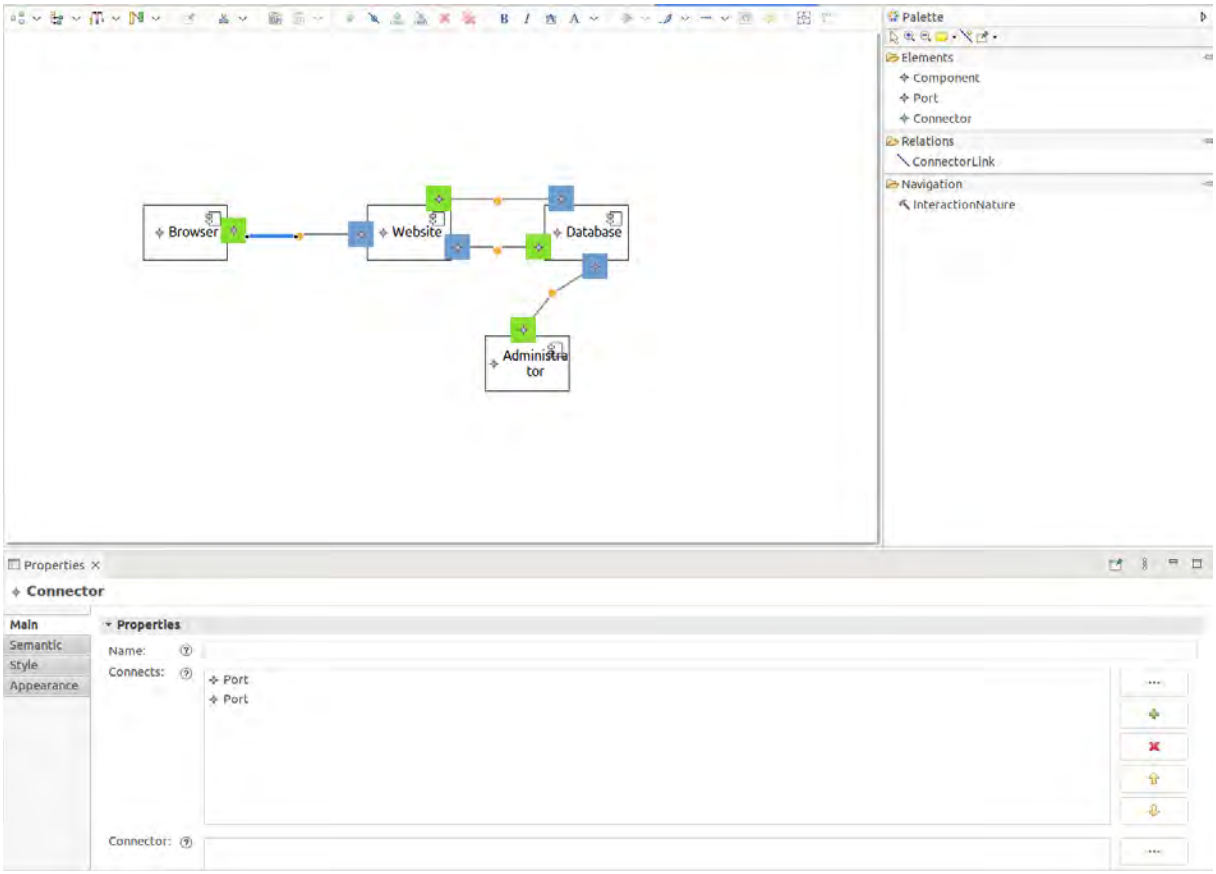

Figure 7.17: Web application example architecture diagram in Sirius representation

views originate from a single underlying `Ecore` model, which serves as a comprehensive representation of the architecture.

The *Structural View* and the *Data View* are presented through dedicated graphical interfaces, each tailored to the elements and relations relevant to their respective domains. For instance, the Structural View manages elements such as components, ports, and connectors, while the Data View focuses on entities such as datatypes, methods, and interfaces. Figure 7.17 shows the representation of the web application using the graphical representation. In this case the Components are representated as white boxes with the port attached to them. Green port represent output while blue represent input Ports. Finally the Ports are linked with Connectors.

The *Behavioral View* is a specialized view represented in the form of sequence diagrams. Its purpose is to model key interaction scenarios within the system, thereby supporting early-stage validation and comprehension of behavioral aspects. This separation of concerns enhances modularity and facilitates focused analysis of the architecture from multiple perspectives.

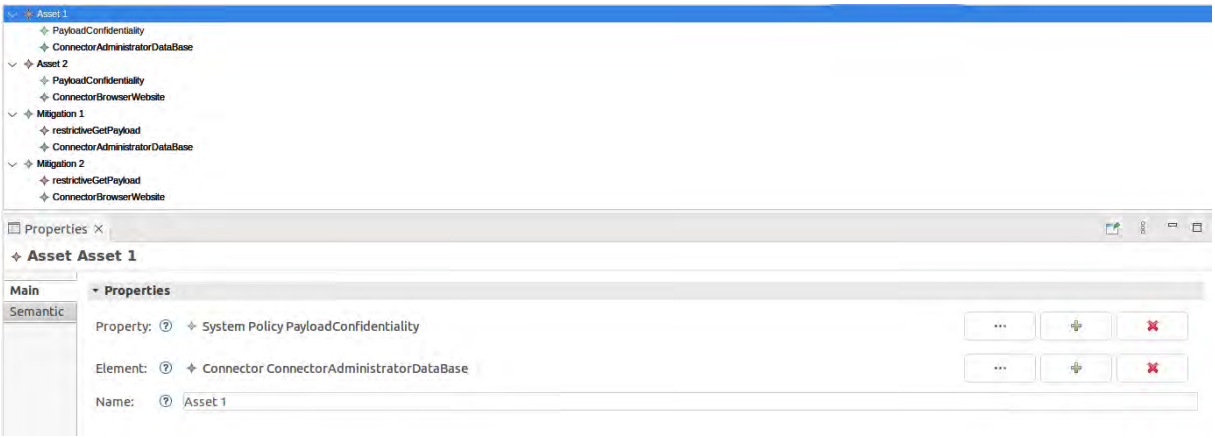

Figure 7.18: Tree-based editor for assets and mitigations in Sirius representation showing the relations between the architectural elements and the security properties of the model

7.5.2 Security Properties

The security properties model associated with a system must import multiple *Ecore* models in order to construct a complete secure system model. First, the architectural model must be imported, followed by the security libraries that define the properties and constraints to be applied to the architecture (see Figure 7.18).

Once the relevant models are imported, a security expert can establish the necessary connections between the security properties and constraints, and the corresponding architectural elements. This process is facilitated through a tree-based editor, which supports the creation of *assets* and *mitigations*—two concepts introduced in our implementation to represent these associations (see Figure 7.19).

Specifically, the notion of *assets* is used to link security properties to architectural elements that are expected to uphold them. For example, if a component is required to exhibit a certain security property, an asset is created to establish this relationship. Conversely, *mitigations* link architectural elements—such as connectors—to security constraints they are intended to satisfy. This abstraction enables a structured and traceable way to manage and enforce security requirements within component-based software architectures.

7.5.3 Model Libraries

To support the design of the security aspects of the architecture, we provide a support to domain experts to model security element libraries. We used the tree-based editor of the EMF. Figure 7.20 shows an Ecore model representing three requirement from the CIAAA model. Additionally, three mechanism capabilities that fulfill these requirements are also added to this library. This library contains domain independent requirements and mechanisms that can then be reused in other models.

As illustrated in Figure 7.16, these libraries encompass both domain-independent and

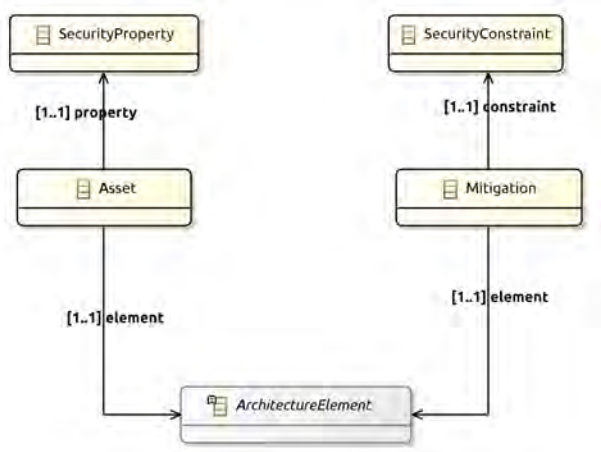

Figure 7.19: Metamodel of the asset and mitigation concepts

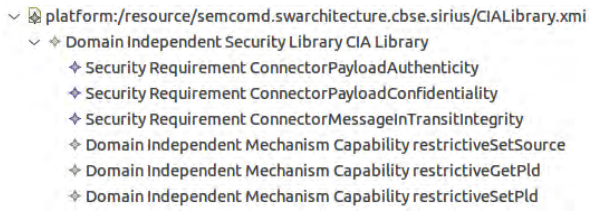

Figure 7.20: Ecore model of a representation of CIAAA SecurityRequirement and DomainIndependentMechanismCapability library related to Confidentiality, Integrity and Authenticity

domain-specific security components presented in Section 5.4, enabling reuse and consistency across different system models. For comprehensive system verification, the security libraries must be formally integrated into the verification toolchain. This ensures that the security elements can be subjected to rigorous formal analysis, thereby enabling end-to-end verification of the system architecture.

These libraries encompass both properties and constraints that define the system's behavior and security characteristics. Specifically, they include mechanisms that are directly associated with their respective requirements. For instance, each security mechanism in the library is linked to a requirement. These libraries may then be integrated into the systems following activities A2.2 and A2.6.

The same process is done to create a domain-specific library (See Figure 7.21). In order to use the CIAAA library created in a different file, we import that model as a resource (see Figure 7.22). Once the model is imported as a resource, then the elements created in the imported model can be used to fill the relations of the elements of the model such as the *Refines* relation between the DomainIndependentMechanismCapability and DomainSpecificMechanismCapability (see Figure 7.23). The process of importing a model as a resource will be repeated in the security system to import the libraries and the architecture to create a complete model.

Figure 7.21: Ecore model of a representation of the Webserver DomainSpecificPolicy, SystemPolicy and DomainSpecificMechanismCapability library

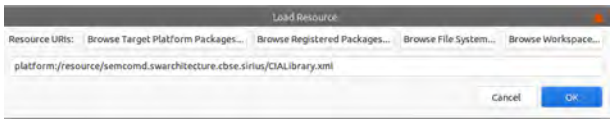

Figure 7.22: Import of the CIAAA library Ecore model as a ressource in EMF

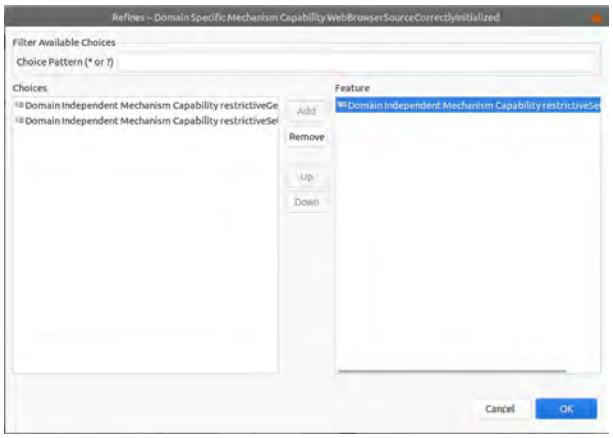

Figure 7.23: Use of the CIAAA library mechanism to set the Refine relation of the domain specific Library

7.6 Property Verification

In this section, we propose a methodology for verifying both functional and non-functional properties of system models. This verification process is carried out using Event-B models within the Rodin platform.

Rodin supports formal verification by automatically generating proof obligations (POs) based on the defined Event-B specifications. These POs represent logical conditions that must be satisfied to ensure the correctness of the modeled system. The platform is equipped with a set of SMT solvers and also provides an interactive interface that allows users to manually guide or discharge complex proof obligations when automation is insufficient.

By integrating this verification step into our methodology, we ensure that both functional behaviors and security-related properties (e.g., confidentiality, integrity, and authenticity constraints) are formally validated at an early stage of the development process.

7.6.1 Metamodels Interpretation in Event-B using Rodin

In Section 6, we presented an interpretation of the modelization using the Event-B formalism. These models were implemented using Rodin.

Refinement Tree

Each Event-B model in the refinement tree illustrated in Figure 6.1 is implemented in Rodin as a pair of `machine` and `context` components. These components are stored in multiple XML files, each capturing distinct aspects of the model. One XML file contains the structural and behavioral specification of the model, while another encodes the corresponding proof obligations (POs).

This separation of concerns at the file level enables modifications to be made to the model definition without directly affecting the associated proof obligations, and vice versa. Although these files are hidden from the user in the Rodin environment, the platform provides a set of integrated editors and views that allow seamless manipulation and navigation of both the model and its verification artifacts.

Rodin Views

Rodin offers multiple integrated views to facilitate the development and manipulation of Event-B models. Initially, the models are created using the default perspective, which provides a dedicated Integrated Development Environment (IDE) for editing both `machine` and `context` components. This environment allows users to define events, variables, invariants, sets, and constants in a structured and user-friendly interface, supporting the development of correct-by-construction formal models.

Rodin provides different views to manipulate models. First we implement the models in the default view that contains machine and context IDE. This is done using the

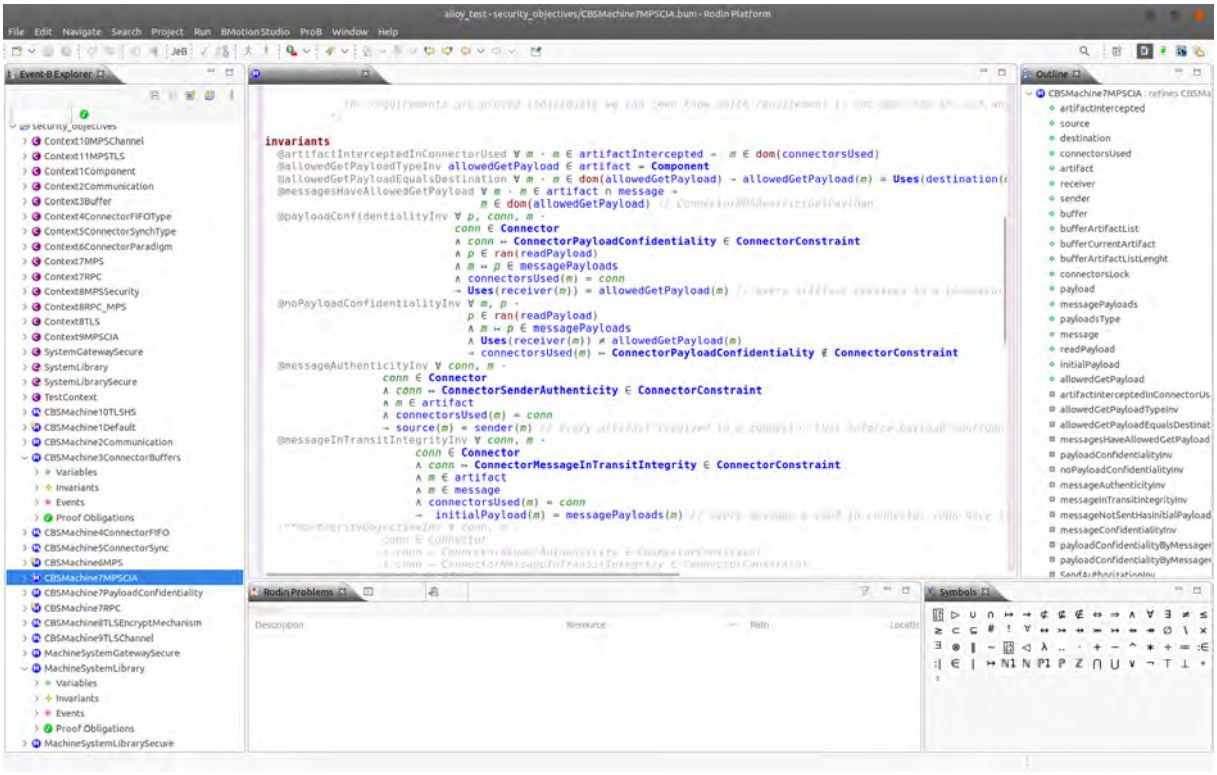

Figure 7.24: Rodin machine IDE perspective with CamilleX

CamilleX editor that creates text representation of the Event-B model (See Figure 7.24).

Finally, the ProB add-on can be integrated into the Rodin platform to enable simulation and model checking of the specified Event-B models. ProB provides an animation engine that allows users to explore the possible execution traces of their models interactively. This feature is particularly useful for validating the dynamic behavior of the system and gaining insights into its execution without requiring full formal proofs. Thus, ProB significantly enhances the usability and confidence in the correctness of Event-B models by providing an alternative and complementary form of formal verification.

In our work, we utilized the ProB tool to simulate the system and thereby validate the rules implemented for both functional and non-functional requirements. Simulation allows us to interactively explore execution traces of the system and verify that the defined behavior aligns with the expected properties of a Component-Based Systems. For instance, Figure 7.25 simulates a scenario in which a message is sent from the *Administrator* component to the *Database*, and observe that all triggered events follow a behavior that we deem acceptable according to the specified constraints.

This capability enables early validation of the system architecture by demonstrating its dynamic consistency. When combined with the formal proof obligations generated by the Rodin platform, the use of simulation provides a robust verification framework. This dual approach—simulation for behavioral validation and proof for logical correct-

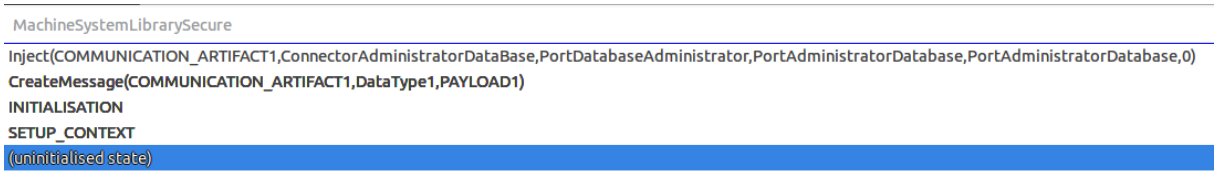

Figure 7.25: Event trace to Send a message from the Administrator and the Database in ProB

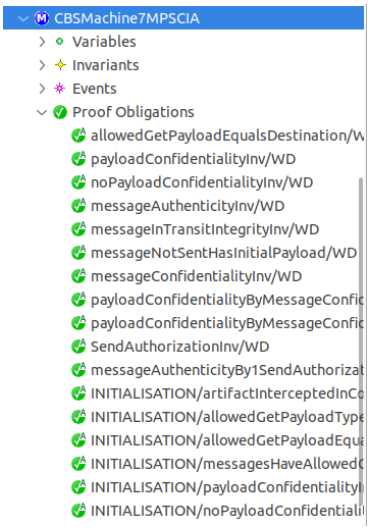

Figure 7.26: Rodin proof obligations list for the CIAAA machine

ness—strengthens the confidence that the Event-B model accurately represents a correct and secure Component-Based Systems.

Proof Assistant

In addition to the modeling view, Rodin provides an *interactive proof view*, which is essential for the verification of proof obligations (POs). Once a model is specified, Rodin automatically generates a set of POs that must be discharged (see Figure 7.26) to ensure the correctness of the model with respect to its invariants and refinement relationships. The interactive proof view allows users to manually guide the proof process in cases where automatic provers are insufficient. It displays the proof obligations and enables users to apply proof rules, tactics, and strategies interactively (see Figure 7.27). This view is particularly useful for handling complex proofs and for understanding why certain properties are or are not provable. The combination of automatic and interactive proving facilities supports a wide range of users, from beginners to formal methods experts, in constructing and verifying robust system models.

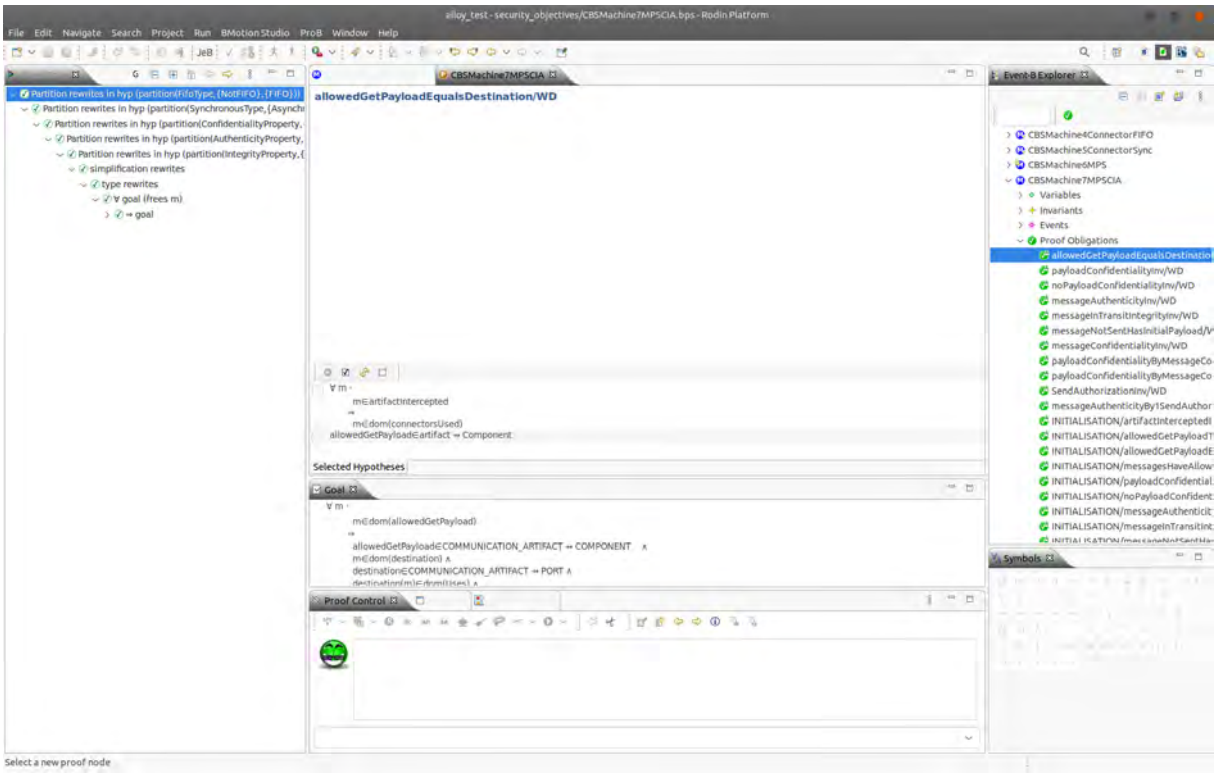

Figure 7.27: Rodin proof obligations interactive solver perspective used to discharge proofs that cannot be solved by automatic solvers.

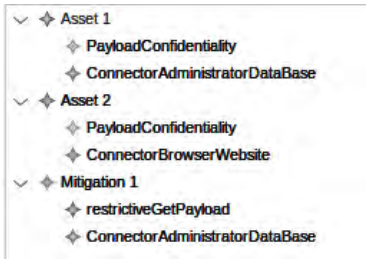

Figure 7.28: Assets and mitigations of the Webserver system without *restrictiveGetPayload* between the Browser and the Website

7.6.2 Feedback from the Formal Analysis to the Model Design

Once the Event-B model of an architecture is imported into Rodin, the associated proof obligations, particularly those derived from theorems, are automatically generated. The status of these proof obligations serves as the primary indicator of the validity of the declared properties. A *proven* status signifies that the property holds true given the current specification and assumptions. Section 6.5 showed an example of an instantiation of a system and the proof process associated with it.

As illustrated in Steps A2.4 and A2.5 of Figure 7.1, this verification process is only *partially automated*. While Rodin provides automated tools capable of discharging many proof obligations, some of the more complex ones may require manual intervention. Nevertheless, Rodin includes interactive proving features that can assist even non-expert users in completing the remaining proofs, thereby supporting a semi-automated verification workflow.

The status of the PO in the formal analysis process serves as critical feedback for the architectural model design. When a PO cannot be discharged automatically, it must be handled manually. However, the analysis cannot easily distinguish between a PO that is unprovable and one for which no suitable proof strategy has yet been applied. In such cases, the assistance of a formal methods expert is often necessary to determine whether the issue stems from the model itself or from the proving approach.

After every step is done if the proofs cannot be discharged an iterative modification of the model can take place until the system display the properties required with the mechanisms in place.

Example using the Webserver system

Suppose that, during the modeling of the web server system, the architect omits the use of the *restrictiveGetPayload* mechanism. As a result, the defined assets and mitigations for the system are those shown in Figure 7.28. In this configuration, there are two confidentiality-related assets associated with the payload, but only a single mitigation is applied.

In this scenario, when the architect performs the system analysis, a theorem's PO is not automatically discharged (see Figure 7.29). However, this status alone does not

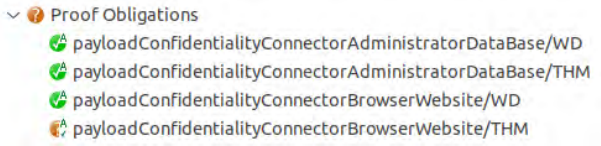

Figure 7.29: Status of the POs for the Webserver when the system is not secure

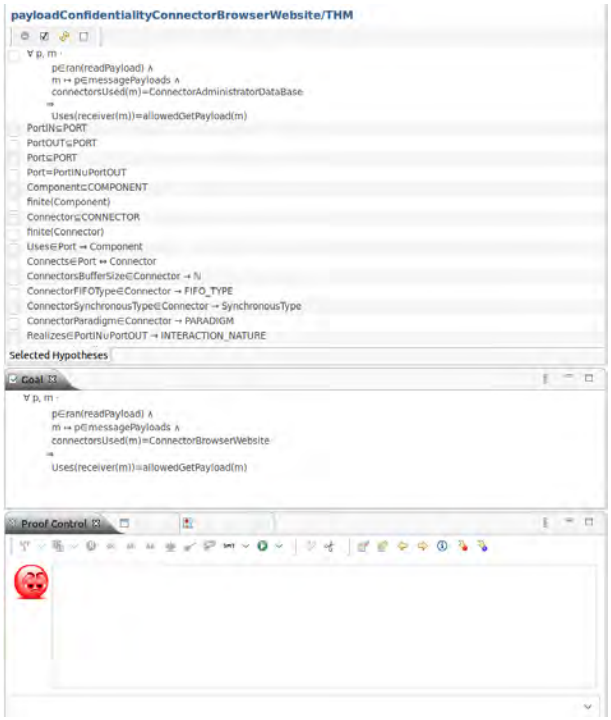

Figure 7.30: Interactive prover interface for the PO that cannot be automatically proven.

definitively indicate that the property is unprovable. At this stage, it is necessary to use the interactive prover of the Rodin platform to analyze which portion of the proof fails given the current set of hypotheses. Figure 7.30 illustrates the user interface of the interactive prover for the unresolved proof obligation. This interface displays the current set of hypotheses at the top, the goal to be proved in the middle, and the status of the PO at the bottom. At this stage, expert intervention is typically required to interpret the proof context, determine whether the obligation is fundamentally unprovable under the current hypotheses, or identify whether the proof can be completed manually through additional reasoning steps.

Once it is determined that a system property cannot be satisfied due to the system's design, the specific PO that failed to discharge provides valuable diagnostic information and feedback that can guide the architect to improve their model. In this example, the unresolved obligation is *payloadConfidentialityConnectorBrowserWebsite/THM*, indicating that the confidentiality property for the payload exchanged between the *Browser* and the *Website* is not satisfied. Within the Event-B model, properties are expressed as the-

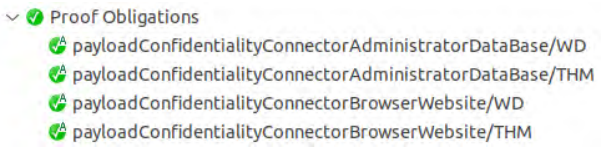

Figure 7.31: Status of the POs for the Webserver when the system is correctly initialized

orems, which implies that their proofs can only rely on information previously defined in the model. Consequently, the inability to discharge the proof obligation indicates that a supporting mechanism must be declared earlier in the development process in order to definitively establish the theorem. As such, the analysis of the theorem's PO reveals that an additional mechanism is required to satisfy the corresponding property. This conclusion stems from the principle that properties are intended to be satisfied through the introduction of appropriate mechanisms. In our Event-B model, the system behavior is deliberately kept as unconstrained as possible, such that only the explicitly defined constraints can guide the system toward satisfying the specified properties. In this example, the variable set *allowedGetPayload* is present in the proof goal (see Figure 7.30) meaning that we require a mechanism link to this element of the model to satisfy the property. At this stage, an iterative refinement process is initiated to address the violation. This involves introducing an appropriate mitigation mechanism, guided by the *Enforces* relation that links the SystemPolicy to the DomainSpecificMechanismCapability that gives a model representation between the property and the constraint associated with it (i.e. *payloadConfidentiality* and *restrictiveGetPayload*). In this example, the relation can be observed in the secure Webserver library presented in Figure 7.21, which was designed in coordination between the security expert and the system architect using the CIAAA Library.

When the system is correctly configured by adding a mitigation linking the *ConnectorBrowserWebsite Connector* with the *restrictiveGetPayload* DomainSpecificMechanismCapability as depicted in Figure 7.18, the corresponding Event-B machine reveals that all proof obligations (PO) are automatically discharged. This outcome indicates that all the specified properties for this configuration are satisfied, as shown in Figure 7.31.

7.6.3 System Verification

In Table 6.3, we gave the count of proof obligations (PO) generated for each refinement level. Table 7.1 shows how the POs were discharged using Rodin. Some have been discharged automatically. Others required interactive proof through explicit case analysis and/or hypothesis instantiation and rewriting. It also presents the count of *Events* and *Guards* used to specify both the properties and behavior of the system. The Library machine represent the machine generated with the tool.

Table 7.1: Analysis results of the Event-B model in Rodin with the Proof obligations discharged method

Machine	Events	Guards	Automatic PO	Interactive PO
2 Communication	3	11	27	0
3 Buffers	3	14	22	6
4 FIFO	3	4	21	5
5 Synchro	3	1	4	0
6 MPS	4	12	42	0
7 RPC	5	13	35	0
Authenticity	5	1	14	0
Payload confidentiality	5	1	24	0
Message confidentiality	5	1	16	0
Integrity	5	1	5	0
Library	4	0	5	0

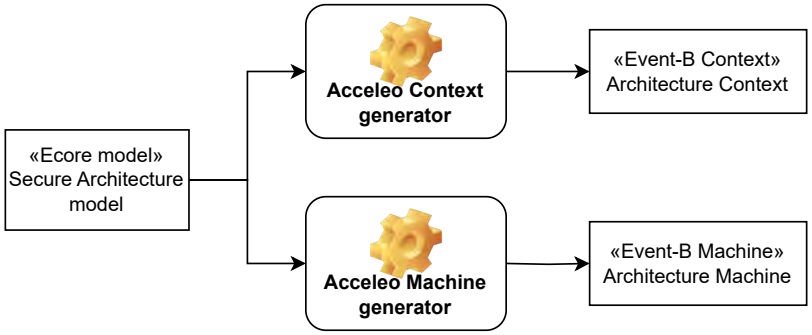

Figure 7.32: Acceleo transformation from the architecture model to a text representation of the Event-B model

7.7 Model Transformation

Once the model is described using the DSL, we will use Acceleo and the Eclipse modeling toolbox to generate the Event-B equivalent of the system that we can use. This Event-B code is presented as a text file that can then be manually inserted into the CamilleX editor that will translate it into a set of files that are usable by the Rodin editor.

```
1 constants [for (e ∈ ArchitectureElement | aSecureSystem·architecture·elements)][e·name/] [/for]
      [for (comp ∈ Port | aSecureSystem·architecture·elements→selectByType(Component)·uses)][
      comp·name/] [/for]
```

Listing 7.1: Acceleo generator Ecore to Event-B Context: Declare all variables

This transformation takes the Ecore model of the secure architecture and turns it into a Machine and a Context (See Figure 7.32) using two Acceleo generators following the transformation rules presented in Chapter 6 as shown in Table 7.2. Listing 7.1 shows the

Table 7.2: Acceleo transformation summary

Model element	Event-B interpretation	Transformation
Structural element	Create a constant and putting it into the corresponding set	Listings 7.1 and 7.2
Element relation	Creating an entry in the corresponding relation	Listing 7.3

Acceleo generator line taking every architectural element and declaring them as *constants* of the system.

Listing 7.2 visualizes an example of the definition of the *Component* set with all the *Components* defined in the system Ecore model. All constant sets are defined similarly as a mapping to all the architectural elements of the system (i.e. *Ports*, *Connectors*, *Data*, ...)

```
1 @axmFillSystemComponents partition(Component, [for (comp ∈ Component | aSecureSystem·
    architecture·elements→selectByType(Component)) separator (',')]{[comp·name/]}[/for])
```

Listing 7.2: Acceleo generator Ecore to Event-B Context: Fill Constant with components

Listing 7.3 illustrates the population of the *realize* relation constant, which links each *Port* to its corresponding *Component*.

```
1 @axmFillRealizes partition (Realizes, [for (p ∈ Port | aSecureSystem·architecture·elements→
    selectByType(Component)·uses) separator (',')]
2       [for (d ∈ Data | p·realizes→selectByType(Data)) separator (',')]
3
4       {[p·name/] ↦ [d·name/]}[/for]
5   [/for])
```

Listing 7.3: Acceleo generator Ecore to Event-B Context: Relation transformation

Finally, Listing 7.4 complete the context with the relation between each *Connectors* and their constraints following the *Mitigations* of the secure system as shown in Section 6.6.

```
1 @axmFillConnectorProperty partition (ConnectorConstraint,
2       [for (m ∈ Mitigation | aSecureSystem·elements→selectByType(Mitigation)) separator (',
    ')]
3       [if (m·element→selectByType(Connector) ≙ null)]
4       [for (dimc ∈ DomainIndependentMechanismCapability | m·constraint→selectByType(
    DomainSpecificMechanismCapability)·refines) separator (', ')]
5         {[m·element·name/] ↦ [dimc·name/]}[/for][/if][/for])
```

Listing 7.4: Acceleo generator Ecore to Event-B Context: Connector constraints

Currently only the properties and constraints presented in Chapter 6 can be translated into the Event-B model. If a property or constraint that is not implemented is added to the secure architecture model, then an error will be raised during the Event-B translation, preventing compilation and the creation of proof obligations.

Chapter 8

Evaluation of the Proposed Methodology

Contents

8.1 Introduction

In this chapter, we present the evaluation of the proposed approach through a representative case study: a smart metering system 2.8.2. This example was chosen to illustrate the applicability and versatility of the proposed Domain-Specific Language, its formal representation using Event-B, and the supporting toolchain. The validation process is carried out using formal verification tools such as ProB, which allows for model checking, animation, and constraint solving of Event-B models. Through this case study, we demonstrate how the toolchain, metamodels, and generated formal artifacts can be effectively used to verify both functional and non-functional properties, including security objectives and system constraints. This chapter answers **RO5**.

This Chapter is organized as follows, Section 8.2 present the usage of the toolchain through the smart meter gateway example from the modelization using the DSL to the generation of the Event-B model. Section 8.3 presents the formal verification and validation of the generated Event-B code using Rodin. Finally, Section 8.4 opens the discussion of the approach for its application, generalization and the limitations of the resulting toolchain.

8.2 Case Study modeling

We use the Smart Meter Gateway case study 2.8.2 as an illustrative instantiation of our proposed approach. This case study provides a concrete and relevant example to demonstrate how our framework supports the design, modeling, and formal verification of secure component-based software architectures. The approach aims to enhance existing methodologies by integrating MDE techniques with formal methods, offering a structured and rigorous development process. By applying our toolchain and methodology to this real-world scenario, we illustrate how system security requirements and constraints can be explicitly captured, linked to architectural elements, and formally verified using Event-B. This instantiation also showcases the benefits of separating concerns across functional and security domains, allowing domain experts to focus on their areas of expertise while ensuring system-wide coherence and security assurance.

8.2.1 Expressing the Architecture

We begin by defining the system architecture through the use of the different modeling views introduced in Section 7.4.1. Figure 8.1 illustrates the model corresponding to the *Structural* view, which captures the core architectural elements of the system, including components, ports, and connectors. This view establishes the static structure of the system and serves as the foundation for interaction modeling. In parallel, the *Data* view is employed to describe all elements related to data interactions, such as datas and datatypes. This abstraction is depicted in Figure 8.2, highlighting how the system handles and exchanges information through defined paradigms.

In this system, the *Smart Gateway* component is positioned centrally, acting as a hub that connects all other components. For this example, four meters are connected via a shared secure connector, while an additional meter is linked through a separate, unsecured connector. At this level of abstraction, security aspects are not yet explicitly represented and must be integrated at a later stage of the modeling process. The *Interactions* within this system remain simple, with each connector transmitting a single datatype. This design reflects the logical separation enforced by the *Smart Gateway*, which isolates and channels data flows between components based on their roles and security requirements.

8.2.2 Security Analysis

In the following steps, we enrich the system model with security-specific information by incorporating security properties and constraints. To achieve this, we leverage a predefined library of security requirements and policies that are structured according to the well-established CIAAA model focused on Confidentiality, Integrity and Authenticity. This library, illustrated in Figure 7.20, is reused from the web application model presented in Chapter 7 and provides a reusable and systematic set of security elements that can be applied to various system architectures.

These properties and constraints are applied to the system using the mitigation and

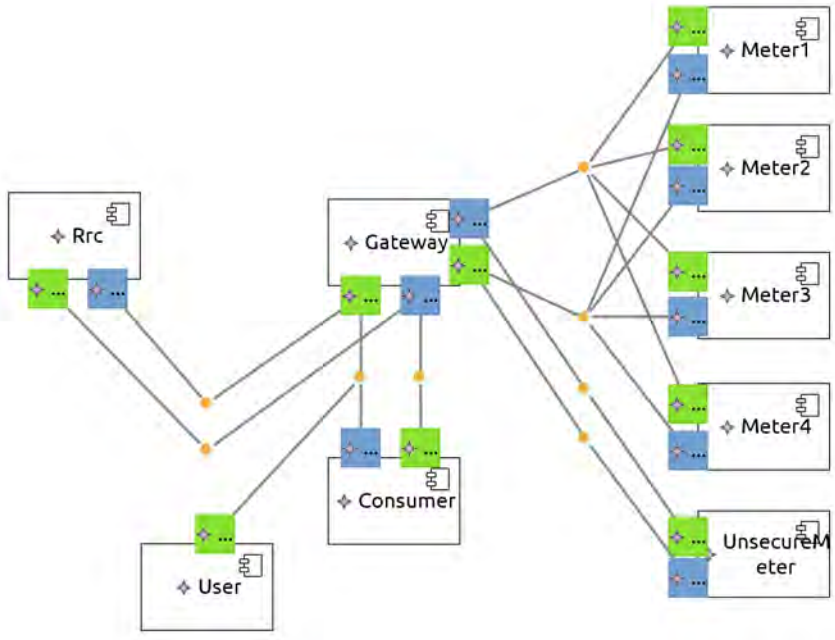

Figure 8.1: Description of the smart meter case study *Structural* view using Sirius

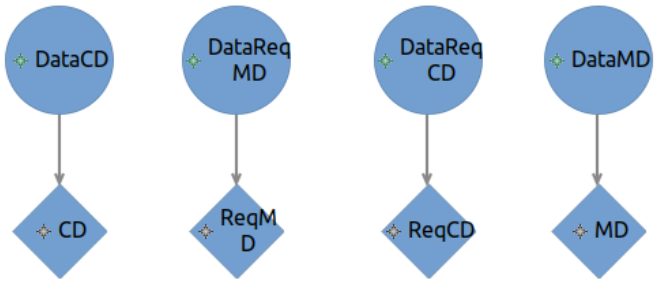

Figure 8.2: Description of the smart meter case study *Data* view using Sirius

Figure 8.3: Tree-based view of the assets and mitigations for the Smart Meter gateway case study implementing Confidentiality, Integrity and Authenticity properties and constraints to the $ConnectorGatewayMeterMD$ Connector

assets view that links the security properties and constraints to the system architecture. Figure 8.3 shows the assets and mitigations applied to the $ConnectorGatewayMeterMD$ Connector to apply three constraints and expecting three properties in the form of DomainSpecificMechanismCapability and SystemPolicy. The $ConnectorGatewayMeterMD$ is the connector linking the *Smart Gateway* and the four meters for the transmition of the metering datas. These properties and constraints aim to answer the S_Req defined in Section 2.8.2 for the metering data.

8.2.3 Generate Event-B Models

Using the Acceleo generators presented in Section 7.7, we automatically generate the corresponding Event-B `machine` and `context` components. The acceleo generator takes the secure architecture model and use the translation rules presented in Section 7.7 to create a textual representation of the Event-B model than can then be inserted into a CamilleX editor in the Rodin platform to create the Event-B model and generate the corresponding Proof Obligations.

The Event-B model generated for this system produced 10 PO associated to the properties that were all discharged automatically thanks to the constraints.

8.3 Model Verification and Validation

In this section, we present an experiment using the smart meter gateway example where a User that has gained access to the connector connecting the Consumer and the Gateway attempts to impersonate the Gateway (see Figure 8.4). In our modeling, the User attempting to impersonate the Gateway is represented through an Inject event where the source of the message is set to Consumer and the actual sender is set to User. We use the security constraint *restrictiveSetSource* to protect against this behavior. We consider two scenarios: (1) the connector enforces the constraint, and (2) the connector does not enforce the constraint.

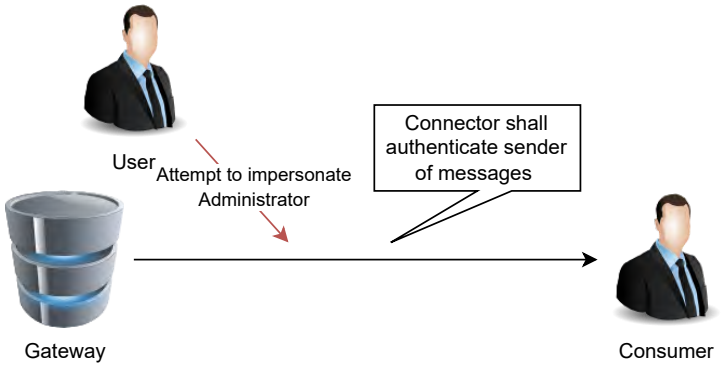

Figure 8.4: User trying to impersonate the Gateway on a connector enforcing message authenticity

8.3.1 Event-B Verification

For the Event-B formal modeling, the Gateway structural model is defined as extension of the *CBSE MPS CIAAA Event-B context* and the Gateway behavioral model is defined as a refinement of the *CBSE MPS CIAAA Event-B model*.

```
1 context SystemGatewaySecureContext extends Context9MPSCIAAA
2
3 constants CD MD ReqCD ReqMD DataCD DataMD DataReqMD DataReqCD Gateway Meter1 Rrc
      Consumer ConnectorGatewayMeterMD ConnectorGatewayMeterReqMD ConnectorGatewayRrcCD
      ConnectorGatewayRrcReqCD ConnectorGatewayConsumerCD ConnectorGatewayConsumerReqCD
      Meter3 Meter2 Meter4 UnsecureMeter ConnectorGatewayUnsecureMeterMD
      ConnectorGatewayUnsecureMeterReqMD User ...
4 axioms
5   @axmFillSystemComponents partition(Component, {Gateway}, {Meter1}, {Rrc}, {Consumer}, {
      Meter3}, {Meter2}, {Meter4}, {UnsecureMeter}, {User})
6   ...
7   @axmFillSystemConnects partition(Connects,
8   {PortGatewayMD ↦ ConnectorGatewayMeterMD},
9   {PortMeter1MD ↦ ConnectorGatewayMeterMD}, ...)
```

Listing 8.1: Library structural model

The *ConnectorMessageAuthenticity* property in this model is verified when the behavior of the corresponding connector is constrained by *restrictiveSetSource* (Scenario 1), by construction thanks to the Event-B refinement mechanism. The resulting Gateway system behavior satisfies the security properties established in the *CBSE MPS CIAAA Event-B model*.

From the security perspective, in Scenario 1 (see Listing 8.2) the *ConnectorGatewayConsumerCD* linking the Consumer to the Gateway satisfies the *ConnectorMessageAuthenticity* property (line 2-3) while Scenario 2 does not (line 6). This distinction can be done in the model representation of the system as the presence of the mitigation on the connector or not.

```
1 // Scenario 1
2   @axmFillConnectorProperty partition (ConnectorProperty,
3         {ConnectorGatewayConsumerCD ↦ ConnectorMessageAuthenticity})
4
5 // Scenario 2
6    @axmFillConnectorProperty ConnectorProperty = ∅
```

Listing 8.2: ConnectorProperty relation filled with and without the constraint applied

In addition, in the *CBSE MPS CIAAA Event-B model* we add a theorem representing the expected property of the connector between the Consumer and the Gateway (see Listing 8.3). The verification of this theorem shows that it is automatically discharged in Scenario 1 and that it cannot be discharged in Scenario 2, which verifies the intended property.

```
1 theorem @messageAuthTheorem ∀ m ·
2         m ∈ artifact
3         ∧ connectorsUsed(m) =
4         ConnectorGatewayConsumerCD
```

Listing 8.3: Gateway theorem

8.3.2 Validation by Animation

Moreover, to validate the resulting system model and for illustration purposes, we used ProB for simulation. In our experiment, ProB allows consistency checking of Event-B machines displaying all the states that were explored until the invariant violation was found. The goal is to demonstrate that the application of the constraint in the machine constrains the operation of the system to satisfy the corresponding security objective, while the absence of the constraint can lead to the violation of the corresponding security objective.

In these two scenarios, after creating a message to be sent, ProB allows to list all the possible parameters that an event can take to be triggered as depicted in Figure 8.5 and Figure 8.6. In these figures, the columns *conn*, *d*, *s* and *aSender* represent the used connector, the destination of the message, the source of the message and the actual sender.

The trace of the two scenarios is executed as follows: first we initialize the model with the INITIALISATION event, as depicted in Listing 8.4. It sets the default value

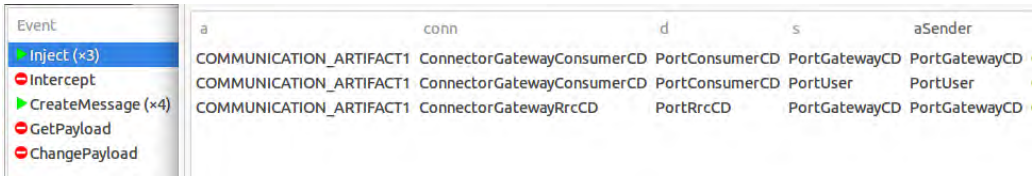

Figure 8.5: Possible events for Scenario 1

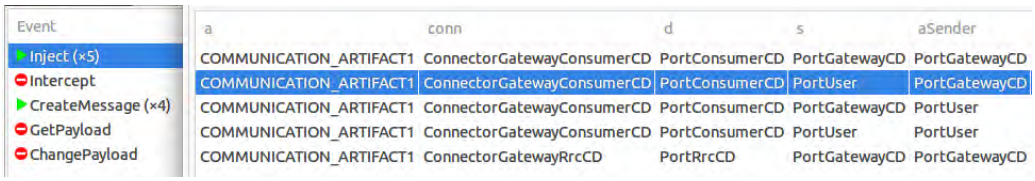

Figure 8.6: Possible events for Scenario 2

of all the variables declared in the machines and sets the basis for recursive proof of the model. Listing 8.4 show the initialization of the variable used in the Authenticity invariant depicted in Listing 6.12.

```
1 event INITIALISATION
2 then
3     @sourceInit source := ∅
4     @senderInit sender := ∅
5     @connectorsUsedInit connectorsUsed := ∅
6     @ artifactInit   artifact  := ∅
7     ...
8 end
```

Listing 8.4: INITIALISATION event

After the initialization, a message is created using the CreateMessage event. This message has a datatype that is usable in the connector between the Consumer and the Gateway. It also contains a payload representing the content of the message.

Finally, the Inject event can be triggered with different parameters depending on the scenario. In Scenario 1 (Figure 8.5), the only possible parameters for the Inject event are the ones where the source of the message is the actual sender. Leading to the satisfaction of the property. In Scenario 2 (Figure 8.6), where the constraint is not applied, two additional set of parameters are possible compared to Scenario 1. The set of parameters in blue shows a possibility where the User can impersonate the Gateway, the source is referencing the Gateway while the actual sender is the User violating the property.

8.3.3 Results

The realization of this use case serves as a validation of our proposed approach by demonstrating its applicability to a realistic system architecture. The generated model is composed of ten components, eight connectors and 20 ports. It also created ten theorems that in turn generated ten POs. The generated POs encapsulate both structural and security-related theorems that must hold for the system to be considered correct.

Non-expert				Expert			
System design	Security analysis	Formal modeling	Formal analysis	System design	Security analysis	Formal modeling	Formal analysis
20h	10h	5h	10h	10h	4h	1h	1h

Table 8.1: Anticipated time taken for the every steps of the approach for expert and non-expert

Writing model		Acceleo transformation	
Non-expert	Expert	Non-expert	Expert
5h	1h	<1sec	<1sec

Table 8.2: Comparison between Acceleo transformation and manually writing the Event-B interpretation of the model

The evaluation highlights a key advantage of the proof-based approach: the number of proof obligations remains stable as additional meters are introduced into the system, provided that the new components connect to existing secure connectors. This contrasts significantly with model-checking approaches, where each new component expands the state space, potentially leading to state explosion. Our method thus offers better scalability and predictability in the verification process, especially for systems with modular or replicated components.

Tables 8.1 and 8.2 present our estimation of the time required to model a system and conduct a security analysis using our proposed toolchain. While the estimated modeling time is naturally higher for non-experts compared to expert users, the results demonstrate that our approach enables non-expert architects to describe and analyze secure systems with the support of formal methods. This highlights the usability and accessibility of the framework, even for users without prior expertise in formal verification or security analysis.

The conception and realization of this approach and the associated toolchain is the result of a four year work. During that time we also produced published papers based on the results. We estimate that the conception of the metamodels took in total 6 months of work with an iterative process to get to the final results. The realization of the Event-B model in association with the metamodel took a year with some trial and errors and some non-finished work that could not make it into this manuscript due to time constraint . Finally the realization of the EMF toolchain for the DSL with Sirius and Acceleo was an additional 6 months of work. The remaining time was spend on the redaction of papers and this manuscript as well that the constant process of litterature review trhough out the thesis.

The conception and realization of this approach, along with the associated toolchain, represent the outcome of four years of research.

The design and refinement of the metamodels required approximately 6 person months. The majority of this effort was spent on the iterative process to reach a stable and expressive specification. The development of the Event-B models, in conjunction with the metamodels, required 12 person months. The most important part of the effort was spent in an

iterative process to produce a reusable formal model with a high level of expressiveness. The rest of the time was spent on exploratory efforts, and partial implementations—some of which could not be included in this manuscript due to time constraints.

The implementation of the EMF-based toolchain for the DSM, including the integration with Sirius and Acceleo for model visualization and code generation, required 6 person months. This included the effort made to understand the usage of EMFT to specify metamodels in Ecore and to generate other representations.

The remaining time of this thesis was devoted to the preparation of publications, the writing of this manuscript, and continuous literature review conducted throughout the doctoral research.

8.4 Discussions

In this section, we present the application of the proposed approach as well as its generalization and limitations.

8.4.1 Applications of the Proposed Approach

The proposed approach offers significant benefits for both system architects and security experts. By enabling the detection and analysis of security concerns at an early stage of system development, it supports a proactive rather than reactive security engineering process. A key strength of this method lies in the clear dissociation between security *properties*—which describe high-level requirements and policies—and *constraints*—which define the mechanisms that enforce those properties within the system architecture. This separation facilitates a modular and traceable design process, whereby security constraints can be systematically applied to the architecture and verified against the original properties. The formal verification step ensures that the implemented constraints effectively fulfill the specified security requirements, thereby increasing the reliability and trustworthiness of the final system.

Additionally, the proposed approach leverages predefined libraries of security requirements and mechanisms. These libraries can be designed and maintained by security experts, enabling system architects—who may have limited security expertise—to integrate verified security mechanisms into their designs confidently. This clear separation of concerns enhances both the reusability of security components and the trustworthiness of the overall development process. Figure 8.7 shows how the proposed approach provides tools for every stakeholders for the modelization and verification of secure architectures.

Furthermore, our use of a proof-based formal representation adds a critical layer of assurance. By formally modeling and verifying the relationship between security properties and their enforcing constraints, we demonstrate that the system satisfies its specified requirements not only by design intent but also by formal proof. This contributes to a higher level of rigor and dependability in the system's security assurance.

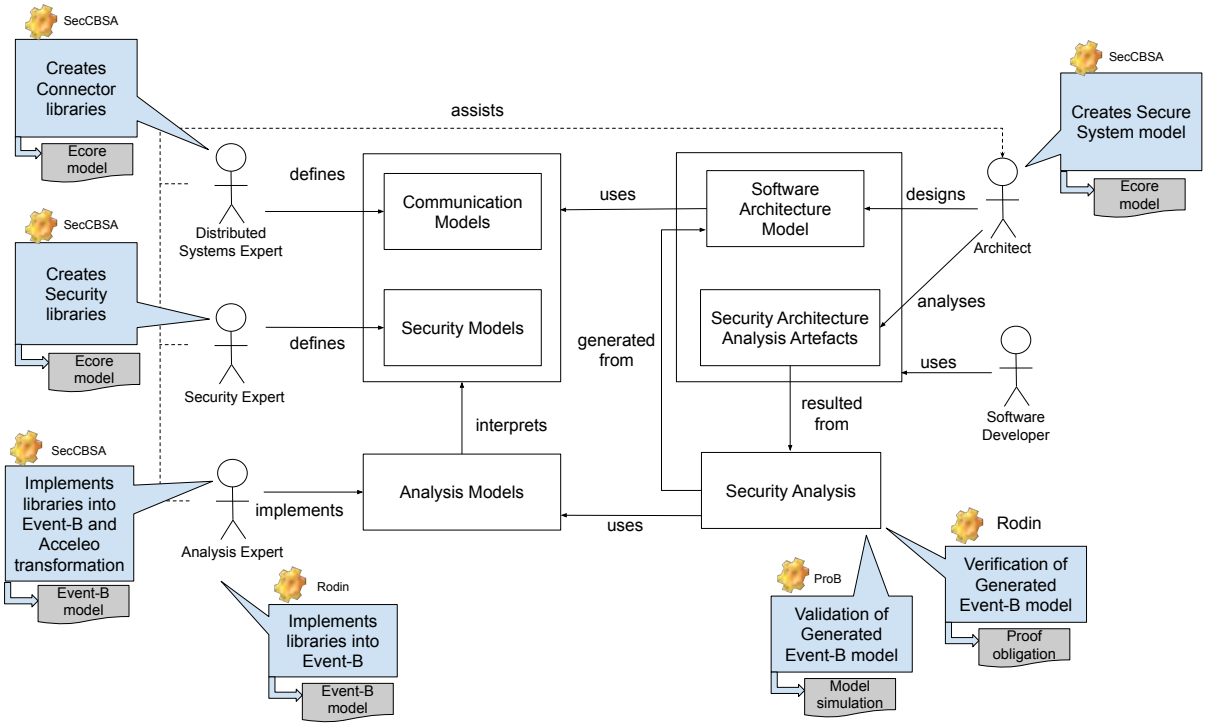

Figure 8.7: CBSE Process for Security engineering: An instance of the involved Stakeholders with the associated activities

8.4.2 Generalization

The realization of the smart gateway use case serves as a validation of our proposed approach by demonstrating its applicability to a realistic system architecture. The generated model in Section 8.2.3 is composed of ten components, eight connectors and 20 ports. It also created ten theorems that in turn generated ten proof obligations. The generated proof obligations encapsulate both structural and security-related theorems that must hold for the system to be considered secure.

In addition to the previous experiment, we developed an alternative gateway system comprising 100 meters (See Figure 8.8). This example is an extension of the model presented in Figure 8.1 with 100 meters instead of the four in the initial model. It was created by executing a Java program on the gateway model defined using the graphical syntax. The program duplicates the meters along with their respective ports and connects them to the appropriate connector. All of those models and codes are available in the public git repository. Although the graphical representation is not well suited for such a large-scale example we can still use the hierarchical representation of EMF to manipulate it as shown in Figure 8.8. This model provides a valuable indication of the scalability of the proposed approach. With the transformation presented in Section 7.7, the Event-B model is easily generated and the number of proof obligations remains at ten, demonstrating that the proof burden does not scale with the number of connected components and ports. This example demonstrates that the number of generated theorems and proof obligations remains the same as in the initial case study.

The evaluation and the scaled example highlight a key advantage of the proof-based approach: the number of proof obligations remains stable as additional meters are introduced into the system, provided that the new components connect to existing secure connectors. This contrasts significantly with model-checking approaches, where each new component expands the state space, potentially leading to state explosion. Our method thus offers better scalability and predictability in the verification process, especially for systems with modular or replicated components. Additionally, the generation of the system's formal interpretation in Event-B is fully automated, eliminating the error-prone and time-consuming manual process of creating the corresponding Context and Machine models.

The tool support developed for this approach provides a user-friendly environment that spans from the architectural design phase to the formal verification of system properties, all within a model-based framework that promotes reuse. This approach enables a clear separation of concerns between system design and formal verification by offering automated tools that seamlessly connect the two. It supports an incremental process in which the design phase automatically generates the corresponding formal artifacts, while the results of formal verification provide feedback to refine and improve the system design.

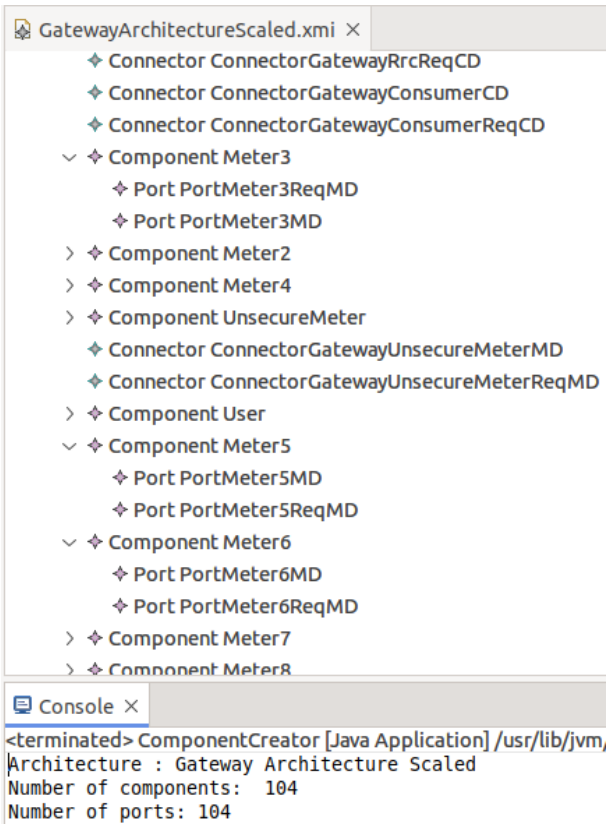

Figure 8.8: Hierarchical representation of the scaled gateway model, showing the calculated numbers of ports and components.

8.4.3 Integration in a SDLC

While this work has been demonstrated in the context of extending the traditional Royce waterfall model using the CIAAA model for security, the proposed approach is not limited to this specific development paradigm. The methodology can be generalized and adapted to alternative Software Development Life Cycles (SDLCs) and to a variety of security frameworks. In particular, the property and constraint modeling strategy could be applied in threat-centric security models by specifying *negative properties*—representing potential vulnerabilities or undesirable behaviors—and using *model checking techniques* to search for counterexamples, as opposed to the proof-based validation presented in this work.

Moreover, the iterative nature of the methodology aligns well with Agile development processes. With appropriate adaptation, the integration of formal methods and security views into Agile iterations can support continuous verification and early-stage threat assessment. This highlights the flexibility of our framework to fit both traditional and modern development approaches, broadening its applicability across diverse project environments and team structures.

This generalization of the approach makes it particularly valuable for practitioners who are familiar with a wide variety of Model-Driven Engineering (MDE) tools and formal verification techniques. It enables experts to integrate security analysis seamlessly into existing MDE workflows, fostering interoperability and reuse of formalized components and security libraries.

8.4.4 Tool Support : Automated Tools and Limitations

In this work, we employed a general-purpose Domain-Specific Language (DSL) designed to be adaptable to a variety of formal techniques. We chose Event-B as the underlying formalism due to the strength of its proof mechanisms and the comprehensive tool support offered by the Rodin platform. This decision enabled us to construct rigorous proofs establishing the relationships between certain properties and constraints, while also providing a formal representation of system architectures.

However, the adoption of Event-B introduced certain challenges. Our approach focuses on general software architecture modeling rather than the specification of a particular system, which diverges from the conventional use cases for Event-B. Notably, Event-B's requirement for a linear refinement hierarchy imposed structural constraints that hindered the modular reuse of unrelated elements and libraries. This limitation necessitated modeling all architectural components as part of a single refinement tree, thereby reducing flexibility and scalability.

Furthermore, our implementation faced difficulties in modeling time-based properties, such as *Availability*. This limitation arises partly from Zeno's paradox [152], which suggests that an infinite number of events could theoretically occur in a finite period of time. Event-B, as a formal modeling language, does not natively support the concept of time, which poses challenges when modeling and verifying time-dependent behaviors or properties. While efforts have been made to incorporate time-related constructs us-

ing set-theoretic encodings [153], these approaches often result in a significant increase in the complexity of the formal model. This added complexity introduces substantial overhead in terms of proof obligations, leading to what is often referred to as "proof noise"—the generation of numerous low-level obligations that do not contribute meaningfully to the primary verification goals. Consequently, while feasible, integrating temporal aspects within Event-B demands a high level of expertise and effort, which may hinder its practical applicability for time-critical systems.

As a result, accurately capturing temporal constraints within our current Event-B framework remains an open challenge. As such, the adoption of alternative formal techniques, such as model checking, stay the prefered approach for such properties where proof-based language are not as suited. Unlike proof-based methods, model checkers can explore system behaviors exhaustively within bounded parameters, which may prove particularly useful for handling temporal properties, availability constraints, or dynamic system configurations. Incorporating such techniques into the toolchain could complement the strengths of Event-B and offer a more comprehensive verification framework.

Chapter 9

Conclusion

Contents

In this chapter, we provide a comprehensive summary of the main contributions presented in this thesis. Additionally, we discuss potential directions for future research that could extend and enhance the results and methodologies proposed in this work.

9.1 Summary

Security is increasingly recognized as a critical quality attribute in modern software-intensive systems, attracting significant attention from both the research and industrial communities. In this work, we proposed a methodological approach to support security modeling and analysis within the context of component-based software architecture. Our approach addresses the positive aspect of security, such as security objectives, requirements and policies. It emphasizes the role of the software architecture model, the underlying communication model, and the explicit representation security objectives, requirements and policies as first-class modeling entities. This enables the specification and analysis of applications within specific domains. Furthermore, our method promotes a design philosophy centered on reusability, facilitating the systematic engineering of secure software system architectures.

This approach has the potential to significantly reduce the cost of engineering secure systems by allowing security concerns to be addressed early in the software development lifecycle—specifically during the architectural design phase—while abstracting away low-

level technical complexities for the developer. The process begins with the specification of a conceptual model that represents both the target system and its associated concerns. This is followed by the formulation of a logical specification of the system architecture and computational model, wherein concerns are expressed as properties using first-order and modal logic. These formalism choices provide a high-level, technology-agnostic means of representation.

Subsequently, tailored modeling languages and their interpretations are developed using tooled formal languages that support the intended design and verification tasks. The outcomes of these efforts are leveraged to formally define models of security objectives, as well as corresponding solutions—such as security mechanisms. Ultimately, this work contributes to establishing a systematic bridge between abstract security models and concrete system implementations.

In Chapter 4, we proposed a methodology for the creation of a design and analysisframework and provided some guidelines on how the resulting methodology may be used to achieve security within exiting SDLC methodologies (answering part of **RO1** and **RO2**).

In Chapter 5, we present a model-based representation of component-based systems (CBS) incorporating different communication paradigms, namely Message-Passing Systems (MPS) and Remote Procedure Calls (RPC) (Answering **RO1.1**). Building on this foundation, we propose an extension of the metamodel to capture both functional and non-functional requirements through the definition of properties and constraints (Answering **RO1.1**). Finally, we introduce a further extension dedicated to security requirements, which encompasses both domain-independent and domain-specific properties and constraints (Answering part of **RO2** and **RO3**).

In Chapter 6, we present a formal interpretation of the proposed metamodels using the Event-B formal method. This interpretation leverages the refinement mechanism of Event-B to incrementally introduce the structural and behavioral details of the metamodels (Answering **RO1.2**). The objective of this formal representation is to promote reusability by enabling the definition of reusable connector structures, along with libraries of both functional and non-functional properties and constraints. We also provide a formal representation of the security concepts introduced in the previous chapter, capturing both domain-independent and domain-specific security properties and constraints. This formalization enables the rigorous analysis and verification of security aspects within the system architecture, ensuring that the modeled requirements can be validated through proof-based techniques (Answering the rest of **RO2** and **RO3**).

In Chapter 7, we propose a complete toolchain that forms a comprehensive framework for the specification and validation of CBS. This framework is built upon the Eclipse Modeling Framework (EMF) and integrates tools such as Sirius and Acceleo to provide graphical modeling and model-to-text transformation capabilities. The toolchain is complemented by the Rodin platform, which enables formal verification using the Event-B method. Together, these tools establish an integrated environment where both the architectural models and their corresponding formal verification can be developed, analyzed, and validated within a unified workflow. This chapter integrates **RO1**, **RO2** and **RO3**

while addressing **RO4**.

Finally, Chapter 8 presents an evaluation of the proposed approach through a case study based on a *Smart Meter Gateway*. This evaluation aims to validate each step of the methodology, as well as the integration and effectiveness of the supporting toolchain. Through this example, we demonstrate that the approach is applicable to a real-world scenario and can be generalized to a broader range of CBSE projects (Answering **RO5**).

9.2 Limitations and Future Work

In this section, we consider the limitations and future work of different parts of the approach.

1. *Property Relationship.* The representation of security property relationships can be significantly enriched by introducing a theoretical framework that formalizes the various types of interactions between properties. To this end, we propose an extension of the property metamodel that incorporates new constructs for capturing such relationships. These constructs may include semantic types such as *implication*, *conflict*, *complementarity*, or *composition*. Each type of relationship provides valuable insights into how properties influence, reinforce, or contradict one another.

   To maintain tool and language independence, the semantics of these relationships should be described in a language-agnostic manner, aligned with the semantic principles discussed in Section 6.2.1. By adopting this approach, we can develop a reusable and extensible library of security properties that are grouped and categorized based on their interrelationships.

   For instance, if a system architect specifies a general requirement for *confidentiality*, the framework could automatically suggest the use of more specific properties such as *payload confidentiality* or *message confidentiality*, based on an implied relationship. Similarly, in a scenario where a requirement for *role-based access control* is specified, the system could recommend a composition of *source authenticity* and *authorization* properties, since together they fulfill the access control requirement.

   Such a framework not only enhances the expressiveness and usability of the property model but also provides architects and security experts with intelligent guidance for the selection and composition of security properties, fostering a more structured and consistent security design process.

2. *Time based properties.* The Event-B formalism proved to be ill-suited for the representation and verification of time-based properties, such as *availability*. During our implementation attempts, we encountered limitations primarily rooted in the challenges associated with modeling time in Event-B. In particular, we faced difficulties related to Zeno's paradox, which arises from the theoretical possibility of an infinite number of events occurring within a finite time frame. This paradox highlighted the

inadequacy of Event-B's refinement-based approach for expressing temporal properties in a precise and verifiable manner. To address this limitation, we concluded that model checking techniques offer a more appropriate formalism for modeling and verifying time-sensitive constraints. Tools such as *ProB*—which supports both animation and model checking of Event-B models—or other formal languages with native support for temporal logic may provide viable solutions. These tools allow the expression of temporal properties and enable the exploration of execution traces to detect violations of availability constraints. Therefore, integrating model checkers with the proposed framework may be a promising direction for extending the verification capabilities to encompass time-based security properties.

3. *Tool support.* The tool support presented in Chapter 7 presents an acceptable level of automation, but it could be improved in several key areas.

   (a) *Results report.* Currently, the verification status of the system is closely tied to the Rodin platform, particularly through its native interface for managing and displaying proof obligations. While this integration offers a seamless development and verification environment, it limits the portability and accessibility of verification results outside of Rodin. To address this limitation, an effective enhancement would be to extract the results of the proof obligations and generate structured reports.

   These reports could provide a comprehensive summary of the verification status, explicitly indicating which security and functional properties have been successfully proven and which remain unverified. Additionally, the report could include diagnostic information about failed or unprovable properties, and propose corresponding mitigation strategies or mechanisms that could be integrated into the system model to address the identified gaps. Such recommendations could be drawn from a predefined library of security mechanisms aligned with the properties, allowing the tool to suggest targeted improvements.

   This decoupling of the verification status from the Rodin interface would not only improve traceability and documentation of the verification process, but also facilitate communication between system architects, security experts, and other stakeholders who may not be familiar with formal verification environments. Moreover, it would support automated reporting pipelines and continuous integration of formal verification into the software development lifecycle.

   (b) *Reuse.* The reuse enabled by the property and constraint libraries facilitates a clear separation of concerns between domain experts, such as system architects and security specialists. These libraries encapsulate reusable knowledge, allowing security expertise to be captured once and applied across multiple systems. However, the current creation and usage of such libraries require familiarity with the Eclipse Modeling Framework (EMF), which presents a barrier to entry for practitioners without expertise in model-driven engineering.

A more user-friendly interface for the tool could significantly facilitate the creation of additional libraries. Furthermore, the establishment of clear guidelines for the development of these libraries would help ensure a consistent level of quality and maintainability across contributions.

4. *Approach Generalization.* We aim to explore new opportunities to apply the proposed approach across a broader range of application domains. This effort involves the instantiation of the complete software engineering methodology and supporting toolchain in diverse contexts, accompanied by the systematic evaluation of user experiences from multiple sectors. In doing so, we seek to validate the generalizability and adaptability of the framework.

   Furthermore, we plan to enhance the proposed approach by integrating it with other model-driven engineering practices, architectural frameworks, and security models—such as the ENISA threat taxonomy [154]. Our goal is also to support interoperability with a wider range of formal verification techniques, including Computation Tree Logic (CTL), and NuSMV. Finally, we intend to align our methodology with contemporary software development life cycles, such as Agile and DevOps, to ensure its practical applicability in modern development environments.

9.3 Perspectives

In this Section, we present the perspectives for comprehensive work based on the work presented in this thesis.

1. *Collaborative security libraries* To improve accessibility and foster broader participation, a potential enhancement would be the development of a collaborative, web-based platform for the creation, management, and sharing of security property and constraint libraries. Such a tool could allow contributors to define and refine security artifacts through intuitive interfaces, supported by guidance and validation mechanisms. By opening the development process to a wider community, this approach could enrich the diversity of available properties and constraints, while also encouraging peer review and increased scrutiny of individual entries.

   This collaborative model would not only improve the usability and scalability of the libraries but also support their integration into various modeling projects, thereby promoting consistency, transparency, and trust in the application of security requirements across systems.

2. *Security in the Age of AI* The rise of artificial intelligence (AI) in recent years raises pressing concerns about the level of confidence we can place in systems whose internal decision-making processes are often opaque or not fully understood. This challenge is particularly acute in the context of *critical systems*, such as those used in healthcare, autonomous vehicles, energy management, or aerospace, where security and correctness are paramount and failures can have catastrophic consequences.

Traditional formal methods rely on precise, mathematical modeling and complete control over the system's behavior to verify its correctness with respect to a set of specified properties. This approach stands in contrast with the data-driven and probabilistic nature of most AI techniques, especially those based on machine learning, which learn behavior from large datasets rather than being explicitly programmed.

In this context, applying formal methods to AI-based systems presents significant obstacles. One possible direction is the integration of formal constraints within the AI system design process. For example, one could formally specify safety envelopes or allowable action spaces that AI components must respect at runtime. Alternatively, AI modules could be encapsulated within formally verified wrappers or monitored by runtime verification systems to ensure they do not violate critical system properties.

These strategies require new methodologies to bridge the gap between formal verification and AI's inherent uncertainty and adaptiveness. Research in this area is still emerging, but it is critical for ensuring that the benefits of AI can be safely realized in domains where reliability, security, and trustworthiness are non-negotiable.

3. *Formal Assurance Cases* Certification of safety-critical systems requires the submission of assurance cases to demonstrate that a system meets the required safety standards. These assurance cases are typically constructed using safety cases, which may rely on either formal or informal argumentation to provide evidence supporting the system's safety. For high-assurance systems, the use of formal argumentation is strongly recommended. Event-B has been shown to be a suitable formalism for expressing safety cases [155]. In this context, it is pertinent to investigate whether the combination of MBE and formal methods can facilitate the construction of trustworthy assurance cases. Such a combination could not only increase the rigor and reliability of the safety arguments but also streamline the certification process, making it both more secure and more efficient.

9.4 Concluding Remarks

With security becoming a critical concern in the development of modern software systems—particularly in the context of increasingly distributed architectures—it is essential to address security considerations at the earliest stages of system conception. In this work, we propose a model-based approach to this challenge, supported by a comprehensive toolchain that integrates formal methods to enhance the level of trust in the resulting architectures. Rooted in the principles of separation of concerns and reusability, our approach aims to equip all stakeholders with the tools necessary to design secure systems with high assurance. We believe that this represents a significant step toward establishing an environment in which security is systematically integrated into the SDLC, achieving parity with traditional functional requirements.

Bibliography

[1] Alessandra Bagnato and Bagnato. *Handbook of Research on Embedded Systems Design*. IGI Global, 2014.

[2] International Organization for Standardization. ISO/IEC 19502:2005 information technology — meta object facility (mof), 2005.

[3] J. Miller and J. Mukerji. Mda guide version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[4] BSI. Protection Profile for the Gateway of a Smart Metering System (Smart Meter Gateway PP). Common Criteria Protection Profile BSI-CC-PP-0073, Bundesamt für Sicherheit in der Informationstechnik, 2014.

[5] Ron Ross, Mark Winstead, and Michael McEvilley. Engineering Trustworthy Secure Systems. Technical Report NIST Special Publication (SP) 800-160 Vol. 1 Rev. 1, National Institute of Standards and Technology, November 2022.

[6] J. Bezivin and O. Gerbe. Towards a precise definition of the OMG/MDA framework. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 273–280, November 2001. ISSN: 1938-4300.

[7] Douglas C. Schmidt. Model-driven engineering. *Computer-IEEE Computer Society-*, 39(2):25, 2006. Publisher: Citeseer.

[8] Ivica Crnkovic. Component-based software engineering — new challenges in software development. *Software Focus*, 2(4):127–133, December 2001.

[9] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4):19:1–19:36, October 2009.

[10] Jean Coulouris and Pearson Wesley. Distributed systems. https://user.it.uu.se/ justin/Archive/Teaching/DS/Slides/lecture1.pdf.

[11] Bran Selic. On the precise semantics of the software layering design pattern. *Journal of Object Technology*, 20(2):2, 2021.

[12] Nick Rozanski and Eoin Woods. *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley, 2012.

[13] Brahim Hamid, Adel Ziani, and Jacob Geisel. Towards tool support for pattern-based secure and dependable systems development. In *Proceedings of the workshop on ACadeMics Tooling with Eclipse*, pages 1–10, Montpellier France, July 2013. ACM.

[14] Eduardo B. Fernandez, Hironori Washizaki, Nobukazu Yoshioka, Atsuto Kubo, and Yoshiaki Fukazawa. Classifying Security Patterns. In Yanchun Zhang, Ge Yu, Elisa Bertino, and Guandong Xu, editors, *Progress in WWW Research and Development*, volume 4976, pages 342–347. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. Series Title: Lecture Notes in Computer Science.

[15] Quentin Rouland, Brahim Hamid, and Jason Jaskolka. Specification, detection, and treatment of STRIDE threats for software components: Modeling, formal methods, and tool support. *J. Syst. Archit*, 117:102073, 2021.

[16] Quentin Rouland, Brahim Hamid, and Jason Jaskolka. Formal specification and verification of reusable communication models for distributed systems architecture. *Future Generation Computer Systems*, 108:178–197, 2020. Publisher: Elsevier.

[17] Loïc Thierry, Jason Jaskolka, Brahim Hamid, and Jean-Paul Bodeveix. Specification and Verification of Communication Paradigms for CBSE in Event B. In Yamine Aït-Ameur, Ferhat Khendek, and Dominique Méry, editors, *27th International Conference on Engineering of Complex Computer Systems, ICECCS 2023, Toulouse, France, June 14-16, 2023*, pages 157–166. IEEE, 2023.

[18] Loïc Thierry, Brahim Hamid, and Jason Jaskolka. A Formal Approach for Verifying and Validating Security Objectives in Software Architecture. In Belgacem Ben Hedia, Mohamed Ghazel, and Bruno Monsuez, editors, *Verification and Evaluation of Computer and Communication Systems - 17th International Conference, VECoS 2024, Djerba, Tunisia, October 16-18, 2024, Proceedings*, volume 15466 of *Lecture Notes in Computer Science*, pages 143–158. Springer, 2024.

[19] Loïc Thierry, Brahim Hamid, and Jason Jaskolka. Reusable Formal Model Libraries for Specifying and Analyzing Security Objectives in Event-B. In Carlos Ordonez, Giancarlo Sperlì, Elio Masciari, and Ladjel Bellatreche, editors, *Model and Data Engineering - 13th International Conference, MEDI 2024, Naples, Italy, November 18-20, 2024, Proceedings*, volume 15590 of *Lecture Notes in Computer Science*, pages 55–63. Springer, 2024.

[20] International Organization for Standardization. ISO/IEC/IEEE 24765:2017 systems and software engineering — vocabulary, September 2017.

[21] Lvar Jacobson and James Rumbaugh Grady Booch. *The unified modeling language reference manual*. Addison-Wesley, 2021.

[22] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component software: beyond object-oriented programming*. Pearson Education, 2002.

[23] M. Torngren, DeJiu Chen, and I. Crnkovic. Component-based vs. model-based development: a comparison in the context of vehicular embedded systems. In *31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 432–440, August 2005.

[24] Desmond F D'Souza and Alan Cameron Wills. *Objects, components, and frameworks with UML : the catalysis approach*. Addison-Wesley Professional, 1998.

[25] Colin Atkinson, Joachim Bayer, and Dirk Muthig. Component-Based Product Line Development: The KobrA Approach. In *Software Product Lines*, The Springer International Series in Engineering and Computer Science, pages 289–309. Springer, Boston, MA, 2000.

[26] Ian Graham, Brian Henderson-Sellers, and Houman Younessi. *The OPEN Process Specification*. ACM Press/Addison-Wesley Publishing Co., 1997.

[27] Mathieu Jan, Christophe Jouvray, Fabrice Kordon, Antonio Kung, Jimmy Lalande, Frédéric Loiret, Juan F. Navas, Laurent Pautet, Jacques Pulou, Ansgar Radermacher, and Lionel Seinturier. Flex-eware: a flexible model driven solution for designing and implementing embedded distributed systems. *Softw., Pract. Exper.*, 42(12):1467–1494, 2012.

[28] Ivica Crnkovic. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.

[29] 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.

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

[31] D. Schmidt. Model-Driven Engineering. *in IEEE computer*, 39(2):41–47, 2006.

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

[33] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[34] OMG. MetaObject Facility 2.4.2, Specification. http://www.omg.org/spec/MOF/2.4.2/, 2014. [Accessed: July-2014].

[35] OMG. UML 2.5. http://www.omg.org/spec/UML/2.5/, 2015. [Accessed: December-2015].

[36] OMG. MOF QVT 1.1 Specification. http://www.omg.org/spec/QVT/1.1, 2011. [Accessed: June-2014].

[37] OMG. OCL 2.2 Specification. http://www.omg.org/spec/OCL/2.2, 2010.

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

[39] 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.

[40] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

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

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

[43] Julia H. Allen, Sean Barnum, Robert J. Ellison, Gary McGraw, and Nancy R. Mead. *Software security engineering*. Pearson India, 2008.

[44] Stephen Lamb. Security features in windows vista and ie7 – microsoft's view. *Network Security*, 2006(8):3 – 7, 2006.

[45] Daniel Mellado, Carlos Blanco, Luis E. Sánchez, and Eduardo Fernández-Medina. A systematic review of security requirements engineering. *Computer Standards & Interfaces*, 32(4):153–165, 2010. Publisher: Elsevier.

[46] John Viega. Building security requirements with CLASP. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, July 2005.

[47] Gary McGraw. *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

[48] Microsoft. Microsoft Security Development Lifecycle (SDL) – version 5.2, 2012.

[49] OWASP. OWASP CLASP V.A.2. Technical report, OWASP, November 2007.

[50] Gary McGraw. Software Security. *Datenschutz und Datensicherheit - DuD*, 36(9):662–665, September 2012.

[51] Charles Haley, Robin Laney, Jonathan Moffett, and Bashar Nuseibeh. Security requirements engineering: A framework for representation and analysis. volume 34, pages 133–153. IEEE, 2008.

[52] Jon G Hall, Michael Jackson, Robin C Laney, Bashar Nuseibeh, and Lucia Rapanotti. Relating software requirements and architectures using problem frames. In *IEEE Joint International Conference on Requirements Engineering*, pages 137–144. IEEE, 2002.

[53] Andreas L. Opdahl and Guttorm Sindre. Experimental comparison of attack trees and misuse cases for security threat identification. *Inf. Softw. Technol.*, 51(5):916–932, 2009.

[54] G. Sindre and Andreas L. Opdahl. Eliciting security requirements by misuse cases. In *37th International Conference on Technology of Object-Oriented Languages and Systems, 2000. TOOLS-Pacific 2000. Proceedings*, pages 120–131, 2000.

[55] N. Moebius, K. Stenzel, H. Grandy, and W. Reif. SecureMDD: A Model-Driven Development Method for Secure Smart Card Applications. In *International Conference on Availability, Reliability and Security*, ARES'09, pages 841–846, 2009.

[56] Torsten Lodderstedt, David A. Basin, and Jürgen Doser. Secureuml: A uml-based modeling language for model-driven security. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, UML '02, pages 426–441. Springer-Verlag, 2002.

[57] Jan Jürjens. Towards development of secure systems using umlsec. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, FASE '01, pages 187–200. Springer-Verlag, 2001.

[58] Jan Jürjens. UMLsec: Extending UML for Secure Systems Development. In Jean-Marc Jézéquel, Heinrich Hu\s smann, and Stephen Cook, editors, *UML 2002 - The Unified Modeling Language, 5th International Conference, Dresden, Germany, September 30 - October 4, 2002, Proceedings*, volume 2460 of *Lecture Notes in Computer Science*, pages 412–425. Springer, 2002.

[59] Holger Grandy, Dominik Haneberg, Wolfgang Reif, and Kurt Stenzel. Developing Provable Secure M-Commerce Applications. In *Emerging Trends in Information and Communication Security*, pages 115–129. Springer, Berlin, Heidelberg, 2006.

[60] Younghwa Lee, Zoonky Lee, and Choong Kwon Lee. A study of integrating the security engineering process into the software lifecycle process standard (IEEE/EIA 12207). *AMCIS 2000 Proceedings*, page 182, 2000.

[61] Younghwa Lee, Jintae Lee, and Zoonky Lee. Integrating Software Lifecycle Process Standards with Security Engineering. *Computers & Security*, 21(4):345–355, 2002.

[62] IEEE Standards Association.

[63] Jeannette M Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–22, 1990.

[64] Raymond M Smullyan. *First-order logic*. Courier Corporation, 1995.

[65] Brian F Chellas. *Modal logic: an introduction*. Cambridge university press, 1980.

[66] Hans Van Ditmarsch, Wiebe van Der Hoek, and Barteld Kooi. *Dynamic epistemic logic*, volume 337. Springer Science & Business Media, 2007.

[67] Georg Henrik Von Wright. Deontic logic. *Mind*, 60(237):1–15, 1951.

[68] E Allen Emerson. Temporal and modal logic. In *Formal Models and Semantics*, pages 995–1072. Elsevier, 1990.

[69] Rodin User's Handbook v.2.8: The First Machine: A Traffic Light Controller. `https://stups.hhu-hosting.de/handbook/rodin/current/html/tut_first_machine.html`.

[70] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[71] J. R. Abrial, C. Métayer, and L. Voisin. Event-B language. *Deliverable*, 3:2005, 2005.

[72] D. Steinberg, F. Budinsky, M. Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[73] Lars Vogel. Eclipse RCP. `http://www.vogella.de/articles/EclipseRCP/`, 2015. [Accessed: August-2016].

[74] J. McAffer, J-M. Lemieux, and C. Aniszczyk. *Eclipse Rich Client Platform*. Addison-Wesley Professional, 2nd edition, 2010.

[75] 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, November 2010.

[76] Jens Bendisposto, Fabian Fritz, Michael Jastram, Michael Leuschel, and Ingo Weigelt. Developing Camille, a text editor for Rodin. *Software: Practice and Experience*, 41(2):189–198, February 2011.

[77] OWASP. Application threat modeling. `https://www.owasp.org/index.php/Application_Threat_Modeling`, 2017. [Accessed: July-2025].

[78] M. Douglas McIlroy, J. Buxton, Peter Naur, and Brian Randell. Mass-produced software components. In *Proceedings of the 1st international conference on software engineering, Garmisch Pattenkirchen, Germany*, pages 88–98, 1968.

[79] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, October 1972.

[80] R.N.Taylor and N. Medvidovic. *Software architecture: Foundation, theory, and practice*. Wiley, 2010.

[81] I. Crnkovic. Component-based Software Engineering for Embedded Systems. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 712–713. ACM, 2005.

[82] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003. Publisher: IEEE Computer Society Press.

[83] M. Rodano and K. Giammarc. A Formal Method for Evaluation of a Modeled System Architecture. *Procedia Computer Science*, 20:210–215, 2013.

[84] Michael Shin, Hassan Gomaa, and Don Pathirage. A software product line approach for feature modeling and design of secure connectors. In *13th International Conference on Software Technologies*, ICSOFT 2018, pages 506–517, 2018.

[85] Tomas Bures and Frantisek Plasil. Communication style driven connector configurations. In C. V. Ramamoorthy, Roger Lee, and Kyung Whan Lee, editors, *Software Engineering Research and Applications*, pages 102–116. Springer Berlin Heidelberg, 2004.

[86] André C. Neto, Filippo Sartori, Fabio Piccolo, Riccardo Vitelli, Gianmaria De Tommasi, Luca Zabeo, Antonio Barbalace, Horacio Fernandes, Daniel F. Valcárcel, and Antonio JN Batista. MARTe: A multiplatform real-time framework. *IEEE Transactions on Nuclear Science*, 57(2):479–486, 2010. Publisher: IEEE.

[87] Arnaud Cuccuru, Sébastien Gérard, and Ansgar Radermacher. Meaningful Composite Structures. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems*, volume 5301, pages 828–842. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISSN: 0302-9743, 1611-3349 Series Title: Lecture Notes in Computer Science.

[88] Hans-Erik Eriksson, Magnus Penker, Brian Lyons, and David Fado. *UML 2 toolkit*. John Wiley & Sons, 2003.

[89] John Hutchinson, Mark Rouncefield, and Jon Whittle. Model-driven engineering practices in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 633–642, Waikiki, Honolulu HI USA, May 2011. ACM.

[90] Gregor Gössler and Joseph Sifakis. Composition for component-based modeling. *Science of Computer Programming*, 55(1-3):161–183, 2005. Publisher: Elsevier.

[91] David Akehurst and Stuart Kent. A Relational Approach to Defining Transformations in a Metamodel. In Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *"UML" 2002 — The Unified Modeling Language*, volume 2460, pages 243–258. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. Series Title: Lecture Notes in Computer Science.

[92] Samuel Kounev, Christoph Rathfelder, and Benjamin Klatt. Modeling of event-based communication in component-based architectures: state-of-the-art and future directions. *Electronic Notes in Theoretical Computer Science*, 295:3–9, 2013. Publisher: Elsevier.

[93] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[94] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes part I. *Information and Computation*, 100(1):1–40, September 1992.

[95] CAR. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, 1978.

[96] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[97] Andrea Ferrara. Web services: A process algebra approach. In M. Aiello, M. Aoyama, F. Curbera, and M.-P. Papazoglou, editors, *2nd international conference on Service oriented computing - ICSOC '04*, pages 242–251, New York, NY, USA, 2004. ACM Press.

[98] R. Alur R and DL. Dill. A Theory of Timed Automata. *heoretical Computer Science*, 12(6):183–235, 1994.

[99] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. The theory of timed i/o automata, 2011.

[100] Maurice H. Ter Beek, Clarence A. Ellis, Jetty Kleijn, and Grzegorz Rozenberg. Synchronizations in Team Automata for Groupware Systems. *Computer Supported Cooperative Work (CSCW)*, 12(1):21–69, February 2003.

[101] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997.

[102] Wei Ke, Xiaoshan Li, Zhiming Liu, and Volker Stolz. rCOS: a formal model-driven engineering method for component-based software. *Frontiers of Computer Science*, 6(1):17–39, February 2012.

[103] R. Khosrav, M. Sirjani, N. Asoudeh, S. Sahebi, and H. Iravanchi. Modeling and Analysis of Reo Connectors Using Alloy. In *Coordination Models and Languages*, pages 169–183. Springer Berlin Heidelberg, 2008.

[104] Amel Mammar, Lazhar Hamel, and Mohamed Graiet. An event-b-based approach to model and verify behaviors for component-based applications, 2021.

[105] Jean-Paul Bodeveix, Arnaud Dieumegard, and Mamoun Filali. *Event-B Formalization of a Variability-Aware Component Model Patterns Framework*, pages 54–74. Springer International Publishing, 2018.

[106] Guillaume Babin, Yamine Ameur, and Marc Pantel. *Formal Verification of Runtime Compensation of Web Service Compositions: A Refinement and Proof Based Proposal with Event-B*, pages 98–105. IEEE, 6 2015.

[107] Mohamed Graiet, Lazhar Hamel, Amel Mammar, and Samir Tata. A verification and deployment approach for elastic component-based applications. *Formal Aspects of Computing*, 29(6):987–1011, 11 2017.

[108] Mohamed Graiet, Imed Abbassi, Lazhar Hamel, Mohamed Bhiri, Mourad Kmimech, and Walid Gaaloul. *Event-B Based Approach for Verifying Dynamic Composite Service Transactional Behavior*, pages 251–259. IEEE, Santa Clara, CA, USA, 6 2013.

[109] J Christian Attiogbé. Mapping component models on distributed architectures: Correctness checking. In *MoDeVVa@ MoDELS*, pages 1–9, 2016.

[110] Daniel Le Berre and Pascal Rapicault. Dependency management for the eclipse ecosystem: eclipse p2, metadata and resolution. In *Proceedings of the 1st international workshop on Open component ecosystems*, pages 21–30, Amsterdam The Netherlands, August 2009. ACM.

[111] Gary McGraw. Software Security: Building Security In. *Datenschutz und Datensicherheit - DuD*, 36(9):662–665, September 2012.

[112] Michael Howard and Steve Lipner. *The security development lifecycle*, volume 8. Microsoft Press Redmond, 2006.

[113] Archiveddocs. The STRIDE Threat Model, November 2009.

[114] OWASP. *The CLASP Application Security Process*. Secure software, inc. edition, 2005.

[115] Torsten Lodderstedt, David Basin, and Jürgen Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook,

editors, *UML 2002 — The Unified Modeling Language*, volume 2460, pages 426–441. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. Series Title: Lecture Notes in Computer Science.

[116] Jan Jürjens. UMLsec: Extending UML for Secure Systems Development. In Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *UML 2002 — The Unified Modeling Language*, volume 2460, pages 412–425. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. Series Title: Lecture Notes in Computer Science.

[117] Atif Mashkoor, Alexander Egyed, Robert Wille, and Sebastian Stock. Model-driven engineering of safety and security software systems: A systematic mapping study and future research directions. *Journal of Software: Evolution and Process*, 35(7):e2457, 2023. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2457.

[118] Uwe Glässer, Piper Jackson, Ali Khalili Araghi, and Hamed Yaghoubi Shahir. Intelligent decision support for Marine safety and Security Operations. In *2010 IEEE International Conference on Intelligence and Security Informatics*, pages 101–107, May 2010.

[119] Uwe Glässer, Piper Jackson, Ali Khalili Araghi, Hans Wehn, and Hamed Yaghoubi Shahir. A Collaborative Decision Support Model for Marine Safety and Security Operations. In Mike Hinchey, Bernd Kleinjohann, Lisa Kleinjohann, Peter A. Lindsay, Franz J. Rammig, Jon Timmis, and Marilyn Wolf, editors, *Distributed, Parallel and Biologically Inspired Systems*, volume 329, pages 266–277. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. Series Title: IFIP Advances in Information and Communication Technology.

[120] Bastian Best, Jan Jurjens, and Bashar Nuseibeh. Model-based security engineering of distributed information systems using UMLsec. In *29th International Conference on Software Engineering (ICSE'07)*, pages 581–590. IEEE, 2007.

[121] Valentina Casola, Alessandra De Benedictis, Carlo Mazzocca, and Vittorio Orbinato. Secure software development and testing: A model-based methodology. *Computers & Security*, 137:103639, 2024. Publisher: Elsevier.

[122] Rahma Bouaziz, Brahim Hamid, and Nicolas Desnos. Towards a Better Integration of Patterns in Secure Component-Based Systems Design. In Beniamino Murgante, Osvaldo Gervasi, Andrés Iglesias, David Taniar, and Bernady O. Apduhan, editors, *Computational Science and Its Applications - ICCSA 2011*, volume 6786, pages 607–621. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. Series Title: Lecture Notes in Computer Science.

[123] Tomas Kulik, Brijesh Dongol, Peter Gorm Larsen, Hugo Daniel Macedo, Steve Schneider, Peter W. V. Tran-Jørgensen, and James Woodcock. A Survey of Practical

Formal Methods for Security. *Formal Aspects of Computing*, 34(1):1–39, March 2022.

[124] Carl E. Landwehr. Formal Models for Computer Security. *ACM Computing Surveys*, 13(3):247–278, September 1981.

[125] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[126] David Von Oheimb and Sebastian Mödersheim. ASLan++ — A Formal Security Specification Language for Distributed Systems. In Bernhard K. Aichernig, Frank S. De Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 6957, pages 1–22. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. Series Title: Lecture Notes in Computer Science.

[127] Ekkart Kindler, Vladimir Rubin, and Robert Wagner. Component Tools: Integrating Petri Nets with Other Formal Methods. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Susanna Donatelli, and P. S. Thiagarajan, editors, *Petri Nets and Other Models of Concurrency - ICATPN 2006*, volume 4024, pages 37–56. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. Series Title: Lecture Notes in Computer Science.

[128] Quentin Rouland, Brahim Hamid, Jean-Paul Bodeveix, and Jason Jaskolka. Formalizing the Relationship between Security Policies and Objectives in Software Architectures. In *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*, pages 151–158, March 2023. ISSN: 2768-4288.

[129] Linas Laibinis, Elena Troubitsyna, Inna Pereverzeva, Ian Oliver, and Silke Holtmanns. A formal approach to identifying security vulnerabilities in telecommunication networks. In *Formal Methods and Software Engineering*, pages 141–158, 2016.

[130] Nazim Benaissa and Dominique Méry. Cryptographic protocols analysis in Event B. In *Perspectives of Systems Informatics*, pages 282–293. Springer Berlin Heidelberg, 2010.

[131] Víctor Rivera, Sukriti Bhattacharya, and Néstor Cataño. Undertaking the tokeneer challenge in event-b. In *Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering*, pages 8–14, 2016.

[132] Thai S Hoang, David Basin, and Jean-Raymond Abrial. Specifying access control in event-b. *Technical report*, 624, 2009.

[133] Amjad Gawanmeh, Sofiène Tahar, and Leila Jemni Ben Ayed. Formal verification of secrecy in group key protocols using Event-B. *Intl J. of Communications, Network and System Sciences*, 5(03):165–177, 2012.

[134] P. Devanbu, S. Stubblebine, S. S. Premkumar, and T. Devanbu. Software engineering for security - a roadmap. In *Proceedings of the Conference on The Future of Software Engineering, ICSE '00*, pages 227–239. ACM, 2000.

[135] L. C. Paulson. Proving Properties of Security Protocols by Induction. Technical Report 409, Computer Laboratory, University of Cambridg, 1996.

[136] Marc Frappier, Frédéric Gervais, Régine Laleau, Benoît Fraikin, and Richard St-Denis. Extending statecharts with process algebra operators. *Innovations in Systems and Software Engineering*, 4(3):285–292, October 2008.

[137] Dominique Méry and Neeraj Kumar Singh. Automatic code generation from event-B models. In *Proceedings of the 2nd Symposium on Information and Communication Technology*, SoICT '11, pages 179–188, New York, NY, USA, October 2011. Association for Computing Machinery.

[138] Quentin Rouland, Brahim Hamid, and Jason Jaskolka. Specification, detection, and treatment of stride threats for software components: Modeling, formal methods, and tool support. *J. Syst. Archit*, 117:102073, 2021.

[139] OMG. Unified Modeling Language (UML), Version 2.5.1. `https://www.omg.org/spec/UML/2.5.1/About-UML`, 2017. [Accessed: January-2023].

[140] OMG. CORBA Specification, Version 4.0. `http://www.omg.org/spec/CCM`, 2009. [Accessed: January-2023].

[141] OMG. Object Constraint Language (OCL), Version 2.4. `https://www.omg.org/spec/OCL`, 2014. [Accessed: January-2023].

[142] SAE. Architecture Analysis & Design Language (AADL). `https://www.sae.org/standards/content/as5506d/`, 2022. [Accessed: January-2023].

[143] Drasko M. Sotirovski and Philippe B. Kruchten. Implementing dialogue independence. *IEEE Software*, 12(6):61–70, 1995. Publisher: IEEE.

[144] Tim Dierks and Christopher Allen. The TLS protocol version 1.0. Technical report, 1999.

[145] ISO-IEC. Information technology - security techniques - information security management systems - overview and vocabulary. `https://www.iso.org/obp/ui/#iso:std:iso-iec:27000:ed-5:v1:en:term:3.36`, 2018.

[146] Alexander Chagrov and Michael Zakharyaschev. *Modal logic*. Oxford University Press, 1997.

[147] Amel Mammar, Marc Frappier, Steve Jeffrey Tueno Fotso, and Régine Laleau. A formal refinement-based analysis of the hybrid ERTMS/ETCS level 3 standard. *Int. J. Softw. Tools Technol. Transf.*, 22(3):333–347, 2020.

[148] Yamine Aït-Ameur, Guillaume Dupont, Ismail Mendil, Dominique Méry, Marc Pantel, Peter Rivière, and Neeraj K. Singh. Empowering the Event-B Method Using External Theories. In Maurice H. Ter Beek and Rosemary Monahan, editors, *Integrated Formal Methods*, volume 13274, pages 18–35. Springer International Publishing, Cham, 2022. Series Title: Lecture Notes in Computer Science.

[149] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Natasha Sharygina, and Helmut Veith, editors, *Computer Aided Verification*, volume 8044, pages 696–701. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. Series Title: Lecture Notes in Computer Science.

[150] Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre. ProVerif 2.00: automatic cryptographic protocol verifier, user manual and tutorial. *Version from*, 16:05–16, 2018.

[151] Philippe B. Kruchten. The 4+ 1 view model of architecture. *IEEE software*, 12(6):42–50, 1995. Publisher: IEEE.

[152] Guillaume Dupont, Yamine Ait-Ameur, Neeraj Kumar Singh, and Marc Pantel. Formally verified architectural patterns of hybrid systems using proof and refinement with Event-B. *Science of Computer Programming*, 216:102765, April 2022.

[153] Dominique Cansell, Dominique Méry, and Joris Rehm. Time Constraint Patterns for Event B Development. In Jacques Julliand and Olga Kouchnarenko, editors, *B 2007: Formal Specification and Development in B*, pages 140–154, Berlin, Heidelberg, 2006. Springer.

[154] European Union Agency for Network and Information Security (ENISA). Threat Taxonomy. `https://www.enisa.europa.eu/topics/threat-risk-management/threats-and-trends/enisa-threat-landscape/threat-taxonomy/view`, 2016. [Accessed: November-2018].

[155] Yuliya Prokhorova, Linas Laibinis, and Elena Troubitsyna. Facilitating construction of safety cases from formal models in Event-B. *Information and Software Technology*, 60:51–76, 2015. Publisher: Elsevier.

Titre : Modélisation formelle d'architectures sécurisées dans une perspective positive à l'aide d'Event-B : propriétés, modèles, analyse et outillage

Mots clés : Securité, Objectifs, Architecture, Modèles, Techniques formelles

Résumé : Le développement des systèmes informatiques distribués et leur utilisation dans des domaines tels que l'Internet des Objets, le Big Data, etc., soulèvent de nombreuses questions quant aux outils disponibles pour modéliser la complexité de ces systèmes. Les méthodes de modélisation non formelles échouent à fournir un cadre rigoureux permettant de décrire et d'analyser ces systèmes. Dans cette thèse, nous proposons de définir une approche permettant de modéliser et d'analyser à la fois les aspects fonctionnels et non fonctionnels de ces systèmes à l'aide de techniques formelles.
Les objectifs de ce travail incluent :
(1) explorer l'utilisation des méthodes formelles basées sur la preuve pour la spécification et la vérification des propriétés structurelles et comportementales des systèmes à base de composants,
(2) décrire et valider des propriétés de sécurité dans des bibliothèques réutilisables,
(3) décrire et classifier les relations entre les propriétés de sécurité,
et (4) développer un ensemble d'outils.
Pour atteindre ces objectifs, nous cherchons à :
(a) décrire les concepts de haut niveau pour l'architecture système selon une approche composant-port-connecteur sous forme de métamodèle ;
(b) développer une interprétation Event-B de ce métamodèle en y ajoutant des caractéristiques de communication telles que le tamponage, le premier entré-premier sorti (FIFO) et la synchronicité ;
(c) étendre ce modèle formel pour représenter les propriétés de sécurité ;
et (d) créer une suite d'outils automatisés basée sur ce modèle formel, pouvant être intégrée dans un cycle de développement.
L'approche combine à la fois l'ingénierie dirigée par les modèles (IDM) et les techniques formelles afin de créer des méthodes et des outils pour la conception, la vérification et la validation des architectures systèmes complexes. Les méthodes et outils proposés s'intéressent non seulement aux exigences fonctionnelles, mais aussi aux exigences non fonctionnelles, en s'appuyant sur des bibliothèques réutilisables de propriétés de sécurité, utilisables dès la phase de conception. Pour valider notre travail, nous étudions un ensemble d'objectifs de sécurité représentatifs issus de la classification confidentialité, intégrité, disponibilité (CIA), ainsi qu'une étude de cas sur le protocole TLS (Transport Layer Security).

Title: Formal modeling of secure architecture within the positive perspective using Event-B: properties, model, analysis and tool support

Key words: Security, Objectives, Architecture, Models, Formal techniques

Abstract: The development of distributed computing systems and of their usage in domains such as the Internet of Things, Big Data, etc., raises numerous questions on the tools available to model the complexity of such systems. Non-formal modeling methods fail to create a rigorous way to describe and analyze these systems. In this thesis, we propose to find an approach to model and analyze both functional and non-functional aspects of these systems using formal techniques.
The objectives of this work include: (1) exploring the use of proof-based formal methods for the specification and verification of both structural and behavioral properties of component-based systems, (2) describing and validating security properties in reusable libraries, (3) describing and classifying the relationships between security properties, and (4) developing a set of supporting tools.
To achieve these objectives, we seek to:
(a) describe high-level concepts for system architecture in a component-port-connector fashion as a metamodel; (b) develop an Event-B interpretation of the metamodel adding communication characteristics such as buffering, first-in-first-out (FIFO), and synchronicity; (c) extend this formal model to represent security properties; and (d) create an automated tool suite using the formal model that can be integrated into a systems development lifecycle.
The approach leverages both model-driven engineering (MDE) and formal techniques to create methods and tools for designing, verifying, and validating complex system architectures. The methods and tools will not only focus on functional requirements, but also on non-functional requirements by creating reusable libraries of security properties that can be used during the design phase. To validate our work, we explore a set of representative security objectives extracted from the confidentiality, integrity, availability (CIA) classification and a case study on the Transport Layer Security (TLS) protocol.