THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *29/10/2021* **par :**

Quentin ROULAND

Rigorous development of secure architecture within the negative and positive statements: properties, models, analysis and tool support

JURY

Jean-Paul Bodeveix	Professeur des Universités	Président du Jury
Samia Bouzefrane	Professeure des Universités	Examinateur
Brahim Hamid	Professeur des Universités	Directeur de Thèse
Jason Jaskolka	Associate Professor	Examinateur
Régine Laleau	Professeure des Universités	Rapporteur
Carsten Rudolph	Associate Professor	Rapporteur

École doctorale et spécialité :
MITT : Informatique et Télécommunications

Unité de Recherche :
Institut de Recherche en Informatique de Toulouse (UMR 5055)

Directeur de Thèse :
Brahim HAMID

Rapporteurs :
Régine LALEAU et *Carsten RUDOLPH*

Résumé

Notre société est devenue plus dépendante des systèmes logiciels complexes, tels que les systèmes de technologies de l'information et de la communication (TIC), pour effectuer des tâches quotidiennes (parfois critiques). Cependant, dans la plupart des cas, les organisations et particulièrement les plus petites placent une valeur limitée sur les données et leur sécurité. Dans le même temps, la sécurité de ces systèmes est une tâche difficile en raison de la complexité et connectivité croissante dans le développement des TIC. De plus, la sécurité a une incidence sur de nombreux attributs tels que la transparence, la sureté et l'utilisabilité. Ainsi, la sécurité devient un aspect très important qui devrait être pris en compte dans les premières phases du cycle de développement. Dans ce travail, nous proposons une approche afin de sécuriser les architectures logicielles des TIC pendant leur développement en considérant la vision positive, qui se manifeste par l'étude des objectifs de sécurité (ex., confidentialité), et la vision négative, qui se manifeste par l'étude des menaces (ex., usurpation). Les contributions de ce travail sont triples: (1) un framework de conception intégré pour la spécification et l'analyse de bibliothèques de modèles (formels) réutilisables pour les architectures logicielles sécurisées; (2) une nouvelle méthodologie basée sur les modèles pour développer une architecture logicielle sécurisée par réutilisation; et (3) une suite d'outils support.

L'approche associe l'ingénierie dirigée par les modèles (IDM) et les techniques formelles pour concevoir un ensemble de langages de modélisation afin de spécifier et analyser des modèles d'architecture et de propriétés permettant la réutilisation et ainsi capitaliser un savoir-faire en matière de sécurité. Les résultats sont fournis sous forme de deux artéfacts complémentaires: (a) un processus de développement de bibliothèques de modèles réutilisables pour la spécification et la vérification d'objectifs et de menaces de sécurité par un expert en sécurité; et (b) un processus de conception d'architecture sécurisé par un architecte s'appuyant sur les bibliothèques spécifiées dans le processus (a). Le processus (a) comprend les activités suivantes: (1) la spécification formelle d'objectifs et de menaces de sécurité comme propriétés d'un modèle en utilisant un langage de spécification indépendant technologiquement; (2) l'interprétation des bibliothèques de

modèles résultants dans un langage formel outillé; et (3) la définition de politiques de sécurité en tant que solutions abstraites de sécurité pour assurer les propriétés de sécurité. Le processus (b) comprend les activités suivantes: (1) l'analyse d'un modèle d'architecture concret afin de vérifier les exigences et d'identifier les problèmes de sécurité en réutilisant les modèles de propriétés; (2) la sélection et l'intégration des modèles de politiques pour atténuer les problèmes de sécurité identifiés; et (3) la génération de feedbacks sous forme de rapport sur le modèle d'architecture logicielle et sa sécurité (menaces résiduelles, validation d'objectifs, etc.).

Pour valider notre travail, nous avons exploré un ensemble de menaces représentatives extraites de la classification STRIDE et un ensemble d'objectifs de sécurités représentatives extraites de la classification CIAA dans le contexte de développement d'architecture à base de composants et de communication à base de messages. Dans le cadre de l'assistance au développement d'architectures sécurisées, nous avons mis en place une chaine d'outils autour de la plateforme Eclipse afin de soutenir les différentes activités de notre approche s'appuyant sur un ensemble de langages de modélisation et de langages formels outillés existants. L'évaluation de ce travail est présentée à travers un exemple simple de site web de bibliothèque universitaire issue de OWASP et un système de passerelle de compteur intelligent issue d'un projet de recherche collaborative de notre équipe.

Abstract

Our society has become more dependent on software-intensive systems, such as Information and Communication Technologies (ICTs) systems, to perform their daily tasks (sometimes critical). However, in most cases, organizations and particularly small ones place limited value on information and its security. In the same time, achieving security in such systems is a difficult task because of the increasing complexity and connectivity in ICT development. In addition, security has impacts on many attributes such as openness, safety and usability. Thus, security becomes a very important aspect that should be considered in early phases of development. In this work, we propose an approach in order to secure ICT software architectures during their development by considering two visions to formulate security statements using the negative view, as the study of threats (e.g., usurpation) and positive view as the study of the security objectives (e.g., confidentiality). The contributions of this work are threefold: (1) an integrated design framework for the specification and analysis of reusable (formal) model libraries for secure software architectures; (2) a novel model-based methodology for developing secure software architecture by reuse; and (3) a set of supporting tools.

The approach associates Model-Driven Engineering (MDE) and formal techniques to design a set of modeling languages for specifying and analyzing architecture and property models which allows reuse of capitalized security-related know-how. The results are provided as two complementary artifacts : (a) a process of development of reusable formal model libraries for the specification and verification of security threats and objectives by a security expert; and (b) a process of secure architectural design and analysis by an architect reusing the libraries specified in the process (a). Process (a) includes the following activities: (1) the formal specification of the security threats and objectives as the properties of a model using technology-independent specification language; (2) the interpretation of the resulted model libraries in a tooled formal language; and (3) the definition of security policies as abstract security countermeasures to ensure security properties. Process (b) includes the following activities: (1) security analysis of a concrete architecture model to verify the security requirements and identify security issues reusing the property models; (2) selection and integration of policy models to mitigate the identified security issues; and (3) the generation of feedback as reports on the software architecture model and its security (residual threats, objectives validation, etc.).

To validate our work, we have explored a set of representative threats extracted from the STRIDE classification and a set of representative security objectives extracted from the CIAA classification in the context of component-based architecture and message passing communication. As part of the assistance for the development of secure architectures, we have implemented a tool chain based on Eclipse platform to support the different activities of our approach based on a set of modeling languages and existing formal tooled languages. The assessment of our work is presented via a simple example of a university library website from OWASP and a gateway application for smart-meter system from our previous collaborative research project.

Remerciements

Une importante étape pour moi se finit aujourd'hui, je souhaite donc tout d'abord remercier tous ceux qui ont rendu possible ou contribué, d'une manière ou d'une autre, à l'achèvement de ce manuscrit.

En premier lieu, je tiens à remercier très chaleureusement mon directeur de thèse Brahim Hamid, sans lequel je n'aurais probablement jamais envisagé de réaliser un doctorat. De plus, un grand merci pour m'avoir partagé le gout de la recherche, et fourni un soutien permanent et de nombreux conseils pendant toutes ces années.

Je tiens aussi à remercier Jean-Paul Bodeveix et Mamoun Filali Amine pour les nombreux retours et discussions dans le cadre du projet ISARP qui ont été précieux et clés aux succès et achèvement de ce manuscrit. Je remercie aussi Jason Jaskolka pour l'aide apporter et la collaboration. Nos multiples échanges ont été vraiment stimulants et enrichissants. Je remercie toutes les personnes que j'ai côtoyées à l'IRIT (chercheurs, secrétaires, enseignants, et techniciens) qui ont contribué à un environnement agréable.

Je souhaiterais ensuite remercier Régine Laleau et Carsten Rudoph, pour l'intérêt qu'ils ont apporté à mon travail en acceptant d'être rapporteurs de cette thèse. Je les remercie pour leurs remarques, questions, et perspectives très intéressantes.Je voudrais également remercier Jean-Paul Bodeveix pour avoir accepté de présider le jury de me soutenance. Je remercie également Jean-Paul Bodeveix, Samia Bouzefarne, Brahim Hamid, Jason Jaskolka, Régine Laleau, et Carsten Rudoph, pour leurs questions et commentaires stimulant lors de la soutenance.

Finalement, merci, à toutes les personnes extérieures qui m'ont apporté leurs soutiens durant ces années. Je pense tout d'abord, à ma famille pour leur soutien indéfectible. Mais aussi, à mes amis qui ont contribué tous les jours à ma bonne humeur. Finalement, ma chatte, toujours présente à la fin d'une dure journée de travail.

Acknowledgements

An important milestone for me comes to an end today, so, first of all, I would like to thank everyone who has made possible or contributed, in one way or another, to the completion of this manuscript.

First of all, I would like to warmly thank my thesis director Brahim Hamid, without whom I would probably never have considered doing a doctorate. In addition, a big thank you for sharing my taste for research, and providing ongoing support and advice during all these years.

I would also like to thank Jean-Paul Bodeveix and Mamoun Filali Amine for the much feedback and discussions within the framework of the ISARP project which were invaluable and key to the success and completion of this manuscript. I also thank Jason Jaskolka for the help and collaboration. Our multiple exchanges have been truly stimulating and enriching. I would like to thank all the people I met at IRIT (researchers, secretaries, teachers, and technicians) who have contributed to an environment pleasant.

I would then like to thank Régine Laleau and Carsten Rudoph, for the interest they have shown in my work by accepting to be rapporteurs for this thesis. I thank them for their remarks, questions, and very interesting perspectives. I would also like to thank Jean-Paul Bodeveix for agreeing to chair the jury for my defense. I also thank Jean-Paul Bodeveix, Samia Bouzefarne, Brahim Hamid, Jason Jaskolka, Régine Laleau, and Carsten Rudoph, for their stimulating questions and comments during the defense.

Finally, thank you to all the outside people who have supported me during these years. First of all, I think of my family for their unwavering support. But also, to my friends who contributed every day to my good mood. Finally, my cat, always present at the end of a hard day's work.

Table of contents

List of figures

List of tables

List of listings

Chapter 1

Introduction

Contents

1.1 Context

The shift from traditional computer systems towards the Internet of Things (IoT), i.e., devices connected via the Internet, Machine-to-Machine communication (M2M), wireless communication or other interfaces requires a reconsideration of complex software-dependent and distributed systems engineering processes. In fact, this reconsideration introduces new types and levels of risks, including those inherited from the underlying technologies like communication, virtualization, and containerization. This is especially true for industrial systems which exist in many use cases, and systems using web applications due to the recent growth of more applications in cloud-based computing systems. Many of these systems belong to critical infrastructure, on which other economic and social aspects are based. A comprehensive understanding of modern communication systems and technologies and their implications on the underlying critical infrastructure [23] is the foundation for comprehensive rigorous systems engineering facing strong non-functional requirements such as security [109, 141]. In this dissertation, we take this need towards software engineering for distributed software systems, focusing on the problem of integrating communication styles at the level of architecture design to foster reuse.

When we study distributed systems, we often use models to denote some abstract representation of a distributed system. To encode distributed computing (programs) in such systems, we use a common means of communication [23], where system components have only local vision of the system and interact only with their neighbors with explicit communication models like message passing, remote procedure calls, and distributed shared memory. These communication models are common to most distributed systems. The program executed at each node consists of a set of variables (state) and a finite set of actions. A component can write to its own variables and interact with its neighbors following a specific communication style. In our context, we model software architectures with message passing, remote procedure calls, and distributed shared memory. These communication styles capture those which we expect the architectural description to adhere. The aim of this modeling and verification is to check if the architecture models satisfy all the desired properties such as security properties, and do not hold any undesired properties such as deadlock and starvation.

Security risk assessment is usually done by a security risk analyst in order to verify the actual status of an information system that is already deployed. Risk assessment is performed by a set of meetings between security experts and persons responsible for the information system. The main goal is to produce a report with actual security risks targeting the system, the security strategy to adopt and the security measures to deploy in order to achieve this strategy.

The first issue here is that developers are constrained by the functional requirements of the information system that can hardly change because it would mean choosing a new software system which is very expensive. In order to solve the first problem, security must be thought about at early stages of the system development. In our work, we focus on the architecture development stage where design decisions are still flexible. We employ Model-Driven Engineering (MDE) [123] and attempt to add more formality to improve parts of the system design.

The second issue is related to the fact that architects and developers usually have basic knowledge in security engineering but lack expertise and best practices to apply the correct recommendations issued by security risk assessments (if any). One solution is to use reusable models e.g., policies, patterns.

The work in this thesis is part of our research team effort to promote the use of model- and pattern-based engineering approach to improve the design, implementation, configuration and deployment of multi-concerns systems [42]. In the context of our work, we used a similar definition of a *concern* about an architecture as [119] : 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 and Software Engineering (PBSE) framework [46] considering security and safety requirements within software architectures built on top of the reusable models described in this thesis. Capturing and providing expertise by the way of security patterns has become an area of research in the last years. Security can be captured within patterns that provide reusable generic solutions for recurring security problems, here dealing with architectural problems. Recently a complete catalog of security patterns has been introduced by Fernandez [32]. We plan to transform our Pattern-Based System and Software Engineering (PBSE) pattern modeling concepts to formal specifications to ensure semantic validation.

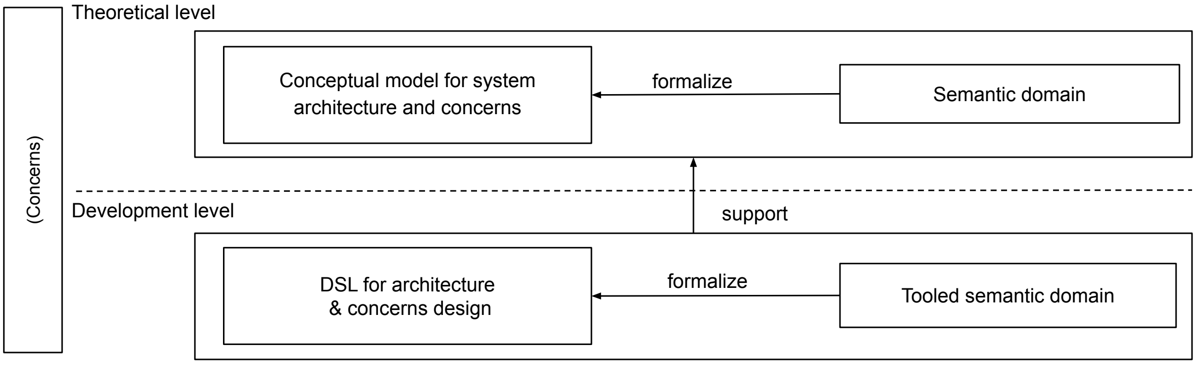


Figure 1.1: Conceptual vision

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

- Theoretical level: Creation of a conceptual model of a system and concerns and the semantics. A conceptual model of a system and 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 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 DSML for the creation of the design environment and existing tooled formal languages for the analysis environment.

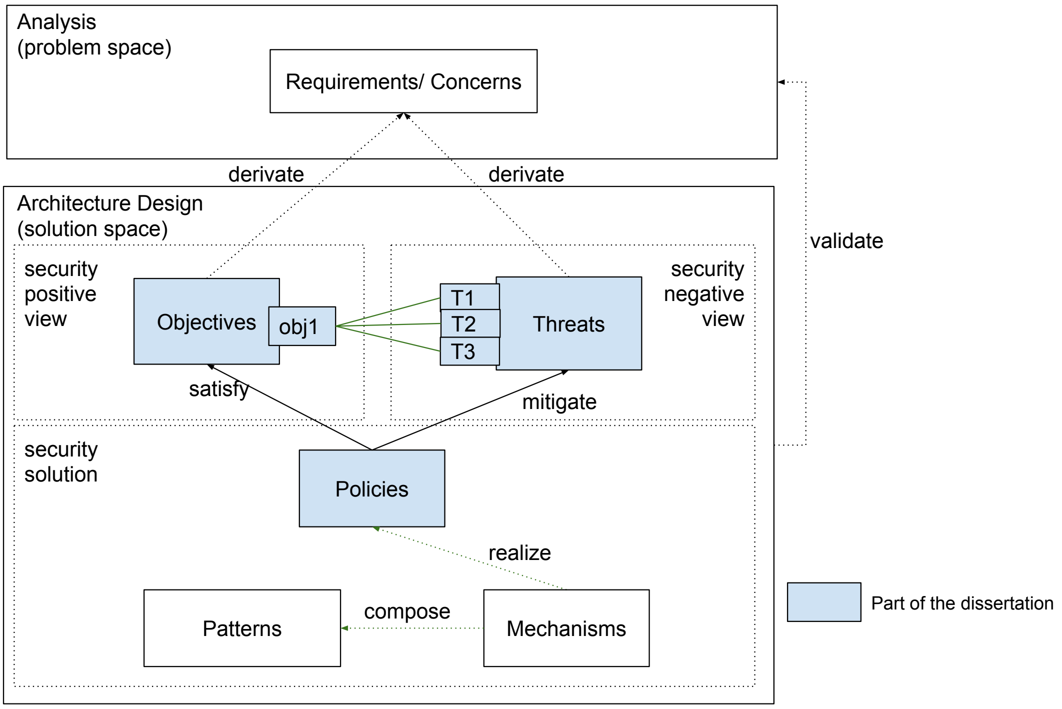


Figure 1.2: Notions and relationships

Figure 1.2, positions the identified notions for this vision and highlights the parts which are studied in this dissertation. Globally, the goal of the approach is to validate set of security requirements/concerns. The requirements are given as inputs i.e., we don't elicit new requirements. Note, however, that the set of requirements/concerns are defined is another work, e.g., validating their completeness is out of scope of our work. We formalize objectives and threats as an implementation of the system requirements or part of them at architecture level through two views: (1) the security positive view where the concerns are specified as security objectives, i.e., a property that must be satisfied in the architecture design; and, (2) the security negative view where the concerns are specified as security threats, i.e., a property that must never be satisfied in the architecture design. Note that two views could be used separately or complementary. This complementary would be exploited by defining association between threats and objectives. Intuitively, a threat could violate an objective. Fulfilling an objective could protect against one or multiple threats. Then, policies, as abstract mechanisms, are specified as treatment for the detected issues. Policies *mitigate* threat(s) for the negative view or/and *satisfy* objective(s) for the positive view. Finally, the concrete mechanism would be proposed and verified to realize the policy, and, patterns would be defined as a composition of mechanisms. This way we obtain a set of verified patterns offering solutions for recurring security problems.

1.2 Problem statement

Based on the previous discussion, we specify our general research problem coming from the lack of methodological tool support of both the system architecture and security.

Define and assess a framework for the development of system architecture and security using reusable models. More specifically, this work aims to provide formalisms, techniques, methods, and tools that enable the design of secure system architectures in a less complex and safer manner, in the context of future automated security-by design for the development of distributed systems.

1.3 Research objectives

Taking into account the previous discussion, we specify our research problem as an overall research objective of this thesis:

Propose an integrated approach for the specification, detection, and treatment of security concerns to support secure systems engineering at early stages of development using reusable models with tool support.

We present a model-based approach for developing secure software systems that employs reusable models to represent communication solutions, security issues, and security solutions. Thereby, this approach promotes: expert knowledge capitalization by allowing architects to reuse previously developed and verified models; and simpler and safer design for secure software system architecture through the combination of MDE and formal methods (model reuse and automatic validation).

Decomposing the overall research objective, we formulate the following top-level research goals:

Research objective 1.

RO1. *Define an approach for the development and formal verification of reusable models for software system architecture and security engineering.*

Research objective 2.

RO2. *Develop an approach towards software system architecture and security engineering using reusable models.*

Research objective 3.

RO3. *Build a tool suite to support the proposed approaches to enable their practical adoptions.*

1.4 Contributions

The contributions of this work are fourfold: (1) a development framework for the specification and the analysis of reusable formal models, (2) a design framework for the specification and analysis of secure software architectures, (3) a novel reusable formal model based methodology, and (4) a support tool.

Here, we map the contributions of this thesis to the research objectives formulated earlier.

RO1 is addressed with the following contributions:

- ***C1***. *The creation of a design and analysis framework for the development of reusable formal models for architecture and security.*
- ***C2***. *Software architecture and security meta-model*
  - ***C2.1***. *Component based architecture meta-model* : we propose a meta-model to capture representative software concepts (component, port, connector, ...) and communication styles (MPS, RPC, ...) commonly used in distributed system architecture modeling.
  - ***C2.2***. *Property meta-model for security*: we extend the meta-model from ***C2.1*** with security concepts for describing threats, objectives, policies and their relationships in the form of categories to build property model libraries for reuse.

- ***C3***. *Specification of the architecture model and properties in a technology-independent formalism (FoL and Modal logic).*
  - ***C3.1***. *Component based architecture model* : we define a logical specification of the architecture component model from ***C2.1***.
  - ***C3.2***. *Property model for security*: we extend the specifications from ***C3.1*** with security concepts for describing threats and objectives as properties of the system architecture model.
- ***C4***. *Interpretation of the proposed architecture and property meta-model in tooled formal languages (Alloy and Coq).*
  - ***C4.1***. *Component based architecture formal model* : we define a formal interpretation of the meta-model from ***C2.1*** and ***C3.1***.
  - ***C4.2***. *Property formal model for security*: we define a formal interpretation of the meta-model from ***C2.2*** and ***C3.2***, extending the formal model from ***C4.1*** with security concepts (threats, objectives, policies and their relationships) as properties of the system architecture model and provide them in the form of reusable formal model libraries.
- ***C5***. *Creation of verified formal model libraries for reuse in the context of the component & MPS based architecture model*
  - ***C5.1***. *Communication model libraries:* we developed a set of reusable connector models as a reusable communication model libraries, including Message Passing System (MPS) communication.
  - ***C5.2***. *STRIDE security threats*: we developed a set of threat models, capturing representative security threats extracted from the Microsoft STRIDE classification, and provide them as a reusable formal model library.
  - ***C5.3***. *CIAA security objectives*: we developed a set of security objective models, capturing representative security objectives extracted from the CIAA classification, and provide them as a reusable formal model library.
  - ***C5.4***. *Security policies*: we developed a set of policy models capturing abstract security mechanisms to be used as a reusable model library to *mitigate* security threats from ***C5.2*** or *satisfy* security objectives from ***C5.3***.

RO2 is addressed with the following contributions:

- ***C6***. *The design of a novel approach for the specification and analysis of secure software architectures using reusable (formal) models.*

  - ***C6.1***. *Design and analyses of secure software architecture by reuse:* these property specifications are included as part of security requirements libraries, so that they can be reused in other concrete instantiations of systems built using component-port-connector architectures and message passing communication. For instance, we provided a set of facilities to type the corresponding software architecture model elements.

  - ***C6.2***. *Integration in existing SDLC*: we studied the support of existing SDLC providing a set of guidelines as a supplement activities in the requirements specification and architecture design phases.

- ***C7***. *Demonstration*: we illustrate our research methodology through a simple example of a college library web application system. In addition, to validate our work, we apply the proposed approaches in a case study of *Smart Meter Gateway*.

RO3 is addressed with the following contributions:

- ***C8***. *MDE tool chain for the design*: we implemented a tool suite using the Eclipse platform to support the proposed approaches from ***RO1*** and ***RO2***. We provides features to model the software architecture model (***C2.1***), the reusable property model libraries (***C2.2***), the reuse of these libraries, the generation of the interpretation of these models in a tooled formal language and the generation of new artefacts related to the security analysis of the architecture model.

- ***C9***. *Automated analysis*: we propose to use existing tooled formal languages (Alloy and Coq) to support the design and analysis of a secure software architecture models, including the architecture models (***C4.1***) and property model libraries (***C4.2***, ***C5***). The results of the analysis are then used by the MDE tool chain for the generation of new artefacts, including feedback (detection of security issues, solutions suggestion) and validation artefacts (threats report, objectives report, policies report).

A summary of the relations between contributions and the research objectives they fulfil are drawn in Table 1.1. We also associate to each contribution the chapter in which it is presented in this manuscript and if applicable some results as they are published in journals and conferences.

Research Objectives	Contributions	Chapters	Publications
RO1	***C1***	**Chapter 3, 4, 5, 6**	[114, 115, 116, 117, 113]
	C2.1	**Chapter 4**	[114, 115]
	C2.2	**Chapter 5, 6**	[113, 116, 117]
	C3.1	**Chapter 4**	[114, 115]
	C3.2	**Chapter 5, 6**	[113, 116, 117]
	C4.1	**Chapter 4**	[114, 115]
	C4.2	**Chapter 5, 6**	[113, 116, 117]
	C5.1	**Chapter 4**	[114, 115]
	C5.2	**Chapter 5**	[116, 117]
	C5.3	**Chapter 6**	[113]
	C5.4	**Chapter 5, 6**	[113, 116, 117]
RO2	***C6.1***	**Chapter 3, 4, 5, 6**	[114, 115, 116, 117, 113]
	C6.2	**Chapter 3, 4, 5, 6**	[117, 113]
	C7	**Chapter 4, 5, 6, 7**	[114, 115, 116, 117, 113]
RO3	***C8***	**Chapter 4, 5, 6**	[115, 116, 117]
	C9	**Chapter 4, 5, 6**	[115, 116, 117]

Table 1.1: Research objectives and results

1.5 Publications

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

In addition to the Quartile SJR Scimago (http://www.scimagojr.com), the impact factor measurement (Impact Factor), we use the Core classification (http://www.core.edu.au) to indicate the known rank of publications..

Fundamental publications

- **Paper A.** [114] **Quentin Rouland**, Brahim Hamid, and Jason Jaskolka. "Formalizing reusable communication models for distributed systems architecture". In: International Conference on Model and Data Engineering (MEDI). Springer. 2018, pp. 198–216.
  doi: https://doi.org/10.1007/978-3-030-00856-7_13.
  Summary: In this article, we aim at specifying software architecture of distributed systems using an approach combining semi-formal and formal languages to build reusable model libraries to represent communication solutions. We provide a set of reusable connector libraries within a set of properties to define architectures for systems with explicit communications models like message passing and remote procedure calls that are common to most distributed systems.

- **Paper B.** [113] **Quentin Rouland**, Brahim Hamid, Jean-Paul Bodeveix, et al. "A Formal Methods Approach to Security Requirements Specification and Verification". In:2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS) (**Rank A**). IEEE. 2019, pp. 236–241.
  doi: https://doi.org/10.1109/ICECCS.2019.00033.
  Summary: In this work, we propose an integrated approach for security requirement specification and treatment during the software architecture design time. The general idea of the approach is to: (1) specify security requirements as properties of a modelled system in a technology-independent specification language; (2) implement the developed model in a suitable language with tool support for requirement satisfaction through model verification; and (3) suggest a set of security policies to constrain the operation of the system and to guarantee the security properties. To validate our work, we explore a set of representative security properties from cate-

gories based on CIA classification in the context of secure component-based software architecture development.

- **Paper C.** [115] **Quentin Rouland**, Brahim Hamid, and Jason Jaskolka. "Formal specification and verification of reusable communication models for distributed systems architecture". In: Future Generation Computer Systems (**Rank A, SCIMAGO SJR Q1, Impact Factor: 5.768**) 108 (2020), pp. 178–197.
  doi: https://doi.org/10.1016/j.future.2020.02.033.
  Summary: In this article, we build reusable model libraries to specify and verify communication styles for modeling software architectures of distributed systems. First, we propose a meta-model to describe high-level concepts of architecture in a component–port–connector fashion focusing on different communication styles. Then, we formalize those concepts and their semantics following some properties (specifications) to check architectural conformance. To validate our work, using a developed tool to support our approach, we provide a set of reusable connector libraries within a set of properties to define architectures for systems with explicit communication models that are common to most distributed systems including message passing, remote procedure calls, and distributed shared memory.

- **Paper D.** [116] **Quentin Rouland**, Brahim Hamid, and Jason Jaskolka. "Reusable Formal Models for Threat Specification". In: ICSR 2020: Reuse in Emerging Software Engineering Practices (**Rank B**). Springer. 2020, pp. 52–68.
  doi: https://doi.org/10.1007/978-3-030-64694-3_4.
  Summary: In this work, we propose an integrated approach for threat specification, detection, and treatment in component-based software architecture models via reusable security threat and requirement formal model libraries. Our solution is based on metamodeling techniques that enable the specification of the software architecture structure and on formal techniques for the purposes of precise specification and verification of security aspects as properties of a modeled system. In addition, we use model-driven engineering techniques for the development of a tool suite to support our approach.

- **Paper E.** [117] **Quentin Rouland**, Brahim Hamid, and Jason Jaskolka. "Specification, detection, and treatment of STRIDE threats for software components: Modeling, formal methods, and tool support". In: Journal of Systems Architecture (**Rank B, SCIMAGO SJR Q2, Impact Factor: 2.552**) (2021).
  doi: https://doi.org/10.1016/j.sysarc.2021.102073.

Summary: In this paper, we propose integrated approach for threat detection and treatment by means of security requirements, during the software architecture design time. The general idea of the approach is to: (1) specify threats as properties of a modeled system in a technology-independent specification language; (2) express conditions that reveal these threats in a suitable language with automated tool support for threat detection through model verification; and (3) suggest a set of security requirements to protect against detected threats. To validate our work, we explore a set of representative threats from categories based on Microsoft's STRIDE threat classification in the context of secure component-based software architecture development.

Publications related to the thesis

- **Paper F.** [47] Brahim Hamid, **Quentin Rouland**, and Jason Jaskolka. "Distributed Maintenance of a Spanning Tree of k-Connected Graphs". In: 2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC) (**Rank B**). IEEE. 2019, pp. 209–217.
  doi: https://dx.doi.org/10.1109/PRDC47002.2019.00052.
  Summary: This work is devoted to the problem of spanning trees maintenance in the presence of crash failures in a distributed environment using only local knowledge. Using a pre-constructed spanning tree of a k-connected graph, we present a protocol to maintain a spanning tree in the presence of k-1 consecutive failures. In addition, we investigate the possible specification and verification of the presented algorithm using Alloy as a tooled formal language for an implementation of this protocol in the asynchronous message passing model.

Distinctions

1. **Best paper award. Paper D.** [116] "Reusable Formal Models for Threat Specification". ICSR 2020.

1.6 Thesis outline

The outline of the dissertation is as follows. Chapter 2 presents the technical frameworks used in of our work. Chapter 3 is dedicated to present a methodology for the creation of a design and analysis framework to assist software architect in the design process. Chapter 4 proposes a component-based software architecture design and analysis framework as a building block to express the security concerns. Chapter 5 presents a security threats design and analysis framework. Chapter 6 introduces a security objectives design and analysis framework. An illustration of the use of the aforementioned frameworks within a case study of a *Smart Meter Gateway* and discussion on the feasibility and the potential applications of the proposed approach, as well as the potential for its generalization and extension are presented in Chapter 7. Finally, Chapter 8 concludes the dissertation and proposes some future works and perspectives.

Chapter 2

Technical frameworks

Contents

2.1 Introduction

Models are used to denote some abstract representation of software-based computing systems and the way they are developed. Specifically, we need models to encode the artifacts and software platforms, models to represent the process activities, to test, to simulate and to validate the proposed solutions. Accordingly, comprehension, study and analysis of computer systems and system engineering processes require models which make it as easy as possible to express and to encode them. As a benefit, the study of problems on high-level models enables us to deduce some properties on other less abstract models.

This chapter present elements of the formal and technical frameworks and some tools we use in the context of our research objectives stated in Section 1.4. We review the most related work and we give the place of model-driven engineering and domain-specific language, formal methods, analysis techniques and tool support regardless of each of the studied research topics. In addition, we present an introduction to the case study that might prove useful in understanding our approach. Reader already familiar with these concepts and definitions may safely proceed to subsequent chapters and consult the definitions and notations herein only as needed.

2.2 Software systems 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 ISI/IEC/IEEE 24765:2010 [2] Software Engineering as follows: *software engineering is 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. 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 Component Based Software Engineering (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 [133].

Szyperski gives the following definition of a component [131]: *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 [28], KobrA [8], Fusion (Coleman 1993), OPEN process framework [36]. Flex-eWare [60] 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 [10] 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 flexible to a lot of 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 [24]. To explain CBD processes, reference development phases corresponding to four abstraction levels of a system model are identified 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 [10]. 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

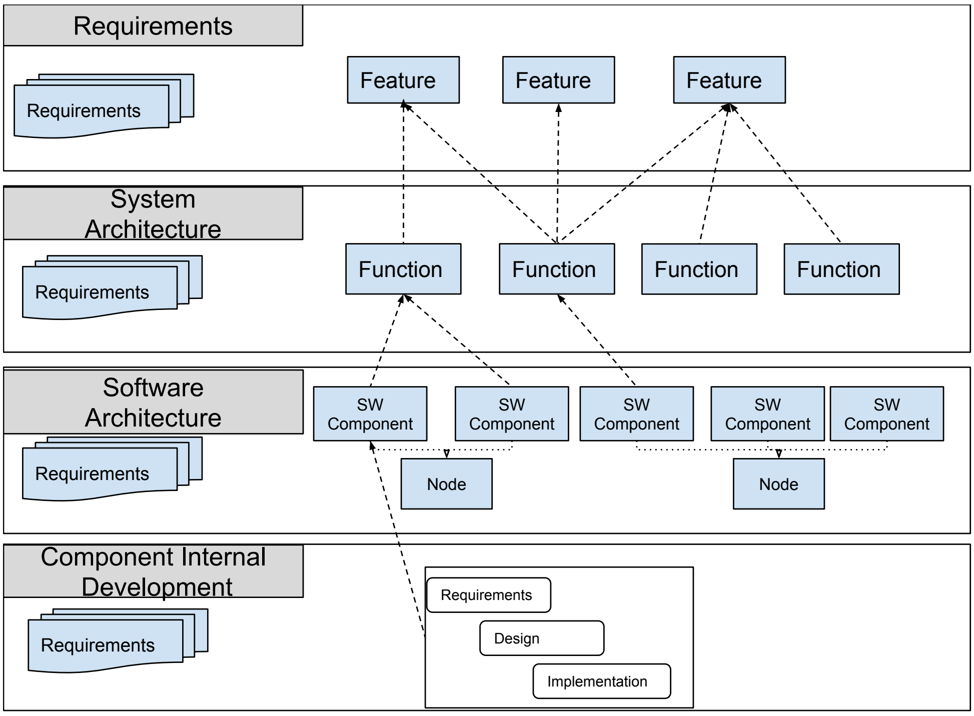


Figure 2.1: Component-Based Software Development Process and Used Artifacts [10]

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 CBD methods using UML such as Catalysis [28], each component is designed and implemented internally as a set of classes that implement the interfaces required externally. In addition, internal interactions are realized.

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. Model-Based Engineering (MBE) based solutions seem very promising to meet the needs of secure and dependable applications development. The idea promoted by MBE is to use models at different levels of abstraction for developing systems. In other words, models provide input and output at all stages of system development until the final system itself is generated.

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 [88].

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) [95], 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).

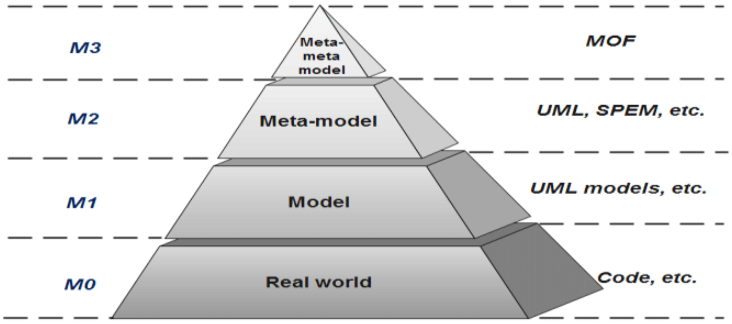


Figure 2.2: Modeling pyramid of the OMG

2.4.2 Model-Driven Engineering (MDE)

The concept of a model is becoming a major paradigm in software engineering. Its use represents a significant advance in terms of level of abstraction, continuity, generality, scalability, etc. Model-Driven Engineering (MDE) is a form of generative engineering [121], in which all or a part of an application is generated from models. MDE is a promising approach since it offers tools to deal with the development of complex systems improving their quality and reducing their development life cycles. The development is based on model approaches, metamodeling, Model-To-Model transformations, development processes and execution platforms. The advantage of having an MDE process is that it clearly defines each step to be taken, forcing the developers to follow a defined methodology. MDE allows to increase software quality and to reduce the software systems development life cycle. Moreover, from a model it is possible to automatize steps by model refinements and generate code for all or parts of the application.

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

The development process cycles are mainly iterative, resulting in different levels of model refinement from analysis to design. There are implementation platforms that address these issues in a specific context (e.g., the MDA standard [17]), but in many other contexts, the links between models refined or processed to solve references (to non-existent elements, elements not referenced, created elements, etc.) are still solved in ad hoc manner, without adequate support from generic technologies.

2.4.3 Domain Specific Modeling Language (DSML)

In software engineering, Domain Specific Modeling (DSM) [33, 38] is a methodology that uses models to specify applications within a particular domain. Domain Specific Modeling Languages (DSMLs) are languages that are specifically tailored to the needs of a particular problem or application domain. Domain experts can understand, validate, modify, test, and sometimes even develop such languages. Domain Specific Modeling Languages (DSMLs) are frequently used in MDE [123].

A language is defined by an abstract syntax, a concrete syntax and the description of semantics [49, 33, 66]. The abstract syntax defines the concepts and their relationships which are often expressed by a metamodel. On the one hand, the concrete syntax defines the appearance of the language. A grammar or set of regular expressions is often used to design the concrete syntax. On the other hand, semantics defines the sense and meaning of the structure by defining sets of rules. Domain Specific Modeling (DSM) in software engineering is used as a methodology using models as first-class citizens to specify applications within a particular domain. The purpose of DSM is to raise the level of abstraction by only using the concepts of the domain and hiding low level implementation details [38]. A Domain Specific Language (DSL) typically defines concepts and rules of the domain using a metamodel for the abstract syntax, and a concrete syntax (textual, tree-structured, tabular, diagrammatic, etc.). Domain Specific Language (DSL) allow specifying systems in a domain-friendly manner. Most metamodels and/or abstract syntaxes offer one or more concrete syntaxes to instantiate their concepts. The UML standard, for example, provides concrete syntaxes with diagrams for different viewpoints, in a graphical manner with icons and links. Other metamodels, and especially domain-specific modeling languages, often come with a textual syntax. As we shall see, processes in DSM reuse a lot of practices from MDE, for instance, metamodeling and transformation techniques.

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 engineered, 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 Systems Security Engineering. 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 Model-Driven Engineering 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 [70], reviews by security specialized companies [80, 135] or by being inspired by industrial projects [79]. An overview will also be given on further standards or approaches.

Microsoft SDL. Microsoft's Security Development Life cycle [82] 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) [105] by the Open Web Application Security Project (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 [78] 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 [79] 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 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 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 [40, 41], 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 [101, 126] 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 [87], SecureUML [75] and UMLsec [62, 63] 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 [37]. 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 literature. In [74, 73], the authors propose an integration model for integrating security engineering approaches into software life cycle standards, mapping the concepts of the software life cycle (Institute of Electrical and Electronics Engineers (IEEE) 12207) to security engineering concepts (a set of concepts collected from various security engineering approaches [74]). 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 [140] 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

2.6.1.1 Finite State machine (FSM)

Finite State machines (FSM) [138] are a type of system model in which the output is determined by the complete history of the system's inputs, rather than just the most recent input. State machines have a performance that is defined by their history, as opposed to strictly functional systems, where the output is only decided by the input. It is an abstract machine that can be in exactly one of a finite number of states at any given time. In response to some inputs, the FSM can transition from one state to another; this is referred to as a transition. A set of states, the beginning state, and the inputs that trigger each transition describe an FSM.

A finite state machine can be described by a state-transition table, which shows the transitions between each conceivable state (depending on the machine's inputs) and the outputs that come from each input. Table 2.1 shows an example of a turnstile state-transition table.

current state input	Locked	Unlocked
push	Locked	Locked
coin	Unlocked	Unlocked

Table 2.1: State-transition table of turnstile

The turnstile state-transition table can be read as:

- If the current state is Locked and Push input occurred then, the state stays Locked, i.e., a locked turnstile remains locked when pushed
- If the current state is Locked and Coin input occurred then, the state changes to Unlocked, i.e., a locked turnstile unlocks when money is inserted
- If the current state is Unlocked and Push input occurred then, the state changes Locked, i.e., an unlocked turnstile locks after its used

- If the current state is Unlocked and Coin input occurred then, the state stays to Unlocked, i.e., unlocked turnstile remains unlocked if you insert more money

A finite state machine can alternatively be represented as a state diagram, which is a directed graph. A node is used to represent each state (circle). Transitions from one state to another are represented by edges (arrows). The input that causes each transition is labeled on each arrow. A circular arrow returning to the original state represents an input that does not create a state change. The start state is indicated by an arrow pointing to the associated node, but it is otherwise disconnected. Figure 2.3 shows an example of a turnstile finite state machine. This representation is equivalent to the turnstile state-transition table depicted in Table 2.1.

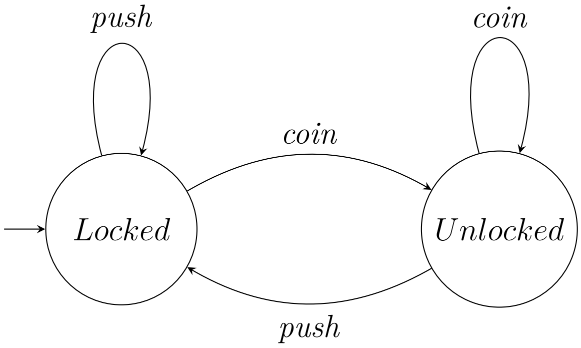


Figure 2.3: States of turnstile

2.6.1.2 First order logic (FOL)

First-order logic (FOL) [128], 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: $\top$
- 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. Predicate is applied to a set of arguments and get a proposition that is either true or false.

For example :

$$is\_the\_husband\_of(women, man)$$

is a predicate indicating that man is the husband of $women$.

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 :

$$man1 = man2$$

is a predicate indicating that $man1$ and $man2$ 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 :

$$get\_wife(man)$$

is a function that return the $wife$ of man.

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 :

$$\exists\ x\ \cdot\ have\_a\_husband(x)$$

is a proposition that is true if some x exists such as the predicate $have\_a\_husband(x)$ is true, i.e., if some x have a husband.

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 :

$$\forall\ x\ \cdot\ have\_a\_wife(x)$$

is a proposition that is true if for all x the predicate $have\_a\_wife(x)$ is true, i.e., if all x have a wife.

2.6.1.3 Modal Logic (ML)

Modal logic (ML) [21] 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: $\vdash A \Rightarrow \Box A$

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 [134], obligation [137], and temporal [29] 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 know that Alice has a husband" as:

$$\mathbb{K}_{bob}(have\_husband(alice))$$

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 wife" as:

$$\mathbb{F}(have\_wife(bob))$$

2.6.2 Formal Techniques

In order to fully cover all aspects of the approach, we focus on formal languages with tool support that should meet the following requirements with respect to the proposed approach (Chapter 3):

1. Enable the creation of a formal architectural metamodel;
2. Enable the creation of an architectural system architecture model, describing the target application model according to the metamodel;
3. Support the specification and verification of concerns and solutions properties of the system model based upon the architecture as reusable formal model libraries; and
4. Support the reuse of the resulting formal model libraries during the creation of a target application model.

While any suitable formal language with tool support such as nuSMV, SPIN, UPPAAL, etc., can be used for modeling and verification, we choose to use Alloy [57] for most of the analysis (Chapter 4, Chapter 5, Chapter 6) because of the simplicity and the straightforward usage of its analyzer. The Alloy Analyzer supports visualizing models and verifying their static properties and dynamic properties (e.g., behavioral aspects). It uses a constraint solver providing automatic simulation and checking to find model instances satisfying the constraints defined during the model specification process. This makes it an appropriate candidate for our intended research work, i.e., modeling reusable models. In our work, the Alloy Analyzer [1] essentially acts as a model checker and counterexample generator. This enables us to construct models incrementally, allowing rapid iterations between modeling and analysis when writing a specification. A recent article [58] has further highlighted the strengths of Alloy for software design. These strengths provide additional motivation and justification for adopting Alloy in this work. An architect can instruct the

Alloy Analyzer to verify whether the property of the system design holds, which involves exhaustively exploring every model instance within a specified scope (upper-bound). If it does not hold, a counterexample will be generated which can be visualized. The absence of counterexamples guarantees that the property holds in the modeled system, within the *specified scope*. As claimed in [57], most counterexamples are found in a reasonably small scope.

Note, however, even if Alloy provides all requirements needed and have a lot strengths, we must be confident that the scope is appropriate to validate a property. For critical applications we could argue that could be a problem. Therefore, practitioners would prefer to use theorem proving for the formal verification (e.g., Coq [12], Event-B [5]). In our case, we present in Chapter 6 an alternative solution using Coq [12]. Coq is a formal proof management system. It includes a formal language for writing mathematical definitions, executable algorithms, and theorems, as well as a development environment for semi-interactively developing machine-checked proofs. In the context of our work, we use Coq as proof assistant to specify the desired properties of the modeled system, the formulation of theorems stating relations between properties; and to support proving these relations semi-automatically using the proof assistant. However, using this environment is more complex with less automation for tool support. It means it will affect our fourth requirement: supporting the reuse. Ultimately, using a proof assistant will be more costly for both the development of the reusable model (complexity) and offer more limited reuse, but will give a proof not bound to any scope.

2.6.2.1 Alloy

Alloy is a lightweight formal modeling language based on the first-order relational logic [57]. It was deeply inspired by Z [129] and influenced by different object-oriented modeling languages such as UML. An Alloy model is composed of a set of signatures each defining a set of atoms. Atoms may have fields which define relations between atoms. In addition, signatures serve as types, and subtyping may be defined as signature extension. There are several ways to specify constraints in the model. One is to treat them as *Facts* that should hold at all times. Another is to treat them as *Predicates* defined in the form of parameterized formulas that can be used elsewhere and as *Assertions* that are intended to follow from the facts of a model. In some situations, functions in the form of parameterized expressions may be used as helpers in the specification and verification processes.

The architect can instruct the Alloy Analyzer to verify whether the property *prop* of the system design holds, with the command: **check** *prop* **for** n, which would exhaustively explore every model instance within a scope of n, i.e., exhaustively explore every model instance to the upper-bound n representing the number of atoms typed by each signature. In the context of our work, the generated models have a maximum of n component instances, n connector instances, etc. If the property does not hold, a counterexample will be generated which can be visualized. The absence of counterexamples guarantees that the property holds in the modeled system, within the specified scope. As claimed in [57], most counterexamples are found in a reasonably small scope. We include a thorough introduction to this language in the following of the section.

Modules. A metamodel can be created in Alloy using one or more Alloy modules, with each module having its own (.als) code.

An Alloy module can begin with a module declaration, which allows elements declared here to be reused by other modules. The keyword **module** is accompanied by a relative path to the file in which the module is found in the module declaration. This path specifies the reach of the module's importability, i.e., it can be used to import any other Alloy module found in the path's core. The **open** keyword is then used, followed by the relative path of the module to import, to perform an import. Listing 2.1, shows an example of module declaration and importation.

```
module family
open util/ordering[Person]
```

Listing 2.1: Module declaration & importation example in Alloy

All constructs declared in a module can be used after it is imported, including signatures, fields, facts, predicates, functions, assertions, let expressions, and commands. All of these terms will be described in the following sections.

Instances. The Alloy Analyzer is a tool that, given an Alloy module, produces a set of Alloy instances, or models whose elements are typed by concepts and relations of the metamodel identified by the Alloy module and that satisfies the module's constraints. If an instance is obtained by verifying an argument, it is referred to as a counter-example. Counter-examples are situations where the verified statement is broken. The following sections include comprehensive details on the declarable structures that make up an Alloy module, as well as guidance on how they affect the Alloy Analyzer's analysis.

Signatures & fields. An Alloy module is mainly made up of signatures and fields, which describe the collection of elements that can be used to create any Alloy instance that the Alloy analyzer can find: Instances are made up of atoms, which are non-dividable entities whose form is determined by signatures, and atom tuples, which are defined by fields. The keyword **sig** is used to declare a signature, which is preceded by the signature's name and a block containing a series of field declarations that relate this signature to others. A signature can also be used to define subtypes by extending another signature.

Any signature declaration can be preceded by a series of modifiers:

- *Multiplicity*: Keywords like **lone**, **one**, and **some** compel the declared signature's number of atoms to be at most one, exactly one, or at least one, respectively. The signatures can type any number of atoms if the multiplicity keyword is not used.

- *Abstract*: Keyword **abstract** ensures that no atom is typed directly by that signature. Note that if the signature which this modifier is applied is not extended, the analyzer will ignore it.

A field's declaration consists of an identifier label, accompanied by a series of arrow-separated signatures. At each arrow's end, the multiplicity can be defined.

An example of defining the concepts and relations of a family is provided in Listing 2.2.

```
abstract sig Person {
    father: lone Man,
    mother: lone Woman
}
sig Man extends Person {
    wife: lone Woman
}
sig Woman extends Person {
    husband: lone Man
}
```

Listing 2.2: Example of signatures & fields declaration in Alloy

This module's instances will be made up of atoms with either the *Man* or the *Woman* signatures (as Person is **abstract**), as well as tuples with either the mother or the father fields. Figure 2.4 shows an example of this type of situation. As this example shows, structural knowledge alone is insufficient to accurately identify families. Constraints are used to impose a collection of properties on instances, such as the fact that a woman cannot be both her own mother and her father's mother. We will show how such constraints can be enforced and/or verified in the following sections.

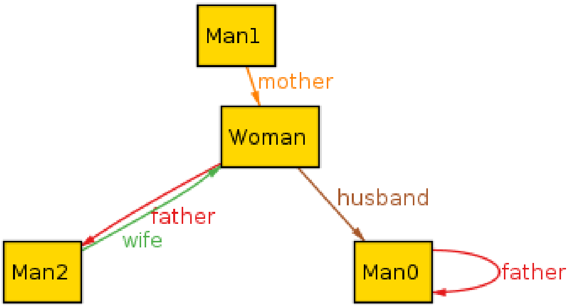


Figure 2.4: An Alloy instance obtained from Listing 2.2

Facts. Facts are used to specify constraints. Constraints are properties that must always hold in instances of the Alloy module in which they are declared. A fact declaration begins with the keyword **fact**, followed by a block containing a boolean-valued Alloy expression and, optionally, a name (only for documentation purposes). The fact depicted in Listing 2.3, for example, will prohibit instances of the Alloy module in Listing 2.2 from containing people who are their own ancestors.

```
fact noPersonIsHisOwnAncestor {
    no p:Person | p in p.@mother or p in p.@father
}
```

Listing 2.3: Fact declaration example in Alloy

It is possible to declare a signature fact when defining invariants specific to a given signature. A signature reality is represented by a block that follows a chosen signature and serves as its meaning. By context, we mean that the fact's invariant applies to any atom typed by the given signature. In Listing 2.4, we present a refined version of Listing 2.3, this time expressed as a *Person* signature fact. Note that the keyword **this** is used to refer to the context (any given person) in a signature reality, while the operator **@** is used to refer to fields declared outside of the context in the signature *Person*.

```
abstract sig Person{
    mother: lone Woman,
    father: lone Man
}{
    not(this in p.@mother or this in p.@father)
}
```

Listing 2.4: Example of an alternative way to declare a fact on a given signature

Functions. Functions are set-valued (return a set of atoms or tuples of atoms) Alloy expressions that can be parameterized. The keyword **fun** is used to declare function, which is accompanied by an identifier, optional parameters, and a block containing Alloy expressions.

In Listing 2.5, we show an example of parameterized function returning the set of persons who are the parents of the person given in parameter.

```
fun getParents[p:Person]: set Person {
    p.mother + p.father
}
```

Listing 2.5: Function declaration example in Alloy

Predicates. As functions, predicates are Alloy expressions that can be parameterized but boolean-valued. The keyword **pred** is used to declare predicates, which is accompanied by an identifier, optional parameters, and a block containing Alloy expressions. In Listing 2.6, we show an example of predicate without parameters returning true if at least one wife exists.

```
pred AtLeastOneWife {
    some m:Man | m.wife != none
}
```

Listing 2.6: Predicate declaration example in Alloy

Assertions. In the sense that they are only used in commands, assertions are special predicates which are declared with the keyword **assert**. Listing 2.7 shows an example of an assertion.

```
assert WifeRequireAtLeastOneWomenandOneMan {
    AtLeastOneWife implies #Man <= 1 and #Women <=1
}
```

Listing 2.7: Assertion declaration example in Alloy

Commands. To be able to produce instances for a given Alloy module, the Alloy Analyzer requires an order. It begins with a keyword that describes the type of analysis to be performed:

- **run**: create a sample of instances where the predicate holds, provided a predicate

- **check**: given a statement (assertion), produce a sample of counter-examples that contradict the assertion

.

Both types of commands should provide details about the domain space in which the analysis will be conducted, with the Alloy analysis being decidable only because it is done on a finite domain. This is accomplished by assigning a scope to each module signature, which is an upper limit on the number of atoms typed by each signature.

The scope for all signatures is set to 3 by default. The keyword **for** can be used to change the global reach. The **but** keyword can also be used to assign a scope to each signature individually or to include both global and unique scopes. Listing 2.8 demonstrates how to use commands.

```
run {} for 4 Man, exactly 2 Woman // Seeking instance with 0 to 5 Man and exactly 2 Women
run AtLeastOneWife for 5 // Seeking instance with 0 to 5 Person where predicate  AtLeastOneWife hold
check WifeRequireAtLeastOneWomenandOneMan for 10 // Seeking counter-example with 0 to 10 Person
```

Listing 2.8: Commands declaration example in Alloy

2.6.2.2 Coq

Coq [12] works as a proof assistant. It is intended for the creation of mathematical proofs, particularly formal specifications, programs, and proofs that programs conform to their specifications. In Coq, the same language (Calculus of Inductive Constructions) is used to formalizes properties, programs, and proofs. All logical judgments are typing judgments, i.e., Coq is a type-checking algorithm.

Language. The Prop sort and the Type sort are the two categories in which Coq objects are sorted :

- *Prop* : Prop is the sort for propositions, which means that well-formed propositions are of this type.

  Typical propositions are shown in Listing 2.9.

```
    ∀ A B : Prop, A ∧ B → B ∨ B
    ∀ x y :  Z, x ∗ y = 0 → x = 0 ∨ y = 0
```

Listing 2.9: Coq example propositions

New predicates can be defined either inductively, as in Listing 2.10, or by abstracting over other existing propositions, as in Listing 2.11.

```
Inductive even : N → Prop :=
| even_0 : even 0
| even_S n : odd n → even (n + 1)
with odd : N → Prop :=
| odd_S n : even n → odd (n + 1).
```

Listing 2.10: Coq example inductive predicate

```
Definition divide (x y:N) := ∃ z, x ∗ z = y.
Definition prime x := ∀ y, divide y x → y = 1 ∨ y = x.
```

Listing 2.11: Coq example definition

- *Type* : Type is the sort for data types and mathematical structures, which means that well-formed types and structures are of this type.
  A basic type is shown in Listing 2.12.

```
Z → Z ∗ Z
```

Listing 2.12: Coq example basic type

Types can also be inductive structures, as in Listing 2.13, or functions over inductive types are expressed using a case analysis, as in Listing 2.14.

```
Inductive nat : Set :=
| 0 : nat
| S : nat → nat.

Inductive list (A:Type) : Type :=
| nil : list A
| cons : A → list A → list A.
```

Listing 2.13: Coq example inductive type

```
Fixpoint plus (n m:nat) {struct n} : nat :=
    match n with
    | O → m
    | S p → S (p + m)
    end
    where ”p + m” := (plus p m).
```

Listing 2.14: Coq example function over inductive type

Assumption. Assumptions extend the global environment with axioms, parameters, hypotheses or variables. An example is given in Listing 2.15.

```
Parameter X Y : Set.
Parameter (R : X → Y → Prop) (S : Y → X → Prop).
Axiom R_S_inv : ∀ x y, R x y ↔ S y x.
```

Listing 2.15: Coq example simple assumptions

Proving. In Coq, proof development is done using a tactic-based language that allows for a user-guided proof process. For example, the **intros** tactic is used to introduce variables appearing with ∀ as well as the premises (left-hand side) of implications. If the goal contains universally quantifiable variables (i.e., ∀), we can use the intros tactic to incorporate those variables into the context. All hypotheses on the left side of an implication can alternatively be introduced as assumptions using intros. If **intros** is used alone, Coq will introduce all of the variables and hypotheses it can and will name them automatically. By supplying the names in order, we can offer your own names. There is a sister tactic **intro** that only introduces one thing.

For example, if we try to prove a *modus_tollens* theorem:

$$(P \wedge Q) \rightarrow P$$

We can start by introducing the variables, as well as the hypotheses, using **intro**. Then, we obtain the following hypothesis:

- *P_implies_Q* : $P \rightarrow Q$
- *not_Q*: $\neg Q$

Then, we can continue the proof by applying other tactics to these terms until the proof is complete.

Listing 2.16 show the corresponding Coq for this example.

```
Theorem modus_tollens : ∀ (P Q : Prop),
  (P → Q) → ¬Q → ¬P.
Proof.
  intros P Q P_implies_Q not_Q.
```

Listing 2.16: Coq tatic intro

A short introduction to set of tactics relevant to the dissertation is available in Appendix B.1.

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.

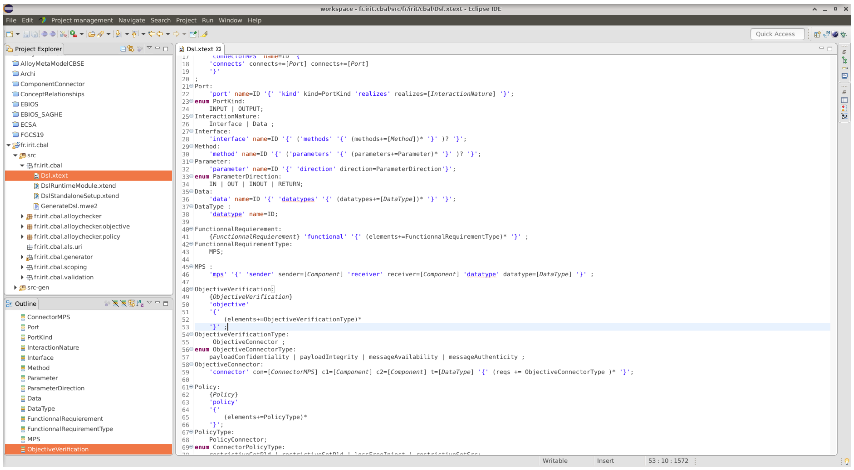
2.7.1 Eclipse Modeling Framework

There are several Domain Specific Modeling (DSM) environments, one of them being the open-source Eclipse Modeling Framework (EMF) [130]. Eclipse Modeling Framework (EMF) provides an implementation of Essential MOF (EMOF), a subset of Meta-Object Facility (MOF), called Ecore2. EMF offers a set of tools to specify metamodels in Ecore and to generate other representations of them, for instance Java. In our context, we use the Eclipse Modeling Framework. Note, however, that our vision is not limited to the EMF platform. Here, we outline the different Eclipse tools used in the development of the DSLs to support the modeling of the Security and Dependability (S&D) artifacts, the repository and its Application Programming Interfaces (APIs). Among the tools used here are cited:

- Eclipse is an open-source software project providing a highly integrated tool platform. The applications in Eclipse are implemented in Java and target many operating systems including Windows, Mac OSX, and Linux [130].
- EMF is a modeling framework and code generation facility for building applications based on a structured data model. In addition, EMF provides the foundation for interoperability with other EMF-based tools and applications [130].
- RCP plug-in allows developers to use the Eclipse platform to create flexible and extensible desktop applications upon a plug-in architecture [136, 77].
- Xtext [16] is a programming language and domain-specific language development framework. With Xtext, you may use a powerful grammar language to define your language. Xtext provides a set of APIs and DSLs to develop such DSLs easily. It not only gives you a generated parser but supports the full stack of infrastructure that is needed, which includes a parser, linker, type checker, compiler, and editing support for Eclipse, any editor that supports the Language Server Protocol.
- Xtend [16] is a statically typed programming language that produces understandable Java code. Xtend is based on the Java programming language in terms of syntax and semantics. However it improves on several aspects:

- Extension methods : add new capabilities to closed types
- Lambda Expressions : short syntax for anonymous function literals
- ActiveAnnotations : annotation processing on steroids
- Make your libraries even more expressive with operator overloading
- Type-based switching with implicit casts with powerful switch expressions
- Polymorphic method invocation a.k.a. multiple dispatch
- Template expressions - intelligent white space handling
- No statements : everything is an expression
- Properties shorthand for accessing and defining getters and setters
- Type inference : rarely need to write down type signatures anymore
- Full support for Java generics : including all conformance and conversion rules

These aspects made Xtend well suited to the task of code generation. Particularly the template engine that is optimized (e.g., automatic indentation support) to write code generation rules to a target language and is fully integrated into the Eclipse Integrated Development Environment (IDE) (e.g., auto completion for model data) and Xtend.

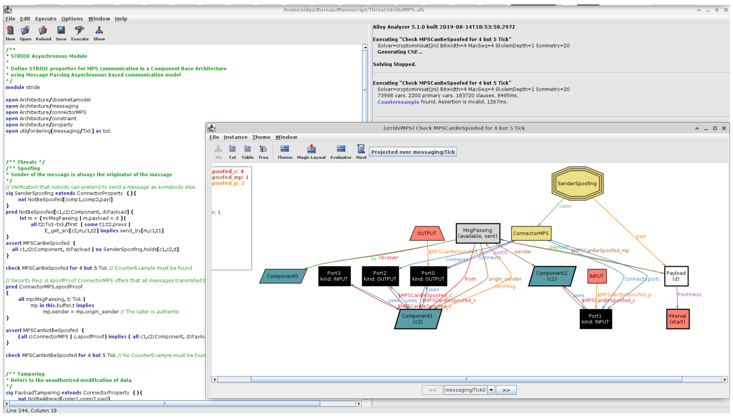


```
    'connects' connects+=[Port] connects+=[Port]
    '}'
;
Port:
    'port' name=ID '{' 'kind' kind=PortKind 'realizes' realizes=[InteractionNature] '}';
enum PortKind:
    INPUT | OUTPUT;
InteractionNature:
    Interface | Data ;
Interface:
    'interface' name=ID '{' ('methods' '{' (methods+=[Method])* '}' )? '}';
Method:
    'method' name=ID '{' ('parameters' '{' (parameters+=Parameter)* '}' )? '}';
Parameter:
    'parameter' name=ID '{' 'direction' direction=ParameterDirection'}';
enum ParameterDirection:
    IN | OUT | INOUT | RETURN;
Data:
    'data' name=ID '{' 'datatypes' '{' (datatypes+=[DataType])* '}' '}';
DataType :
    'datatype' name=ID;

FunctionnalRequierement:
    {FunctionnalRequierement} 'functional' '{' (elements+=FunctionnalRequirementType)* '}' ;
FunctionnalRequirementType:
    MPS;

MPS :
    'mps' '{' 'sender' sender=[Component] 'receiver' receiver=[Component] 'datatype' datatype=[DataType] '}' ;

ObjectiveVerification:
    {ObjectiveVerification}
    'objective'
    '{'
        (elements+=ObjectiveVerificationType)*
    '}' ;
ObjectiveVerificationType:
     ObjectiveConnector ;
enum ObjectiveConnectorType:
    payloadConfidentiality | payloadIntegrity | messageAvailability | messageAuthenticity ;
ObjectiveConnector:
    'connector' con=[ConnectorMPS] c1=[Component] c2=[Component] t=[DataType] '{' (reqs += ObjectiveConnectorType )* '}';

Policy:
    {Policy}
    'policy'
    '{'
        (elements+=PolicyType)*
    '}';
PolicyType:
    PolicyConnector;
enum ConnectorPolicyType:
```

Figure 2.5: Eclipse Modeling Framework

2.7.2 Alloy Analyser

The Alloy Analyzer [59] is a solver that takes a model's constraints and finds structures that meet those constraints. It can be used to both to study the model by creating sample structures and to test the model's features by creating counterexamples. Structures are graphically represented, and their appearance can be tailored to the domain in question.



```
/**
* STRIDE Asynchronous Module
*
* Define STRIDE properties for MPS communication in a Component Base Architecture
* using Message Passing Asynchronous based communication model
*/
module stride

open Architecture/cbsemetamodel
open Architecture/messaging
open Architecture/connectorMPS
open Architecture/constraint
open Architecture/property
open util/ordering[messaging/Tick] as tick



/** Threats */
/** Spoofing
* Sender of the message is always the originator of the message
*/
// Verification that nobody can pretend to send a message as somebody else.
sig SenderSpoofing extends ConnectorProperty { }{
	not NotBeSpoofed[comp1,comp2,payl]
}
pred NotBeSpoofed[c1,c2:Component, d:Payload] {
	 let m = {m:MsgPassing | m.payload = d }|
		all t2:Tick-tick/first  | some t1:t2.prevs |
			E_get_src[c2,m,c1,t2] implies send_by[m,c1,t1]
}
assert MPSCanBeSpoofed  {
   all c1,c2:Component, d:Payload | no SenderSpoofing.holds[c1,c2,d]
}

check MPSCanBeSpoofed for 4 but 5 Tick // CounterExample must be found

// Security Req: A spoofProof ConnectorMPS offers that all messages transmited b
pred ConnectorMPS.spoofProof
{
	all mp:MsgPassing, t: Tick |
		mp in this.buffer.t implies
			mp.sender = mp.origin_sender // The caller is authentic
}

assert MPSCanNotBeSpoofed  {
	(all c:ConnectorMPS | c.spoofProof) implies ( all c1,c2:Component, d:Paylo
}

check MPSCanNotBeSpoofed for 4 but 5 Tick // No CounterExample must be foun


/** Tampering
* Refers to the unauthorized modification of data.
*/
sig PayloadTampering extends ConnectorProperty { }{
	not NotBeAltered[comp1,comp2,payl]
```

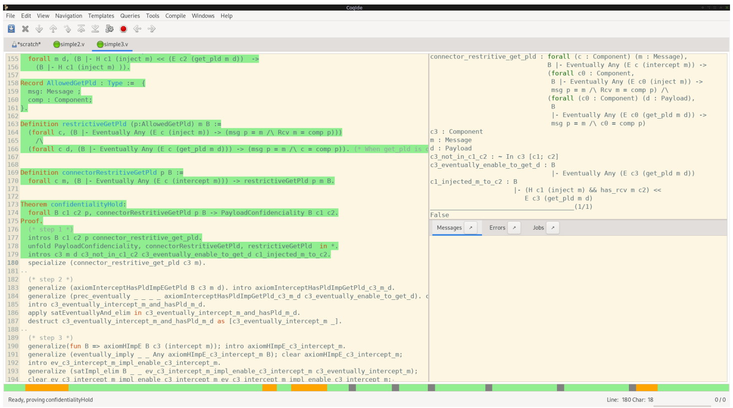
```
Line 244, Column 18
```

```
Alloy Analyzer 5.1.0 built 2019-08-14T18:53:58.297Z

Executing "Check MPSCanBeSpoofed for 4 but 5 Tick"
   Solver=cryptominisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   Generating CNF...

Solving Stopped.

Executing "Check MPSCanBeSpoofed for 4 but 5 Tick"
   Solver=cryptominisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   73998 vars. 2200 primary vars. 183720 clauses. 8495ms.
   Counterexample found. Assertion is invalid. 1267ms.
```

Figure 2.6: Alloy Analyzer

2.7.3 Coq IDE

The Coq IDE [12] is a graphical application that can be used to replace coqtop in a more user-friendly way. Its primary function is to allow the user to travel forward and backward through a Coq file, performing or reversing commands as needed.


CoqIde
File Edit View Navigation Templates Queries Tools Compile Windows Help
scratch
simple2.v
simple3.v
Messages ↗
Errors ↗
Jobs ↗

```
  forall m d, (B |- H c1 (inject m) << (E c2 (get_pld m d))  ->
    (B |- H c1 (inject m) )).

Record AllowedGetPld : Type :=  {
  msg: Message ;
  comp : Component;
}.

Definition restrictiveGetPld (p:AllowedGetPld) m B :=
  (forall c, (B |- Eventually Any (E c (inject m)) -> (msg p = m /\ Rcv m = comp p)))
    /\
  (forall c d, (B |- Eventually Any (E c (get_pld m d))) -> (msg p = m /\ c = comp p)). (* When get_pld is c


Definition connectorRestritiveGetPld p B :=
  forall c m, (B |- Eventually Any (E c (intercept m))) -> restrictiveGetPld p m B.


Theorem confidentialityHold:
  forall B c1 c2 p, connectorRestritiveGetPld p B -> PayloadConfidenciality B c1 c2.
Proof.
  (* step 1 *)
  intros B c1 c2 p connector_restritive_get_pld.
  unfold PayloadConfidenciality, connectorRestritiveGetPld, restrictiveGetPld  in *.
  intros c3 m d c3_not_in_c1_c2 c3_eventually_enable_to_get_d c1_injected_m_to_c2.
  specialize (connector_restritive_get_pld c3 m).
..
  (* step 2 *)
  generalize (axiomInterceptHasPldImpEGetPld B c3 m d). intro axiomInterceptHasPldImpGetPld_c3_m_d.
  generalize (prec_eventually _ _ _ _ axiomInterceptHasPldImpGetPld_c3_m_d c3_eventually_enable_to_get_d). c
  intro c3_eventually_intercept_m_and_hasPld_m_d.
  apply satEventuallyAnd_elim in c3_eventually_intercept_m_and_hasPld_m_d.
  destruct c3_eventually_intercept_m_and_hasPld_m_d as [c3_eventually_intercept_m _].
..
  (* step 3 *)
  generalize(fun B => axiomHImpE B c3 (intercept m)); intro axiomHImpE_c3_intercept_m.
  generalize (eventually_imply _ _ Any axiomHImpE_c3_intercept_m B); clear axiomHImpE_c3_intercept_m;
  intro ev_c3_intercept_m_impl_enable_c3_intercept_m.
  generalize (satImpl_elim B _ _ ev_c3_intercept_m_impl_enable_c3_intercept_m c3_eventually_intercept_m);
  clear ev_c3_intercept_m_impl_enable_c3_intercept_m ev_c3_intercept_m_impl_enable_c3_intercept_m;·
```

```
connector_restritive_get_pld : forall (c : Component) (m : Message),
                               B |- Eventually Any (E c (intercept m)) ->
                               (forall c0 : Component,
                                B |- Eventually Any (E c0 (inject m)) ->
                                msg p = m /\ Rcv m = comp p) /\
                               (forall (c0 : Component) (d : Payload),
                                B
                                |- Eventually Any (E c0 (get_pld m d)) ->
                                msg p = m /\ c0 = comp p)
c3 : Component
m : Message
d : Payload
c3_not_in_c1_c2 : ~ In c3 [c1; c2]
c3_eventually_enable_to_get_d : B
                                |- Eventually Any (E c3 (get_pld m d))
c1_injected_m_to_c2 : B
                      |- (H c1 (inject m) && has_rcv m c2) <<
                         E c3 (get_pld m d)
____________________________________(1/1)
False
```

```
Ready, proving confidentialityHold
Line:  180 Char: 18          0 / 0
```

Figure 2.7: Coq IDE

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 [106], 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:

- *Req_1. It should be possible for a user to visualize a book page.*
- *Req_2. It should be possible for the database to transmit data to the webserver.*
- *Req_3. It should be possible for the administrator to write a shared configuration variable in the database.*

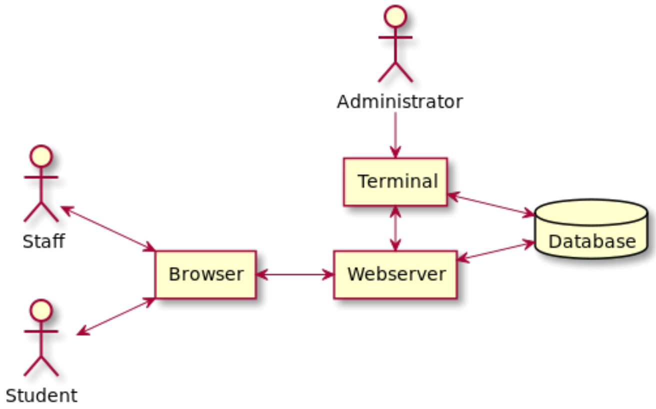


Figure 2.8: A college library web application example

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 software architecture. For instance, there are implicit design decisions that groups 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 connectors 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.

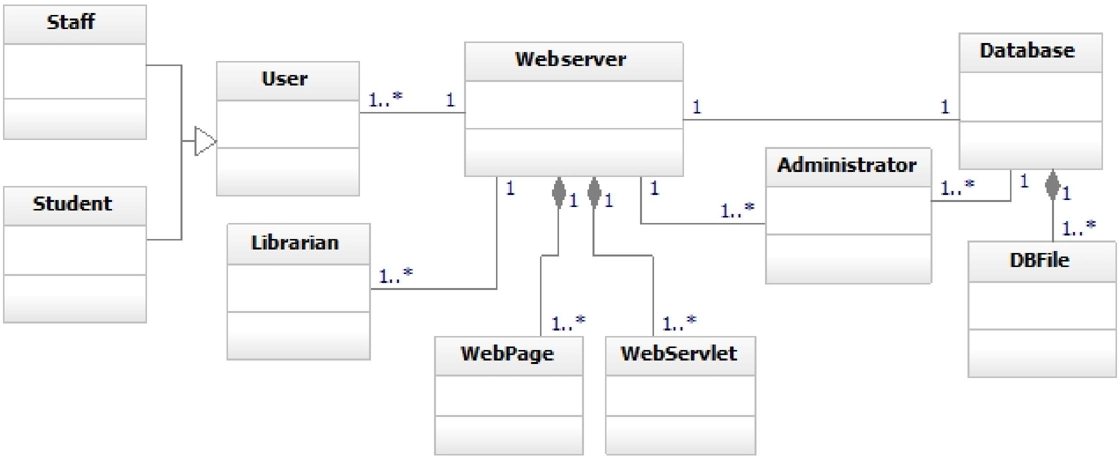


Figure 2.9: A UML description of the high-level architecture of the college library web application example

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 [18] 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 visualized 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.

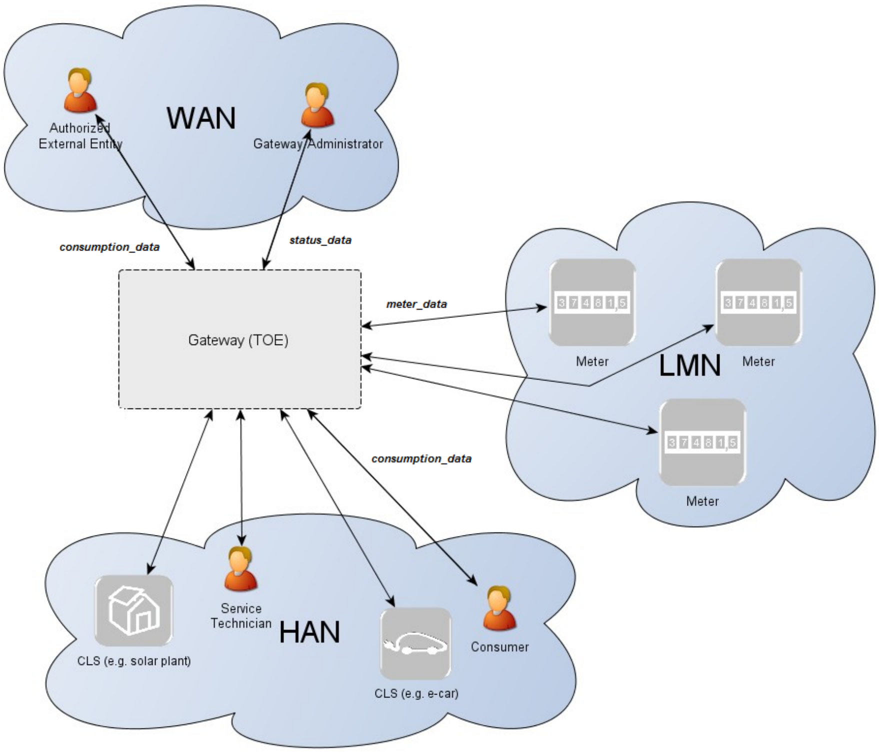


Figure 2.10: Actors and roles in the smart meter gateway scenario [18]

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

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.

Chapter 3

Approach

Contents

3.1 Introduction

In this chapter, we introduce our general conceptual vision to study security at architectural design level from the negative view (i.e., threats) and from the positive view (i.e., objective) in the context of component-based software architecture development. To implement this vision, we proposed a methodology for the creation of a design and analysis framework and provided some guidelines on how the resulted methodology may be used to achieve security within exiting SDLC methodologies. The proposed methodology will be then instantiated for our specific concerns: Section 4.3 for component-based & message passing communication architecture; Section 5.3 for security threats; and Section 6.3 for security objectives. With regard to our contributions, we deal with ***C1*** related to the Research Objective 1 (***RO1***) and ***C6*** related to the Research Objective 2 (***RO2***).

The remainder of the chapter is organized as follows. Section 3.2 presents the conceptual vision of the approach. Then, Section 3.3 describes the methodology to build a design and analysis framework. Section 3.4 presents how our approach could be integrated to existing system development life cycle. Finally, Section 3.5 concludes.

3.2 Conceptual vision

The goal of this research work is to develop a methodological tool support for the design and analysis of security concerns as a quality attribute that is becoming increasingly critical in current software-intensive systems. Notions such as properties, models, reuse and analysis can help in the development of well designed, properly modeled, accurately documented, and well-understood secure systems. For instance, security concerns are captured in the form of desired properties of the system model, analysis activities verify them to identify appropriate solutions (e.g., policies, mechanisms, ...) to improve the system design and treat any identified property violations.

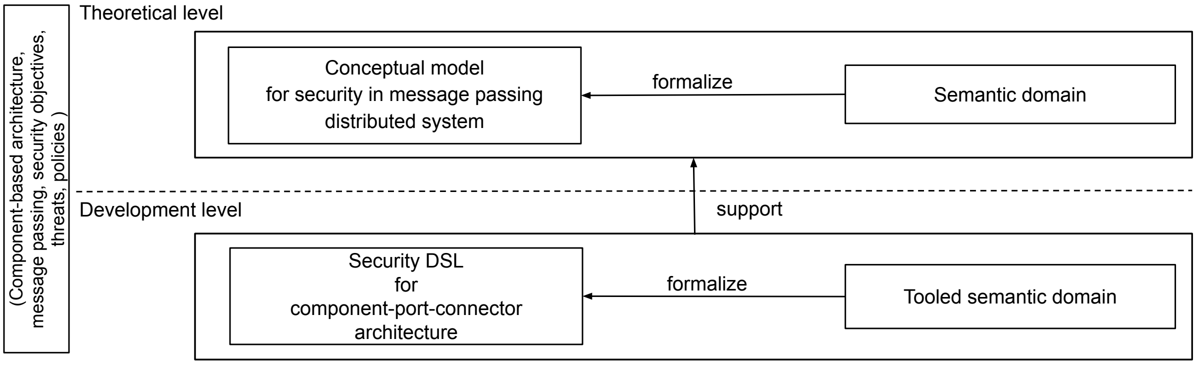

(Component-based architecture, message passing, security objectives, threats, policies)
Theoretical level
Conceptual model for security in message passing distributed system
formalize
Semantic domain
support
Development level
Security DSL for component-port-connector architecture
formalize
Tooled semantic domain

Figure 3.1: Conceptual vision

Figure 3.2 is an application of the conceptual vision introduced in Figure 1.1 to highlight the work achieved in this dissertation.

- Theoretical level: Creation of a conceptual model of a component-based software architecture, message passing communication system, security concerns and the semantics. This conceptual model provides a common understanding of all concepts to implement in the development level, while the semantic domain ensures a precise description of the addressed security problems from a negative perspective (threats) and/or a positive perspective (objectives) and solution approaches (policies). Architecture and security concerns are described as properties of the modeled system. For our purpose, we used textual description, UML class diagram to capture the structural concepts and first-order logic and modal logic for the semantics.

- Development level: Creation of a tooled methodology for the system architect to support the design and analysis activities during the processes of building secure software architecture from the conceptual modeling and the semantic domain. For our purpose, we used EMFT and its features to build the DSLs and Alloy and Coq for the analysis.

3.3 Methodology for the creation of a design and analysis framework

In this section, we proposed an implementation of the conceptual vision introduced in Figure 1.1 which are then applied for the architecture and security concerns in the next chapters. Then, we describe a set of guidelines for the integration of the methodology in existing SDLC methodologies.

In general, the proposed methodology supports the design and analysis of software architecture and concerns. The goal is to capture and formalize various concerns of the system architecture and provide them as verified reusable models. It enables the identification of requirement issues (i.e., concerns violation) of a system architecture by means of verification of desired properties on the architecture model. It also enables the suggestion of solutions to improve the system design and to treat the identified issues. As depicted in Figure 3.2, the methodology is composed of several steps and activities.

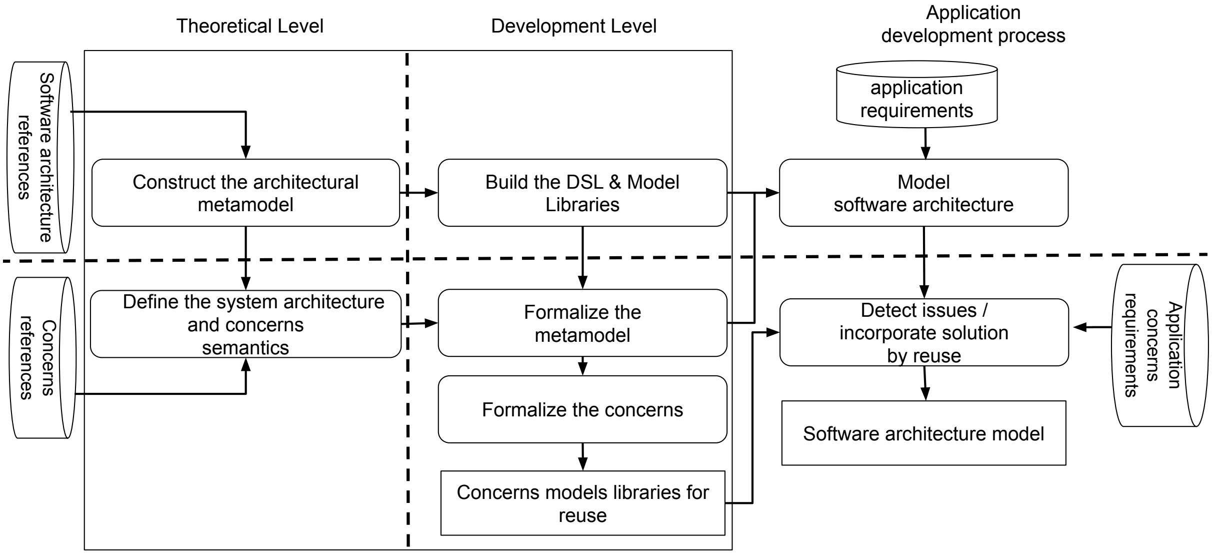

Theoretical Level
Development Level
Application development process
Software architecture references
Concerns references
Construct the architectural metamodel
Define the system architecture and concerns semantics
Build the DSL & Model Libraries
Formalize the metamodel
Formalize the concerns
Concerns models libraries for reuse
application requirements
Model software architecture
Detect issues / incorporate solution by reuse
Software architecture model
Application concerns requirements

Figure 3.2: Methodology for the creation of a design and analysis framework

Conceptual modeling and semantics. We begin by developing an architecture meta-model as high-level concepts of the software architecture. We need to capture the structural and behavioral aspects of the system architecture. In addition, we develop a property view of the meta-model to describe the concerns. The property view describes the concepts required for building reusable property model libraries. The entry point of the approach is an architectural meta-model, manually created with high-level concepts of the software architecture. Then, we define the semantics of the system architecture and concerns according to the system computing model.

Development. We create a DSL from the conceptual model to describe software architecture and the identified concerns in the form of properties on the modeled system. A set of property model libraries for reuse is also provided, capturing the classification of properties. The semantics of the architecture and the concerns are then specified using a suitable formal tooled language. A set of reusable formal model libraries for the identification of the desired and/ or for the violation of properties are also provided. Furthermore, we propose solutions in the form of reusable formal model libraries to mitigate these issues.

3.4 Supporting the approach within SDLC

As shown in the right part of Figure 3.2, the overall approach can easily fit well into various systems development life cycles (SDLCs) as a supplement to the requirements specification and architecture design phases. We build a concrete architecture by instantiating the architecture meta-model and its formal description into a concrete model for an application-specific system. Then, we use the concerns libraries to identify potential issues in the formal system model by checking that it satisfies the desired properties. If the model does not satisfy the properties, we use our developed libraries to suggest new solutions to treat the identified issues.

For simplicity, as shown in Figure 3.3, we illustrate how the approach can be integrated within the Royce iterative waterfall SDLC [118]. The activities defined in Section 3.3 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. This activity concerns the modeling framework development process and should only be done if a new concern requirement is needed to be added in the framework, i.e., if a needed specific concern requirement is not already defined in the framework libraries. (2) The architecture design is extended with the “Model Software Architecture” and “Incorporate / Analysis

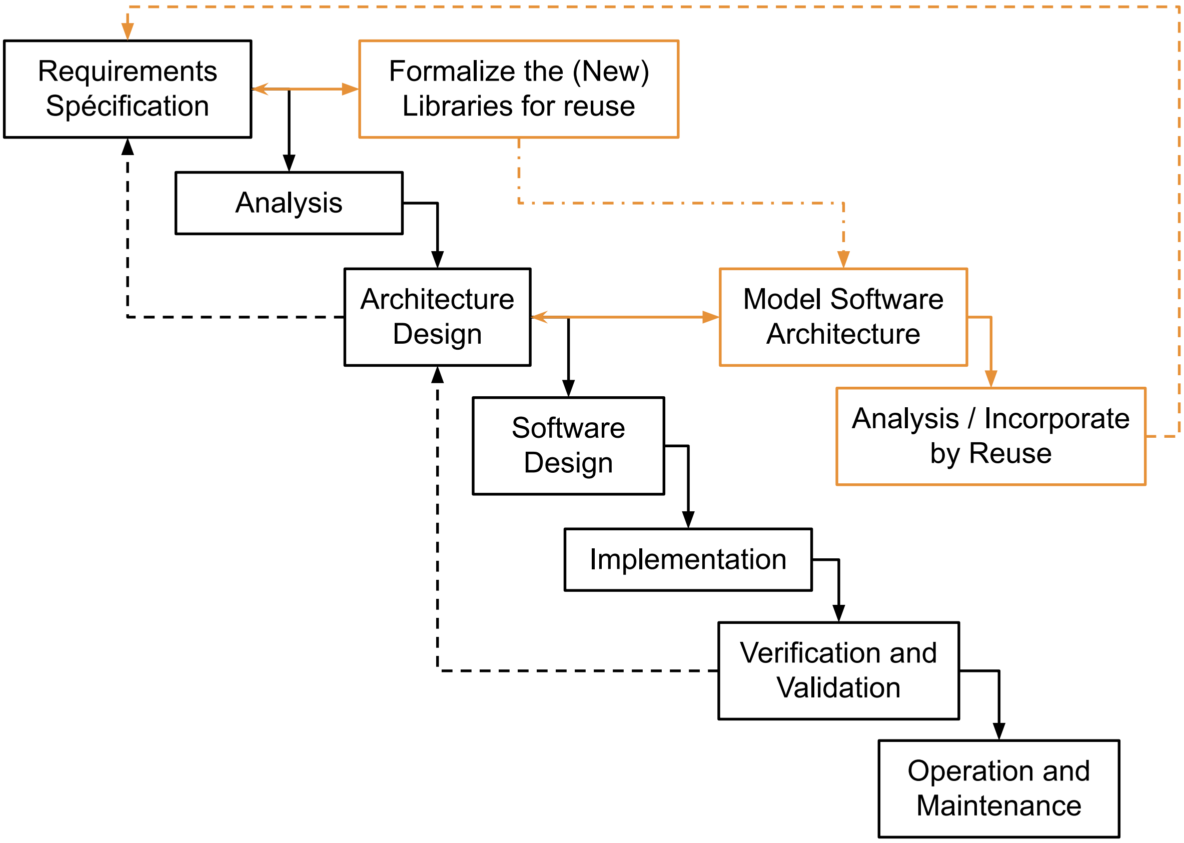

Requirements Spécification
Formalize the (New) Libraries for reuse
Analysis
Architecture Design
Model Software Architecture
Software Design
Analysis / Incorporate by Reuse
Implementation
Verification and Validation
Operation and Maintenance

Figure 3.3: The proposed approach within the Royce iterative waterfall SDLC

by Reuse" activities. The goal of these activities is to ensure that the software architecture model satisfies the desired properties for the designed system using the previously define libraries. (3) Finally, if we determine that the system design does not satisfy the desired properties, we take action to revisit the requirements 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 and mitigate the issues. As a result, the proposed approach enables the generation of formal artifacts early enough in the lifecycle to apply useful analysis within the design loop.

3.5 Conclusion

In this chapter, we discuss the implementation of the conceptual vision and these related notions in the context of software architecture and its concerns.

The proposed methodology allows the creation of a design and analysis framework to assist software architect in the design process. The methodology support the development of reusable formal model libraries for the specification and analysis of concerns by a concern expert; and the architecture design conforming to requirements related to these concerns by an architect by reuse. Therefore, the approach allows to capture expert knowledge on specific concerns in the form of the reusable model libraries. These libraries are then used to aid the designer to design concrete system architectures through detection of issues and the selection of solutions.

In the next chapters, we apply this methodology to create :

1. Component and connector based software architecture design and analysis framework in Chapter 4.

2. Security threats design and analysis framework in Chapter 5

3. Security objectives design and analysis framework in Chapter 6.

Chapter 4

Software architecture

Contents

4.1 Introduction

As presented in Section 1.1, we aim to be able to verify a set of security concerns to check if the architecture models of distributed system satisfy all the desired properties. As a prerequisite, we propose a reliable and verified architectural and interaction model that will be then used as building blocks to specify the security concerns in the negative view, i.e., threats modeling, in Chapter 5 and in the positive view, i.e., objectives modeling, in Chapter 6. We explore the well-known communication paradigms in the context of component-connector-based software architecture development. Therefore, in this chapter we present an approach to specify the architecture model and reusable communication models in the forms of libraries of connectors. With regard to our contributions, we deal with ***C1***,***C2.1***,***C2.1***, ***C3.1***,***C4.1***,***C5.1*** related to the Research Objective 1 (***RO1***), the software architecture concerns from ***C6***,***C7*** related to the Research Objective 2 (***RO2***) and the architecture concerns from ***C8***,***C9*** related to the Research Objective 3 (***RO3***).

We will focus in the following sections on message passing system (MPS) communication paradigm. The methodology developed in this work to build rigorous secure software architecture uses MPS as the communication model. However, the approach can be applied to other communication paradigms. In Appendix A, we provide some illustrations to demonstrate the deployment of the approach on other communication models such as *remote procedure call (RPC)* and *distributed shared memory (DSM).*

The remainder of this chapter is organized as follows. Section 4.2 compares our work with related work. Then, Section 4.3 describes the methodology to build a design and analysis framework for component-based software architecture development. Section 4.4 presents how our approach could be integrated to existing system development life cycle. Section 4.5 presents our component based architectural meta-model. Section 4.6 describes the communication style semantics through logical specification using first order and modal logic, and finite state machine models. Then, Section 4.7 presents and interpretation of the software architecture meta-model and the proposed logical specifications in Alloy. Section 4.8, proposes a tool we developed to support the proposed approach. Finally, Section 4.9 concludes and outlines directions for future work.

4.2 Related work

Recently, there has been a shift in terms of software architecture design [111] towards combining multiple software engineering paradigms, namely, Component-Based Development [25], Model-Driven Engineering [123] and formal methods [112]. In the spirit of using multi-paradigms, many description languages and formalisms for modeling complex distributed systems have been proposed in the literature. A significant proportion of these works have aimed at capturing the communication, concurrency, and some non-functional properties of the components that comprise a given system.

A general methodology for the specification of component-based architecture connection was presented in [6]. It uses architectural connectors with formal semantics. The approach allows to verify architectural compatibility as the type checking in programming languages. We can also cite two other approaches for describing connectors. Bures and Plasil modeled four basic component interconnection types message passing, remote procedure calls, streaming, and blackboard to allow for connector variants to reflect distribution, security, fault-tolerance, etc. in software architectures [19], while Shin et al. proposed a software product line approach modeling the variability of secure software connectors and to promote connector reusability [125],

In addition to the above approaches, several formal languages used in engineering software systems are studied in this respect. Examples of these studies include those using process algebras (e.g., CCS [83], CSP [53], ACP [14], and π-calculus [84])), algebraic specification languages (e.g., CKA [54] and C^{2}KA [61]), architectural modeling languages (e.g., CCM [91], AADL [120], MARTE [94], PSCS [96], SysML [92], and the recent OMG initiative UCM [97]), and architectural formal languages (e.g., OCL [93], Wright [6], labeled transition systems [108]).

Further, communication styles used in software architectures often align with the interactions distinguished in different kinds of middleware such as message-oriented middleware (e.g. CORBA Message Service [91], JMS [103], JORAM [22]), remote procedure call-oriented middleware (e.g. CORBA [91], Java Remote Method Invocation (RMI) [104]), and distributed shared memory (e.g., JavaSpaces [102]).

While each of the above-mentioned modeling formalisms, modeling languages, and specific execution infrastructures have already been successful in many application domains, in this manuscript we build a new communication-based architectural formal modeling language using Alloy for the structural and behavioral specification and analysis of distributed systems. Close to our proposed approach, especially on the relationship between structural and behavior aspects, MARTE [94] Generic Component Model (GCM) and Precise Semantics of Composite Structures (PSCS) [96] allowed to specify semantics on component-based architecture models. Our approach mainly differs in that connectors are defined as concepts close to a component dedicated to communication that provide verified behavior properties such as communication style specific properties. In both other approaches, connectors are simple links and all behavior aspects are specified at the Port/Component level. We believe that by allowing the specification and verification connector libraries, our approach, when compared to others, enables good reusability and makes connectors an ideal place to integrate other verified communication mechanisms (e.g., security, safety, dependability, etc.).

Also closely related to our vision is the approach of Khosrav et al. [65] which provides a modeling and analysis of the Reo connectors using Alloy and the approach of Garlan [26] that describes a formal modeling and analysis of software architectures built in terms of the concepts of components, connectors, and events. In addition, recently, Nawaz and Sun presented an approach for formally modeling, analyzing, and verifying Reo connectors using PVS [90].

In our work, we provide support for specifying systems at various levels of abstraction by combining the characteristics of both state-based and trace-based models, offering a flexible and verifiable view of communication where several non-functional requirements could be specified and treated in a fine-grained fashion. 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.

4.3 Methodology for the creation of a design and analysis framework

In this section, we present an application of the methodology proposed in the previous chapter (Section 3.3) to the development of component and connector based software architecture. Particularly, we consider the use of message passing as a communication paradigm in the context of distributed systems.

As depicted in Figure 4.1, the methodology is composed of several phases and activities. It supports the development of verified and reusable model libraries to represent communication solutions for specifying software architectures of distributed systems.

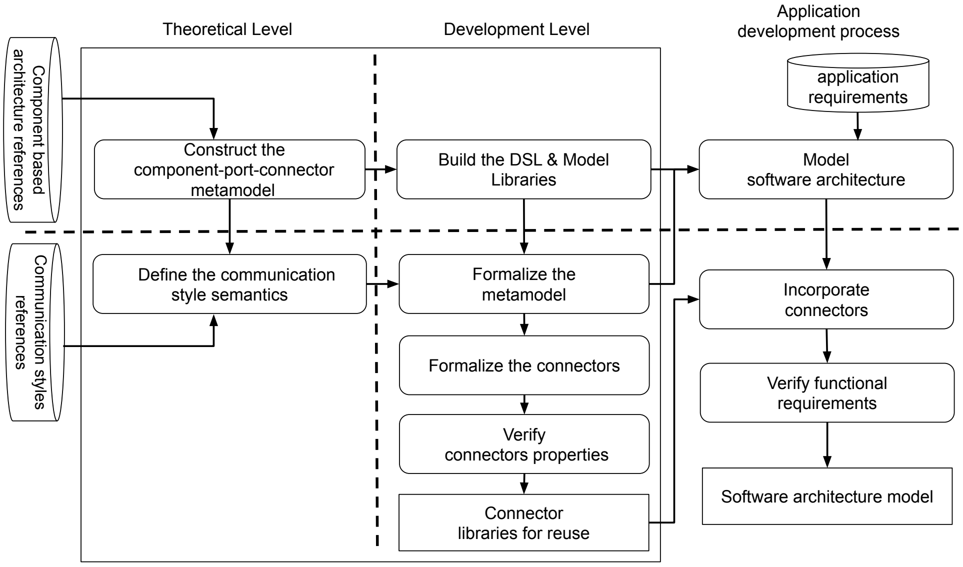

Theoretical Level
Development Level
Application development process
Component based architecture references
Communication styles references
Construct the component-port-connector metamodel
Define the communication style semantics
Build the DSL & Model Libraries
Formalize the metamodel
Formalize the connectors
Verify connectors properties
Connector libraries for reuse
application requirements
Model software architecture
Incorporate connectors
Verify functional requirements
Software architecture model

Figure 4.1: Overview of the proposed approach

Conceptual modeling. We begin by developing a component-port-connector meta-model as high-level concepts of the software architecture. The details of the meta-model are described in Section 4.5. We provide a structural model by describing the structural concepts of the meta-model to ensure that software architectures are well formed. We also model a set of common communication style semantics by rigorously specifying them using finite state machine representations as described in Section 4.6.2. This enables us to define the semantics of the different communication styles so that they can be formalized to verify desirable properties of the communication connectors that will be used to build software architectures for distributed systems.

Development. After constructing the component-port-connector meta-model and interaction semantics, we developed : (1) a DSL to model software architecture and (2) a formal modeling as an interpretation of the meta-model, the communication primitives and their properties using *Alloy* as a tooled formal language. By doing so, we obtain a formal architecture meta-model and a formal specification of the connectors (one for each modeled communication style). Therefore, we are able to formally verify properties for each connector (i.e., interaction) that represent particular connector requirements. The result is a formal architecture meta-model module and set of formal verified connector modules that can be easily reused.

4.4 Supporting the approach within the SDLC

As shown in the right part of Figure 4.1, the overall approach can easily fit well into various systems development life cycles (SDLCs) as a supplement to the requirements specification and architecture design phases. We build a concrete architecture by instantiating the abstract architecture meta-model and abstract verified connectors into a concrete model for an application-specific distributed software system. Then, we use the verified communication primitives offered by the connectors to verify functional requirements of this concrete software architecture. In Section 4.8, we demonstrate how to build a software architecture for the college library web application described in Section 2.8.1 using the proposed approach and the reusable connector libraries, thereby enabling the verification of functional requirements satisfied by the developed software architecture.

For simplicity, as shown Figure 4.2, we illustrate the application of the framework within the Royce iterative waterfall SDLC [118]. 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) Connector Librariries” activities. These activ-

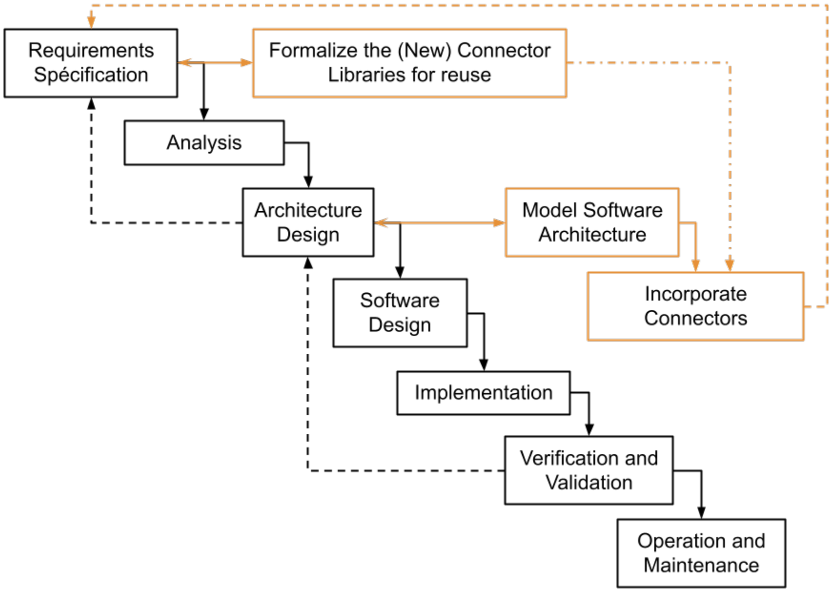

Requirements Spécification
Formalize the (New) Connector Libraries for reuse
Analysis
Architecture Design
Model Software Architecture
Software Design
Incorporate Connectors
Implementation
Verification and Validation
Operation and Maintenance

Figure 4.2: The proposed architecture design approach within the Royce iterative waterfall SDLC

ities 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 "Incorporate Connector" 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 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.

4.5 Software architecture meta-model

In the context of reliable distributed systems, a connection between distributed components should perform a reliable and trusted communication. While this could be done with standard specification of distributed component-based applications, such as those based on CCM [91, 93] and ADL-like vocabulary [120, 6], it would be impossible to configure and control the reliability and trustworthiness of communication connections at design time. This motivates the usage of the connector concept to embed specific interaction semantics and multiple implementations of the semantics within distributed computing systems. 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. Since a connector is responsible for incoming, outgoing, intercepting, and blocking data and messages, it is an ideal place for the integration of security and dependability mechanisms. *In our meta-model, we provide concepts to capture representative communication styles commonly used in distributed systems within component-port-connector architecture model.*

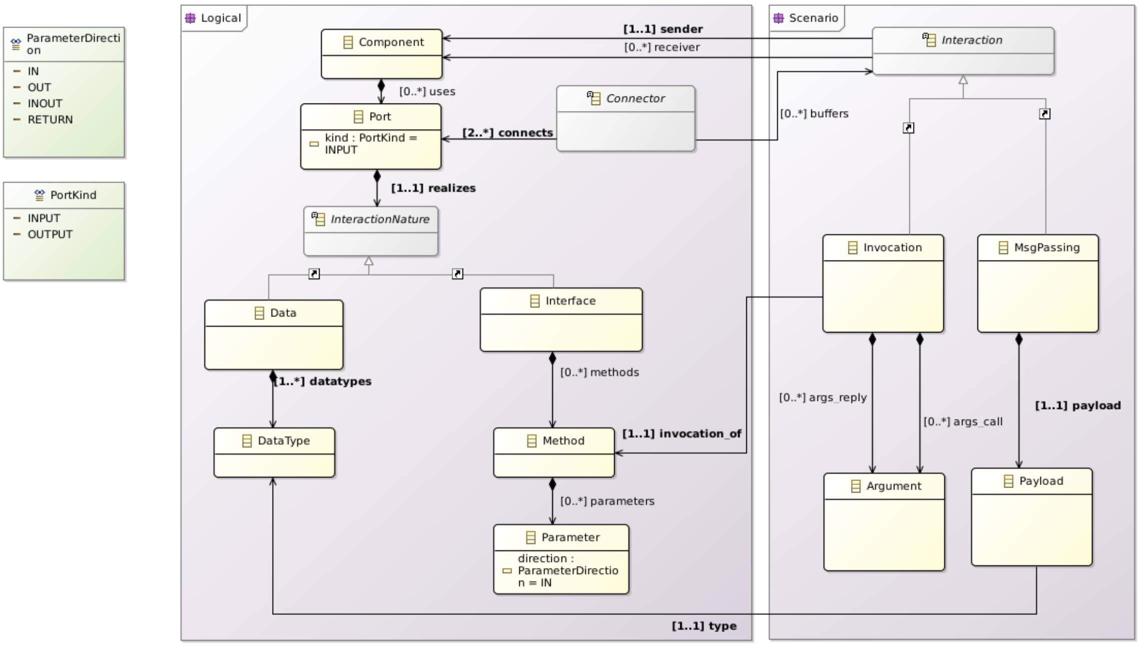


Figure 4.3: Component-port-connector meta-model

We propose to build a modeling framework to define architectural models that are conceptually close to the industrial practice, i.e., containing a UML-like and a UCM-like vocabulary. Figure 4.3 visualizes a meta-model as a class diagram. The meta-model provides concepts for describing software architectures in terms of different views [68], with a focus on:

1. *Logical view* to capture the functional architecture of the system in terms of com-

ponents. This view is concerned with the functionality that the system provides for the end user.

2. *Scenario view* 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 meta-model, as described with UML notations in Figure 4.3.

- *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.

- *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 the following two InteractionNatures:

  - *Data.* A Data defines a data interaction between two Components. Any Component that realizes a Data can be either providing content (data) as a producer or requiring content (data) as a consumer. In the case of a producer, Data is of type OUT. In the case of a consumer, Data is of type IN.

  - *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 (i.e., requires another component to fulfill this contract), the Interface is REQUIRED.

- *Interaction.* An Interaction represents a specific communication behavior between Components (sender and receiver(s)). This is an abstract concept. In our work, we have focused on defining the following two Interactions:

  - *MsgPassing.* A MsgPassing 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). The this interaction type is used for message passing interaction.
  - *Invocation.* An Invocation is the representation of a call and the reply of a 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). The this interaction type is used for remote procedure call and dynamic shared memory interaction.

Example of our meta-model instantiation. In the example of the college library web application described in Section 2.8.1, we can identify the corresponding architectural concepts that will need to be instantiated in order to fulfill the desired requirements:

- *Req_1.* We define two components, namely a *Browser* and a *Website.* The *Browser* needs to use a port that realizes a required interface containing a method *getBook.* The *Website* needs to use a Port that realizes a provided interface containing a method *getBook.* Finally, it needs to use a connector to connect these ports in order to make possible the delivery of *getBook* invocations between the *Browser* and the *Website.*

- *Req_2.* We define two components, namely a *Database* and *Website.* The *Database* needs to use a port that realizes a data producer. In a similar way, the *Website* can use a port that realizes a data consumer. Finally, it needs to use a connector to connect these ports in order to make it possible to pass the messages from the *Database* to the *Website.*

- *Req_3.* We define two components, namely *Terminal* and *Database.* The *Terminal* can use a port that realizes a required interface containing a method *Write*, allowing it to execute write operations. Similarly, the *Database* can use a port that realizes a provided interface with a method *Write*, allowing it to receive write operations. Finally, a connector must connect the two ports of the *Terminal* and the *Database*, allowing it to deliver write invocations between the *Terminal* and the *Database.*

At this level of description, we are able to define: (1) a set of structural elements, mainly components, ports and connectors, (2) required and provided services and (3) dependencies in terms of methods and connections.

4.6 Scenario view

In this section, we present the behavioral semantics specifications of the proposed metamodel for message passing communication. In our component-based software architecture metamodeling framework (see *Scenario* part of Figure 4.3), the communication between two components is done through a channel as a connector connecting two ports belonging to these two components. To understand and apprehend this semantics we used finite state to describe them. Then, used *first-order logic and modal logic* as a formalism that is abstract and technology-independent to specify the desired properties for message passing communication. This provides a more generic and understandable approach with mapping support to different existing property specification languages used in modeling and software development.

4.6.1 Logical specification

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 secure systems, we define the following domain to capture the notion of both legitimate and illegitimate message providers (e.g., *inject*) and message consumers (e.g., *intercept*) of the system messages.

Sets

- $\mathcal{C}$ is the set of components
- $\mathcal{D}$ is the set of message payloads
- $\mathcal{M}$ is the set of messages
- $\mathcal{T}$ is the set of data types

- $\mathcal{I}$ is the set of invocations
- $\mathcal{R}$ is the set of arguments
- $\mathcal{V}$ is the set of variables
- $\mathcal{A}$ is the set of actions
  - send(m, t) denotes that the message $m \in \mathcal{M}$ of type $t \in \mathcal{T}$ is sent into the system
  - receive(m, t) denotes that the message $m \in \mathcal{M}$ of type $t \in \mathcal{T}$ is received from the system
  - call$(i, args_in, args_out)$ denotes that the invocation call $i \in \mathcal{I}$ is sent into the system with the $arg\_ins \in \mathcal{R}$ and the expected $arg\_outs \in \mathcal{R}$
  - executeCall$(m, args_in, args_out)$ denotes that the invocation call $i \in \mathcal{I}$ is received from the system with the $arg\_ins \in \mathcal{R}$ and the expected $arg\_outs \in \mathcal{R}$
  - reply$(m, args_in, args_out)$ denotes that the invocation reply $i \in \mathcal{I}$ is sent into the system with the $arg\_ins \in \mathcal{R}$ and the the concretes $arg\_outs \in \mathcal{R}$
  - executeReply$(m, args_in, args_out)$ denotes that the invocation reply $i \in \mathcal{I}$ is received from the system with the $arg\_ins \in \mathcal{R}$ and the the concretes $arg\_outs \in \mathcal{R}$
  - reply$(var, value)$ denotes that the write instruction for the $v \in \mathcal{V}$ is sent into the system with the $value \in \mathcal{D}$
  - executeWrite$(var, value)$ denotes that the write instruction for the $var \in \mathcal{V}$ is received from the system with the $value \in \mathcal{D}$
  - read(var) denotes that the read instruction for the $v \in \mathcal{V}$ is sent into the system
  - executeRead$(var, value)$ denotes that the read instruction for the $var \in \mathcal{V}$ is received into the system

Modalities

- $\mathbb{H}_c(a)$ is a predicate indicating that action $a \in \mathcal{A}$ $\mathbb{H}$**appens** for component $c \in \mathcal{C}$
- $pred1 < pred2$: is predicate that indicating that all possible sequences of actions that contain a valid predicate $pred2$ also contain a valid predicate $pred1$ that **happens before** $pred2$
- $pred1 \rightsquigarrow pred2$: is predicate that indicating that all possible sequences of actions that contain a valid predicate $pred1$ also contain a valid predicate $pred2$ that **happens after** $pred1$

4.6.2 Communication behavior semantics

As presented in Figure 4.1, we provide the behavioral semantics of message passing communication style in distributed computing systems as abstract communication models. Therefore, in modeling this communication style, each of the two communicating parties (client and server) and the channel (connector connecting two ports) between them are described as a *finite state machine*.

Message passing. In the message passing communication style (MPS), a channel is used for sending a message from a client to a server. The message is simply transmitted without any acknowledgement. The communication channel is modeled as a set of fixed length for messages offering two operations: (a) *push* to add an element in the set and (b) *pull* to remove an element from the set.

The left side of Figure 4.4 shows the states of a client for sending a message. It is shown that if the state is 0 (sent) and a send event occurs when the buffer is not full ($\sim (\#buf = max)$), it changes its state from 0 to 1 for sending [1]. On the other hand, if the buffer is full ($\#buf = max$), it remains at state 0. It also shows that if the state is sending and the message is in the buffer ($mess\_in\_buf$), it changes its states from 1 to 0 for sent.

Similarly, the right side of Figure 4.4 shows the states of a server for receiving a message. It is shown that if the state is 0 (received) and the buffer has a message ($mess\_in\_buf$), it changes its state from 0 to 1 for receiving a message. On the other hand, if the message is not in the buffer, it remains at state 0. It also shows that if the state is receiving and the message is no longer in the buffer, it changes its states from 1 to 0 for received.

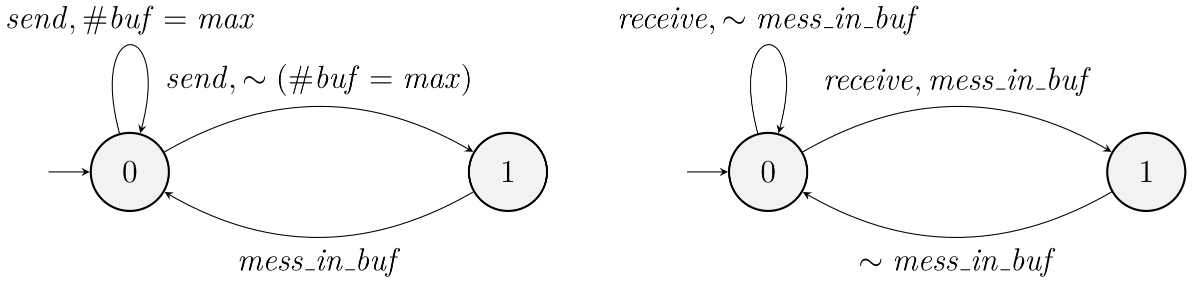


Figure 4.4: States of a client (resp. server) for sending (resp. receiving) messages

Figure 4.5 shows the states of a connector for pulling and pushing a message. It is shown that if the state is 0 for waiting to receive messages from a caller and a message is

[1] $\sim Q$ denotes the negation of the statement Q and $\#A$ denotes the cardinality of the set A.

pushed into the buffer, it changes its state from 0 to 1. If the current state 1 for waiting to receive a message from a caller or retrieving a message from a receiver, it shows that if an event push or pull is executed and the buffer has more than one message but is not full ($max > \#buf > 1$), then it stays in the state 1. Otherwise, if a pull occurs and the buffer only has one message ($\#buf = 1$), it changes its state from 1 to 0. But if a push occurs and the buffer is full minus 1 message ($\#buf = max - 1$), it changes its state from 1 to 2 for retrieving messages from a receiver. Finally, it shows that to change from state 2 to 1 only a pull event is required.

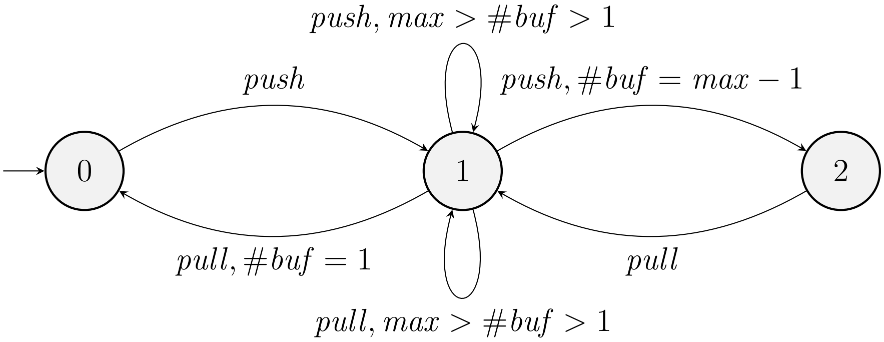


Figure 4.5: States of a MPS connector

Message passing with FIFO ordering. The message passing with first-in-first-out (FIFO) ordering communication style is identical to message passing with a preservation of the order from the perspective of a sender. If a sender sends one message before another, it will be delivered in this order at the receiver. Here, the communication channel is modeled as a queue of fixed length for messages offering two operations: (a) *push* to add an element at the head of the queue and (b) *pop* to remove the element at the tail of the queue.

The left side of Figure 4.6 shows the states of a client for sending a message. It is shown that if the state is sent and a send event occurs when the buffer is not full, it changes its state from 0 (sent) to 1 for sending. On the other hand, if the buffer is full, it remains at state 0. It also shows that if the state is sending and the message is at the head of the buffer ($mess\_head\_buf$), it changes its states from 1 to 0 for sent.

Similarly, the right side of Figure 4.6 shows the states of a server for receiving a message. It is shown that if the state is received and a message is at the tail of the buffer ($mess\_tail\_buf$) it changes its state from 0 (received) to 1 for receiving a message. On the other hand, if the message is not at the tail of the buffer, it remains at state 0. It also shows that if the state is receiving and the message is no longer in the buffer, it changes its states from 1 to 0 for received.

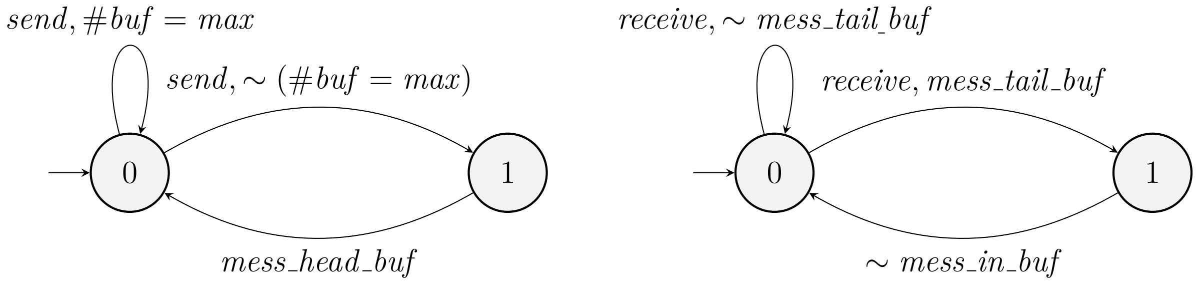


Figure 4.6: States of a client (resp. server) for sending (resp. receiving) messages

Figure 4.7 shows the states of a connector for popping and pushing messages. It is shown that if the state is 0 for waiting to receive a message from a caller and a message is pushed into the buffer, it changes its state from 0 to 1. If its current state is 1 for waiting to receive a message from a caller or retrieving a message from a receiver, it shows that if an event pop or pull happens and the buffer has more than one message but is not full, then it stays in the state 1. Otherwise, if a pop occurs and the buffer only has one message, it changes its state from 1 to 0. But if a push occurs and the buffer is full minus 1 message, it changes its state from 1 to 2 for retrieving messages from a receiver. Finally, it shows that to change from state 2 to 1 only a pop event is required.

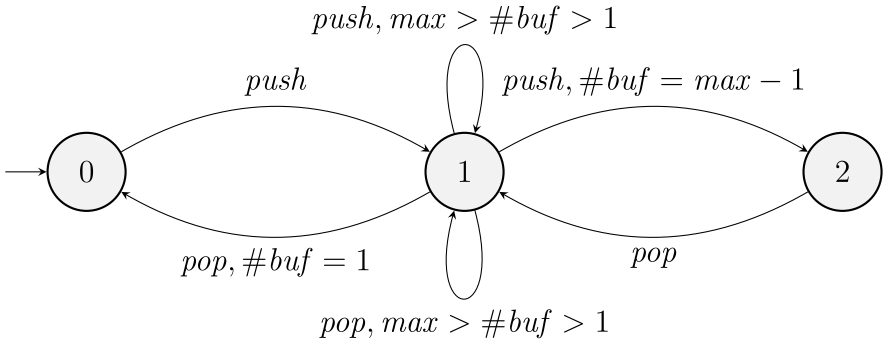


Figure 4.7: States of a MPS FIFO connector

4.6.3 Communication properties specification

Our intent is to illustrate the approach described in Section 4.3 using representative communication style categories extracted from commonly used communication paradigms in distributed systems in the context of a component-port-connector architecture model. Therefore, in following sections, we specify some representative properties for the message passing category.

Once the client $c1$ sends a message to server $c2$ eventually that server receives it.

$$\forall\ c1, c2 \in \mathcal{C}, d \in \mathcal{D}, typ \in \mathcal{T}\ \cdot\ \mathbb{H}_{c_1}(send(d, typ))\ \rightsquigarrow\ \mathbb{H}_{c_2}(receive(d, typ)) \tag{4.1}$$

Once the server $c1$ receives a message, it must already have been sent by a certain client $c2$.

$$\forall\ c1, c2 \in \mathcal{C}, d \in \mathcal{D}, typ \in \mathcal{T}\ \cdot\ \mathbb{H}_{c_2}(send(d, typ))\ <\ \mathbb{H}_{c_1}(receive(d, typ)) \tag{4.2}$$

Messages sent from the client $c1$ to the server $c2$ reach the server $c2$ in the same order as they were sent from $c1$.

$$\begin{gathered}\mathbb{H}_{c_1}(send(d1, typ1))\ <\ \mathbb{H}_{c_1}(send(d2, typ2))\ \wedge\\ \mathbb{H}_{c_2}(receive(d1, typ1))\ <\ \mathbb{H}_{c_1}(receive(d2, typ2))\end{gathered} \tag{4.3}$$

Example of communication properties specification. In the example of the college library web application described in Section 2.8.1, we can identify the corresponding communication semantics that will need to be used in order to fulfill the desired requirements:

- *Req_1.* As this requirement concerns RPC communication, we don't cover it here, but it is available in Appendix A.1.2.2.

- *Req_2.* Both of the message passing communication and the message passing with FIFO ordering styles allow us to express *Req_2.* As identified in Section 4.5, two components (*Website* and *Database*) and a connector are involved in this requirement. To fulfill the requirement, the *Website* must respect the semantics of an MPS client and the *Database* must respect the semantics of an MPS server as described in Figure 4.4. Finally, the connector must respect the semantics of an MPS connector as described in Figure 4.5.

- *Req_3.* As this requirement concerns DSM communication, we don't cover it here, but it is available in Appendix A.1.2.2.

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 finite state machines and, first-order logic and modal logic) which is more understandable by system architects and that can be interpreted in appropriate formal tooled languages.

4.7 Formal specification and analysis in Alloy

In this section, our software architecture meta-model incorporating the concepts of a component-port-connector architecture is formalized using a suitable tooled formal language.

We discuss the formalization of our software architecture meta-model followed by the specification of the connectors and the communication primitives using a suitable tooled formal language : *Alloy.* The semantics of these connectors and communication primitives are the same as those presented in Section 4.6.2. Moreover, we present the definition of a set of properties of these constructs presented in Section 4.6.3.

4.7.1 Formalizing the software architecture meta-model

A software architecture meta-model as described in Section 4.5 is mapped to our Alloy meta-model as follows. The mapping of structural elements is straightforward. Architectural components, ports, connectors, interfaces, methods, and data are mapped to their namesake types in Alloy.

To model the behavior of the system, we consider abstract time in the form of a parameter to express instants of occurrences of actions. An execution of a system is a sequence of steps (instants), where a step is determined by two successive time points. We used the Ordering module provided within Alloy to express a time in a discrete sense, referred to as Tick, where time is explicitly modeled as a set of discrete, ordered Tick instances. Therefore, associations (such as the set of ports connected by one connector) can be made by adding a relationship with the Tick set (i.e., the connects relationship that relates connector to port is a relationship from connector over port to Tick).

A component is connected to a connector through a number of ports. The three basic concepts in the model are components, ports, and connectors that are represented as a set of Alloy signatures as depicted in Listing 4.1. With regard to the scenario view, we defined two additional concepts: *MsgPassing* and *Invocation.* Each of them is produced by the client and consumed by the server.

We encode the set of actions described in Section 4.6.1 considering the notion of execution steps in the context of an asynchronous message-passing system as follows:

- $\mathcal{T}$ is the sequence of execution steps
- Each modality mod$(\mathit{param_1}, \ldots, \mathit{param_n})$ is transformed to a predicate mod$(\mathit{param_1}, \ldots, \mathit{param_n}, t)$ denoting its relation to the execution step $t \in \mathcal{T}$. For example, $\mathbb{E}_c(\text{inject}(m))$ is transformed to E_inject(c, m, t) indicating that a component $c \in \mathcal{C}$ is able to inject a message $m \in \mathcal{M}$ into the system at the step $t \in \mathcal{T}$

```
sig Port {
  realizes: InteractionNature
  kind: PortKind
}
sig Component {
  uses: set Port
}
abstract sig Connector {
  connects: set Port -> Tick
}{
  all disj c1,c2:Component,t:Tick {
    c1.uses + c2.uses in connects.t implies
      some n1,n2:Node {
        c1 in n1.hosts.t
        c2 in n2.hosts.t
        n1 = n2 or some l:Link | n1+n2 in l.connects.t
      }
    }
}
abstract sig Channel extends Connector {
  disj portI,portO: one Port
}{
  all t:Tick | connects.t = portI + portO
}
abstract sig CommunicationArtifact {
  client: one Component,
  server: one Component
}{
  client != server
}
sig MsgPassing extends CommunicationArtifact {
  msgData: one Message
  msgType: one DataType
}
sig Invocation extends CommunicationArtifact {
  invocation_of: one Method,
  arguments_call: set Argument,
  arguments_reply: set Argument -> Tick
}
abstract sig InteractionNature {}
sig Data extends InteractionNature { datatypes: set DataType and kind:
    one DataDirection }
sig DataType {}
sig Interface extends InteractionNature { methods: set Method and kind:
    one InterfaceKind }
sig Method { parameters : set Parameter }
sig Parameter { direction : ParameterDirection }
enum PortKind { INPUT, OUTPUT  }
enum DataDirection { DATA_IN, DATA_OUT}
enum InterfaceKind { PROVIDED, REQUIRED }
enum ParameterDirection{ IN, OUT, INOUT, RETURN }
```

Listing 4.1: Software architecture meta-model in Alloy

Additionally of the properties presented in Section 4.6.3 we add some addition verification of the good integration of the connector within the architecture. Among the set of possible and specified characteristics of the structural architecture, a subset of them are encoded in terms of properties as predicates and the results of their verification are stated below. The first of these properties asserts that components cannot interact without the presence of a connector between them. This constrains the way in which component connections can be established in a system architecture and further prescribes the structural elements that are required to enable communication.

(a) "For two components *c*1 and *c*2 to interact, a connector *con* must be present between them"

```
pred  components_can_interact[ca:CommunicationArtifact, c1,c2:
    Component] {
  some con:Connector, t:Tick, p1:c1.uses, p2:c2.uses |
    c1 = ca.client and c2 = ca.server => p1 + p2 in con.connects.t
}
```

The next two properties assert the consistency of component type operations for each *Interaction* by constraining the structural properties of the components involved in such interactions. We again encode these constraints as predicates in Alloy.

(b) “For two components $c1$ and $c2$ to interact using message passing, each must uses a port that realizes the correct *Data* offering the same *Datatype dt*.”

```
pred  msgpassing_type_check[c1,c2:Component, dt:DataType] {
  all m:MsgPassing |
    m.msgType = dt and c1 = m.client and c2 = m.server =>
    some p1:c1.uses, p2:c2.uses {
      p1.realizes.datatypes = dt
      p2.realizes.datatypes = dt
      p1.realizes.kind = DATA_OUT
      p2.realizes.kind = DATA_IN
    }
}
```

The Alloy Analyzer shows that properties (a) and (b) hold.

4.7.2 Formalizing and verifying connectors and their properties

The MPS connector is defined as a buffer of messages associated with two operations *push* and *pull* to support the high-level communication primitives. Listing 4.2 depicts an excerpt of the formalization of the MPS connector. For instance, once a send is executed by the sender component, *MsgPassing* is buffered in a connector (line 10). When it is received by the receiving component, it is removed from the connector (line 14). Then, to specify the behavior of the MPS connector, we define a fact *traces* (lines 17 to 20) to constrain the acceptable state transitions of the message passing connector to form a valid executable trace. This enables the formalization of the behavioral semantics described in Section 4.6.2.

```
 1  sig ConnectorMPS extends Channel {
 2    buffer : set MsgPassing -> Tick,
 3    capacity: Int
 4  }
 5  pred MPS_init [t: Tick] {
 6    all c:ConnectorMPS | # c.buffer.t = 0
 7  }
 8  pred MPS_push [t, t': Tick, c:ConnectorMPS, mp:MsgPassing] {
 9    #c.buffer.t < c.capacity
10    c.buffer.t' = c.buffer.t + mp
11  }
12  pred MPS_pull [t, t': Tick, c: ConnectorMPS, mp: MsgPassing] {
13    mp in c.buffer.t
14    c.buffer.t' = c.buffer.t - mp
}
fact traces {
  MPS_init[TO/first]
  all t:Tick - TO/last | let t’ = TO/next[t] |
  some c:ConnectorMPS, mp:MsgPassing | MPS_push [t, t’, c, mp]
  iff not MPS_pull [t, t’, c, mp]
}
```

Listing 4.2: Message passing connector

The MPS connector with FIFO ordering, like the previously discussed MPS connector, is also defined as a buffer of messages. Listing 4.3 depicts an excerpt of the formalization of the MPS connector with FIFO ordering. In this case, the buffer is modeled as queue to establish the FIFO behavior of the connector (line 5). Similar to the MPS connector, the high-level communication primitives are supported by *push* and *pop* operations. However, to ensure the FIFO ordering, the Alloy specification employs the enqueue and dequeue operations when defining the *push* and *pop* operations. This enables the formalization of the behavioral semantics described in Section 4.6.2 which constrains the valid executable traces resulting from MPS connector with FIFO ordering to those that ensure that messages are removed from the buffer in the order in which they are received. This is defined as a fact called *traces* (lines 18 to 21).

```
 1 sig QMessage extends QElem {
 2   message: one MsgPassing
 3 }
 4 sig ConnectorMPSFIFO extends Channel {
 5   buffer : one Queue,
 6 }
 7 pred MPSFIFO_init [t: Tick] {
 8   all c:ConnectorMPSFIFO | QEmpty[t, c.buffer]
 9 }
10 pred MPSFIFO_push [t, t’: Tick, c:ConnectorMPSFIFO, mp:MsgPassing] {
11   one qm:QMessage | qm.message = mp and QEnq[t, t’, c.buffer, qm]
12 }
13 pred MPSFIFO_pop [t, t’: Tick, c: ConnectorMPSFIFO, mp: MsgPassing] {
14   QLast[t, c.buffer].message = mp
15   QDeq[t, t’, c.buffer]
16 }
17 fact traces {
18   MPSFIFO_init[TO/first]
19   all t:Tick - TO/last | let t’ = t.next |
20   some mp:MsgPassing, c:ConnectorMPSFIFO | MPSFIFO_push [t, t’, c, mp]
```

```
  iff not MPSFIFO_pop [t, t’, c, mp]
}
```

Listing 4.3: Message passing with FIFO ordering connector

Communication in the message passing communication style is performed using the *send*() and *receive*() primitives (see Listing 4.4). The *send*() primitive (line 7) requires the name of the receiver component, the transmitted data, and the expected data types as parameters, while the *receive*() primitive (line 18) requires the name of the anticipated sender component and should provide storage variables for the message data and the expected data types.

In spite of blocking primitives that are often chosen, for the sake of easier realization, here we consider the semantics of non-blocking primitives to capture the more general asynchronous communication paradigm. The non-blocking *send*(*receiver*, *data*) returns control to the sender immediately and the message transmission process is then executed concurrently with the sender process. The sender executes a *send*(*receiver*, *data*) which results in the communication system constructing a message and sending it to the receiver through the corresponding connector. The receiver executes a *receive*(*sender*, *data*) which causes the receiver to be blocked, awaiting a message from the sender. When the message is received, the communication system removes the message from the corresponding connector, extracts the data from the message and delivers it to the receiver. As a prerequisite, we added the *check_type_interaction_data* predicate (lines 2 to 5) to ensure that the message’s types are supported at both the sending and receiving components. Without data type checking, the support of the message type is only verified at execution time.

```
1  pred check_type_interaction_data[mp:MsgPassing]{
2    one di:Data, p:mp.client.uses | di in p.realizes and di.kind =
         DATA_OUT and
3    mp.msgType in di.DataType
4    one di:Data, p:mp.server.uses | di in p.realizes and di.kind = DATA_IN
          and
5    mp.msgType in di.DataType
6  }
7  pred Component.H_send[receiver:Component, d: Message, typ:DataType, t:
       Tick] {
8    some mp: MsgPassing {
9      mp.client = this
10     mp.server = receiver
11     mp.msgData = d
12     mp.msgData.msgType= typ
13     check_type_interaction_data[mp]
```

```
    one t':t.next | let c = { c:ConnectorMPS | c.portO in mp.client.uses and
    c.portI in mp.server.uses } | MPS_push[t,t',c,mp]
  }
}
pred Component.H_receive[sender:Component, d: Message, typ:DataType, t:Tick] {
  some mp: MsgPassing {
    mp.client = sender
    mp.server = this
    mp.msgData = d
    mp.msgData.msgType= typ
    check_type_interaction_data[mp]
    one t':t.next | let c = { c:ConnectorMPS | c.portO in mp.client.uses and
    c.portI in mp.server.uses } | MPS_pull[t,t',c,mp]
  }
}
```

Listing 4.4: Message passing communication

Among the set of possible and specified characteristics of the behaviors of the message passing communication styles presented Section 4.6.3, a subset of them are encoded in terms of properties as predicates and assertions and the results of their verification are stated below.

(a) "Once the client $c1$ sends a message to server $s1$, eventually that server receives it." (Equation (4.1))

```
pred send_is_eventually_received[c1,c2:Component,d:Message,typ:DataType] {
  one t:Tick | one t':t.nexts |
    c2.H_send[c1,d,typ,t'] => c1.H_receive[c2,d,typ,t]
}
```

(b) "Once the server $s1$ receives a message, it must already have been sent by a certain client $c1$." (Equation (4.2))

```
assert receive_must_be_sent {
  one t:Tick | one t':t.nexts | some c1,c2:Component | some d:Message |  some typ:DataType |
    c2.H_receive[c1,d,typ,t'] => c1.H_send[c2,d,typ,t]
}
```

(c) “Messages sent from the client $c1$ to the server $s1$ reach the server $s1$ in the same order as they were sent from $c1$.” (Equation (4.3))

```
assert is_FIFO {
  all disj c1,c2:Component | all disj d1,d2: Message | some typ1,
      typ2:
    DataType | all ts1:Tick | let ts2 = ts1.nexts | all tr1:Tick |
        all tr2:Tick |
      (c1.H_send[c2,d1,typ1,ts1] and c1.H_send[c2,d2,typ2,ts2]
        and c2.H_receive[c1,d1,typ1,tr1]
        and c2.H_receive[c1,d2,typ2,tr2])
        => tr2 in tr1.nexts
}
```

The Alloy Analyzer shows that properties (a) and (b) hold for both types of message passing connector (simple and FIFO). It also shows that property (c) does not hold for a simple message passing connector. Since the property does not hold, Alloy produces a counterexample, which shows the main reason why the specified property does not hold. However, the Alloy Analyzer shows that this property holds for a message passing connector with FIFO ordering.

Example of building a concrete architecture.

- *Model the software architecture.* The software architecture model is defined as an instance of the proposed meta-model with respect to the functional requirements as identified in Section 4.5. Listing 4.5 depicts the Alloy specification of the architecture of the college library web application example described in Figure 2.8. For instance, the *Browser* can be seen as an instantiation of the *Component* type. We proceed by defining the component types (lines 1 to 4), ports (lines 6 to 11), and interfaces (lines 13 to 18) as simple extensions to the concepts of our software architecture meta-model.

```
1 one sig Browser extends Component {}{ uses = PortInterfaceBrowser }
2 one sig Website extends Component {}{ uses = PortInterfaceWebsite +
      PortDataWebsite }
3 one sig Database extends CentralMemoryManager {}{ uses =
     PortDataDatabase + PortSMDatabase }
4 one sig Terminal extends Component{}{ uses = PortSMTerminal }
5
6 one sig PortInterfaceBrowser extends Port {}{ realizes =
     InterfaceBrowser }
7 one sig PortInterfaceWebsite extends Port {}{ realizes =
     InterfaceWebsite }
```

```
one sig PortDataWebsite extends Port {}{ realizes = DataWebsite }
one sig PortDataDatabase extends Port {}{ realizes = DataDatabase }
one sig PortSMDatabase extends Port {}{ realizes = InterfaceSMDataDatabase }
one sig PortSMTerminal extends Port {}{ realizes = InterfaceSMAdminstrator }

one sig InterfaceBrowser extends Interface {}{kind = REQUIRED and getBook in methods }
one sig InterfaceWebsite extends Interface {}{kind = PROVIDED and getBook in methods }
one sig getBook extends Method {}{ idBook in parameters }
one sig idBook extends Parameter {}{ direction = IN }
one sig InterfaceSMAdminstrator extends Interface {}{ kind = REQUIRED and Write in methods }
one sig InterfaceSMDataDatabase extends Interface {}{ kind = PROVIDED and Write + Read in methods }

one sig DataWebsite extends Data {}{ kind = DATA_IN }
one sig DataDatabase extends Data {}{ kind = DATA_OUT }
one sig DatabaseContent extends DataType {}
```

Listing 4.5: Building a concrete software architecture of the college library web application example in Alloy

- *Incorporate connectors.* At this step, connectors are integrated by reusing the developed and verified connector libraries. In the same way as the meta-model, connectors are instantiated in the concrete architecture by simply defining connector types as extensions of one of the defined connectors. Listing 4.6 depicts the Alloy specification of connectors for the college library web application example described in Section 2.8. The corresponding connector communication style is chosen as identified in Section 4.6.3.

```
one sig BrowerWebsiteConnector extends ConnectorRPC {}{
  portO = PortInterfaceBrowser
  portI = PortInterfaceWebsite
}
one sig DatabaseWebsiteConnector extends ConnectorMPS {}{
  portO = PortDataDatabase
  portI = PortDataWebsite
}

```

```
one sig TerminalDatabaseConnectorSM extends ConnectorSM {}{
  portO = PortSMTerminal
  portI = PortSMDatabase
}
```

Listing 4.6: Instantiate Connectors of a web application example in Alloy

- *Verify functional requirements*

  - *Req_1.* As this requirement concerns RPC communication, we don't cover it here, but it is available in Appendix A.2.3.

  - *Req_2.* We use some of the previously specified and verified structural and communication behavior properties for message passing to specify and verify the application specific functional requirement *Req_2* on the concrete architecture. Listing 4.7 depicts the Alloy specification of this functional requirement as a predicate combining the verified properties *msgpassing_type_check* (line 3) and *send_is_eventually_received* (line 4). These properties are applied to the specific concepts and behaviors identified for this requirement as discussed in Section 4.5 and 4.6.3. The Alloy Analyzer shows that *Req_2* holds.

```
1 pred Req_1 {
2   some w:Website, d:Database, m:Message, typ:DataType {
3     msgpassing_type_check[d,w,typ]
4     send_is_eventually_received[d,w,m,typ]
5   }
6 }
```

Listing 4.7: Using previously verified MPS properties to specify functional requirement of a web application example in Alloy

  - *Req_3.* As this requirement concerns DSM communication, we don't cover it here, but it is available in Appendix A.2.3.

At this level of specification, we are able to: (1) formally represent an interpretation of the software architecture model and properties in Alloy to support property specification and analysis, (2) formalize the behavioral semantics of the connectors described in Section 4.6.2, specifically by encoding the valid executable traces, (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.

4.8 Tool Support

We have implemented a prototype to support the approach as an Eclipse plug-in. Our starting point is the software architecture metamodel presented in Section 4.5 as the metamodel for a software architecture DSL. 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.

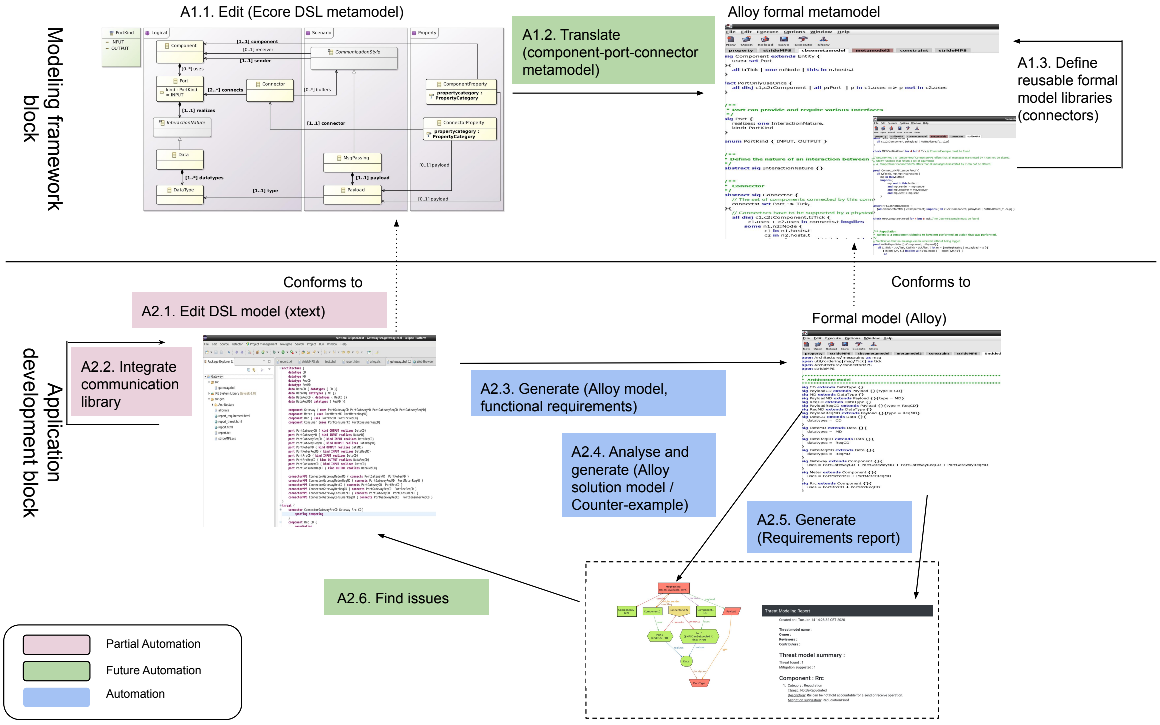

Modeling framework block
A1.1. Edit (Ecore DSL metamodel)
A1.2. Translate (component-port-connector metamodel)
Alloy formal metamodel
A1.3. Define reusable formal model libraries (connectors)
Conforms to
Conforms to
A2.1. Edit DSL model (xtext)
Formal model (Alloy)
Application development block
A2.2. Integrate communication library
A2.3. Generate (Alloy model, functional requirements)
A2.4. Analyse and generate (Alloy solution model / Counter-example)
A2.5. Generate (Requirements report)
A2.6. Find issues
Threat Modeling Report
🟥 Partial Automation
🟩 Future Automation
🟦 Automation

Figure 4.8: Tool support architecture and artifacts of the approach

The architecture of the tool, as shown in Fig. 4.8, is composed of two main blocks:

(1) Modeling framework block and (2) Application development block. Each block is composed of a set of modules to support the corresponding activities (the numbers in parentheses correspond to the activity numbers in Fig. 4.8).

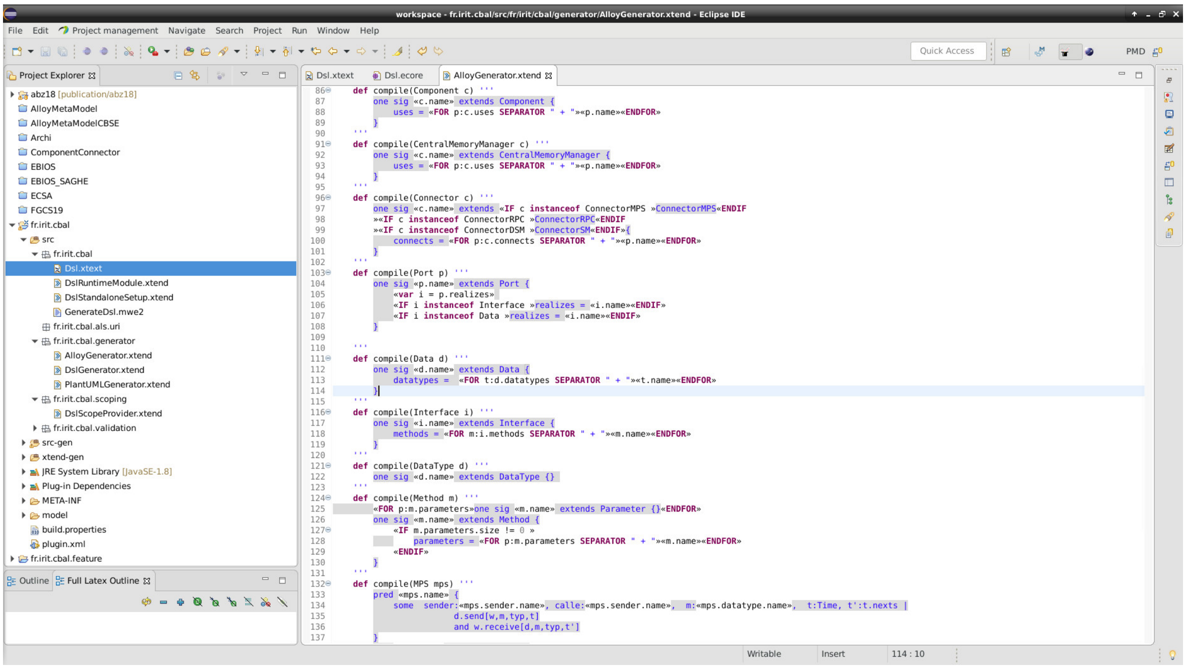
4.8.1 Modeling framework block

The first step (*A1.1*) for the implementation of the tool is the definition of the DSL metamodel using EMF and Xtext. Furthermore, the metamodel can be enriched with logical constraints to avoid some undesired structures, in a similar way to the constraints defined in Section 4.7.1. The second step (*A1.2*) is the formal definition (in Alloy) of the static semantics using the Alloy tool, based upon the DSL metamodel according to the procedure discussed in Section 4.7.1. We will name this definition the formal metamodel. Then in (*A1.3*) reusable connector libraries, their corresponding communication primitives and a set of properties are also defined as Alloy models according to the procedure discussed in Section 4.7.2.

4.8.2 Application development block

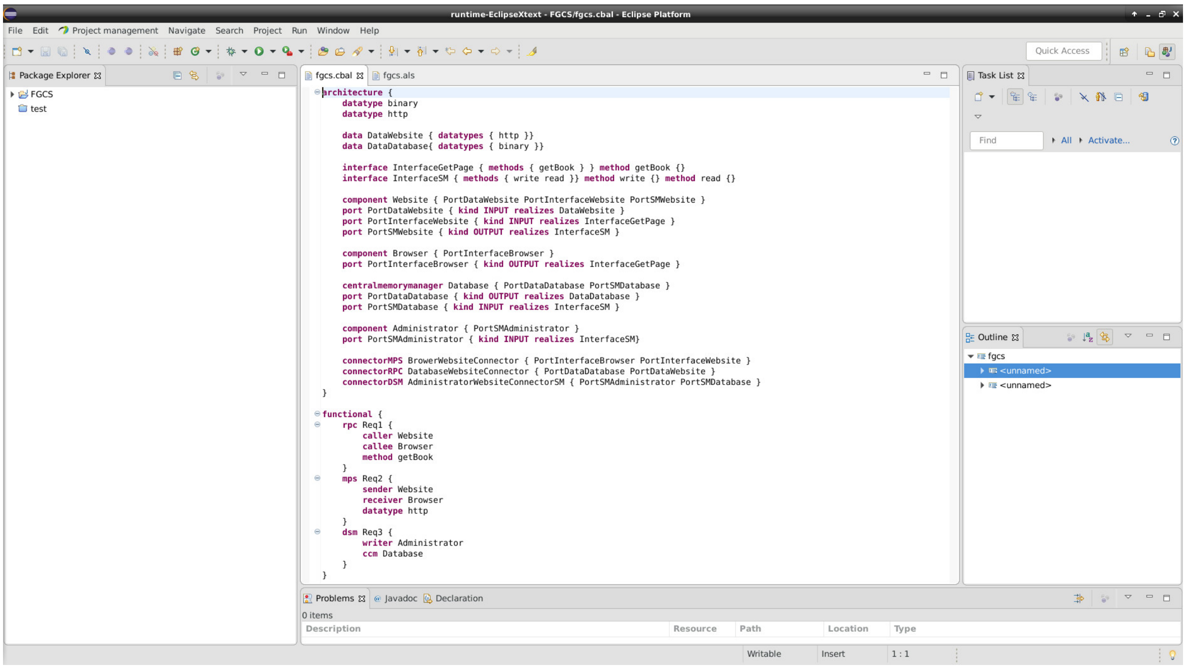
The third step (*A2.1*) is the development of a textual editor to allow the user to model a software architecture (DSL model) conforming to the DSL metamodel. The last step (*A2.3*) is the development of transformations to support the generation of a formal model (in Alloy) from a DSL model, using Xtend (see Figure 4.9). The generation process allows to automatically incorporate the appropriate communication styles in the produced Alloy software architecture model as described in Listing 4.5.

Back to the illustrative example, the plug-in allows a user to edit a DSL model (see Figure 4.10) conforming to the DSL software architecture metamodel as input for generating an Alloy model conforming to the formal metamodel and incorporating the communications. Furthermore, the tool allows the user to define some structural constraints during the specification of the DSL model. The resulting Alloy model can be enhanced with properties in the form of predicates. The Alloy Analyzer is then invoked to verify the desired requirements as properties on the solution model.



```
def compile(Component c) '''
    one sig «c.name» extends Component {
        uses = «FOR p:c.uses SEPARATOR " + "»«p.name»«ENDFOR»
    }
'''
def compile(CentralMemoryManager c) '''
    one sig «c.name» extends CentralMemoryManager {
        uses = «FOR p:c.uses SEPARATOR " + "»«p.name»«ENDFOR»
    }
'''
def compile(Connector c) '''
    one sig «c.name» extends «IF c instanceof ConnectorMPS »ConnectorMPS«ENDIF
    »«IF c instanceof ConnectorRPC »ConnectorRPC«ENDIF
    »«IF c instanceof ConnectorDSM »ConnectorSM«ENDIF»{
        connects = «FOR p:c.connects SEPARATOR " + "»«p.name»«ENDFOR»
    }
'''
def compile(Port p) '''
    one sig «p.name» extends Port {
        «var i = p.realizes»
        «IF i instanceof Interface »realizes = «i.name»«ENDIF»
        «IF i instanceof Data »realizes = «i.name»«ENDIF»
    }

'''
def compile(Data d) '''
    one sig «d.name» extends Data {
        datatypes =  «FOR t:d.datatypes SEPARATOR " + "»«t.name»«ENDFOR»
    }
'''
def compile(Interface i) '''
    one sig «i.name» extends Interface {
        methods = «FOR m:i.methods SEPARATOR " + "»«m.name»«ENDFOR»
    }
'''
def compile(DataType d) '''
    one sig «d.name» extends DataType {}
'''
def compile(Method m) '''
    «FOR p:m.parameters»one sig «m.name» extends Parameter {}«ENDFOR»
    one sig «m.name» extends Method {
        «IF m.parameters.size != 0 »
            parameters = «FOR p:m.parameters SEPARATOR " + "»«m.name»«ENDFOR»
        «ENDIF»
    }
'''
def compile(MPS mps) '''
    pred «mps.name» {
        some  sender:«mps.sender.name», calle:«mps.sender.name»,  m:«mps.datatype.name»,  t:Time, t':t.nexts |
                  d.send[w,m,typ,t]
                  and w.receive[d,m,typ,t']
    }
```

Figure 4.9: Transformations supporting the generation of an Alloy from a DSL model using Xtend



```
architecture {
    datatype binary
    datatype http

    data DataWebsite { datatypes { http }}
    data DataDatabase{ datatypes { binary }}

    interface InterfaceGetPage { methods { getBook } } method getBook {}
    interface InterfaceSM { methods { write read }} method write {} method read {}

    component Website { PortDataWebsite PortInterfaceWebsite PortSMWebsite }
    port PortDataWebsite { kind INPUT realizes DataWebsite }
    port PortInterfaceWebsite { kind INPUT realizes InterfaceGetPage }
    port PortSMWebsite { kind OUTPUT realizes InterfaceSM }

    component Browser { PortInterfaceBrowser }
    port PortInterfaceBrowser { kind OUTPUT realizes InterfaceGetPage }

    centralmemorymanager Database { PortDataDatabase PortSMDatabase }
    port PortDataDatabase { kind OUTPUT realizes DataDatabase }
    port PortSMDatabase { kind INPUT realizes InterfaceSM }

    component Administrator { PortSMAdministrator }
    port PortSMAdministrator { kind INPUT realizes InterfaceSM}

    connectorMPS BrowerWebsiteConnector { PortInterfaceBrowser PortInterfaceWebsite }
    connectorRPC DatabaseWebsiteConnector { PortDataDatabase PortDataWebsite }
    connectorDSM AdministratorWebsiteConnectorSM { PortSMAdministrator PortSMDatabase }
}

functional {
    rpc Req1 {
        caller Website
        callee Browser
        method getBook
    }
    mps Req2 {
        sender Website
        receiver Browser
        datatype http
    }
    dsm Req3 {
        writer Administrator
        ccm Database
    }
}
```

Figure 4.10: Definition of the DSL model and functional requirements for the college library web application

> *Putting all this together and using the developed tooled framework, we are able to: (1) model a concrete component-based software architecture and functional requirements using a domain-specific language, (2) generate an interpretation of the resulted software architecture model and properties in Alloy, (3) verify the satisfaction of a set of functional requirements through reusable specified and verified communication models (connector), and (4) generate new artifacts related to the software architecture model and properties.*

4.9 Conclusion

In this chapter, we described a formal framework to support the rigorous design of software architectures focusing on the communication aspects at the architecture level. The framework is based on a component-port-connector meta-model describing the high-level concepts of distributed software architecture supporting a set of communication styles, namely message passing, remote procedure call, and distributed shared memory. Then, using Alloy, we formally specified and verified a software architecture based on a set of reusable models, namely connectors corresponding to each considered communication styles. In addition, we develop an MDE tool chain to support the proposed approach to assist the architects of model-based rigorous development of computer-based systems, combining modeling and formal techniques.

Our experience in the specification and analysis of various communication styles, including message passing, RPC, and DSM using Alloy are presented. Here, we have verified some most common properties of these three styles of communication and found that the properties hold. Thus from our experience we can say that the connectors and the software architecture using them are verifiable for building reliable distributed systems.

In the next chapters, we study security at the architecture design level, focusing on the message passing paradigm, as an underlying communication system. As we shall see, since a connector is responsible for incoming and outgoing messages, it is an ideal place for the integration of solutions that transparently treat security issues at the architecture level.

Chapter 5

Security threats

Contents

5.1 Introduction

The current practice to formulate security statements from the negative perspective is given through expressing attacker capabilities to e.g., gain access to a protected data from a message observation. Microsoft for example uses a threat taxonomy from the attacker's perspective called STRIDE. Therefore, to define the security architecture of the system, we need an analysis of the possible threats; security solutions can then be introduced to stop or mitigate them.

In this chapter, we propose to use formal methods for the precise specification and analysis of security architecture threats as properties of a modeled system. Starting from an informal description of a threat (i.e., from standards and classifications such as

STRIDE [81]) in the context of component-based software architecture development, a logical specification of these properties is proposed using an abstract system computing model (i.e., a technology-independent specification language such as first-order logic, modal logic) followed by a more concrete specification of the system computing model and the properties (i.e., a suitable language with automated tool support such as Alloy [57]. Finally, a set of security policies are elicited as properties of a modeled system to constrain the operation of the system and to protect against the corresponding threats. With regard to our contributions, we deal with ***C1***,***C2.2***, ***C3.2***,***C4.2***, ***C5.2*** and ***C5.4*** related to the Research Objective 1 (***RO1***), the security threats concerns from ***C6*** and ***C7*** related to the Research Objective 2 (***RO2***) and the security threats concerns from ***C8*** and ***C9*** related to the Research Objective 3 (***RO3***).

To evaluate our approach, we study a set of representative threats based on Microsoft's STRIDE [81] threat classification against the components and the communication links in component-port-connector architecture views [25]. We use MDE abstraction mechanisms to define and handle software architecture model, security threats and policies through a meta-model that unifies those concepts. Moreover, we use MDE transformation mechanisms that can adapt and generate different artifacts and representations. In this work, Eclipse Modeling Framework Technology (EMFT) is used to build the support tools for our approach.

The remainder of the chapter is organized as follows. Section 5.2 discusses related work. Then, Section 5.3 describes the methodology to build a design and analysis framework for security architecture threats. Section 5.4 presents how our approach could be integrated to existing system development life cycle. Section 5.5 presents the property metamodel for threats and describes the formalization of threats using first-order and modal logic following the STRIDE categories. Then, Section 5.6 presents the interpretation of the property metamodel and the logical specification of STRIDE security threats in Alloy. It also includes a set of policies as architectural solutions. Section 5.7 describes the architecture of the tool suite. Finally, Section 5.8 concludes.

5.2 Related work

The formalization of threat models and security properties for the verification and validation of system models has been studied in the recent past. Existing formalization attempts for threats include the work in [55] using VDM++ to specify the core components of threat modeling techniques including STRIDE , DREAD[1], and basic confidentiality, integrity, availability, authentication, authorization, and non-repudiation security mechanisms. The work does not describe the whole approach, but rather presents the core ideas and makes a case for the adoption of formal methods in threat modeling and secure system development. Works from authors in [76] used a logic-based representation for describing abstract security properties which were implemented and verified using Coq [15]. Moreover, [50] used Software Cost Reduction (SCR) tables to specify and analyze security properties codifying system requirements.

From another perspective, modeling threat scenarios as vulnerabilities in a system that makes threats possible is proposed to help understand how attackers may exploit flaws in the architecture made from well-known references (e.g., STRIDE [81], CAPEC [85], and CWE [86]). These references informally describe a set of threat scenarios. Each threat scenario has a signature. This signature specifies the conditions in which a threat can occur. Thus, it defines the threats according to a certain scenario. However, the threat scenarios are described informally and thus applying them manually is error-prone and time consuming. To this end, [7] used OCL [93] to specify signatures of these threats scenarios to automatically detect threats in the architecture model. Four threat scenarios are studied, including denial of service, tampering, injection, and man-in-the-middle. The software architecture is modeled in UML and is called the System Description Model (SDM). The SDM is mapped to the Security Specification Model (SSM) that represents security mechanisms. Another approach to elaborate security threat identification is proposed by extending use cases to misuse cases [100] in the context of security-oriented requirements engineering. The idea behind this approach is to describe functions the system should not allow, eliciting security requirements, and the following constraints on assets.

Other approaches are proposed in specific development contexts. The work in [124] describing a framework called FATHoM (FormAlizing THreat Models) which is a state-based relational model with a logic-based specification of security properties. The goal of FATHoM is to enable the detection of inconsistencies in threat models, especially in the domain of virtualized systems. A semi-automated approach for analyzing a security runtime architecture for security and conformance to an object-oriented implementation, called SECORIA (SEcurity Conformance of Object-oriented Runtime vIews of Architecture) was proposed in [3]. The approach involves specifying security architectures using the Acme architecture description language [35, 89] and formalizing constraints using

[1] DREAD is a risk assessment model for risk rating security threats using five categories: *Damage*, *Reproducibility*, *Exploitability*, *Affected users*, and *Discoverability*.

first-order logic. The constraints focus mostly on global information flow vulnerabilities such as spoofing, tampering, and information disclosure according to the STRIDE classification and based on previous work in developing an STRIDE-based security model in Acme [4]. The security model is based on data-flow diagrams (DFDs).

At the code level, a framework for detecting flaws in code was defined in [13]. The code is first transformed to STRIDE DFDs using static analysis. Then, based on a “best practice” repository where threat patterns are stored, an automatic check is performed to detect the threats and identify security measures that may be applied as annotations to DFDs to mitigate these threats.

In general, the output of these methods is a set of recommendations and guidelines to detect, evaluate, and mitigate possible threats. This eases in defining security requirements. However, a global closed-loop process has not been applied to be able to iterate to specify the complete set of requirements. It should be noted that most of these works have limitations in formalizing threats (only a subset of threat categories are considered), in reusing and extending them, and in automating the verification process. With this contribution, we propose to improve the global process and to apply it on a concrete security architecture design. We also propose tool support to facilitate the iteration of the loop. Our focus is to apply formal methods with tool support by determining how to design a secure system with the addition of more formality.

5.3 Methodology for the creation of a design and analysis framework

In this section, we present an application of the methodology proposed in Section 3.3 to the development of secure software architecture, from the negative perspective (i.e., threat). Particularly, we consider architecture threat modeling in the context of component-port-connector architectures and message passing communication. As depicted in Figure 5.1, the methodology is composed of several phases and activities. It supports the development of verified and reusable model libraries to represent threat detection and treatment for specifying secure software architectures of distributed systems.

Conceptual modeling. We begin by adding a new package called *Property* to the software architecture meta-model, introduced in the previous chapter, to capture security concerns (problem and solution) as properties of a modeled system. Security threats concerns (problem and solution) will be provided as property model libraries for reuse.

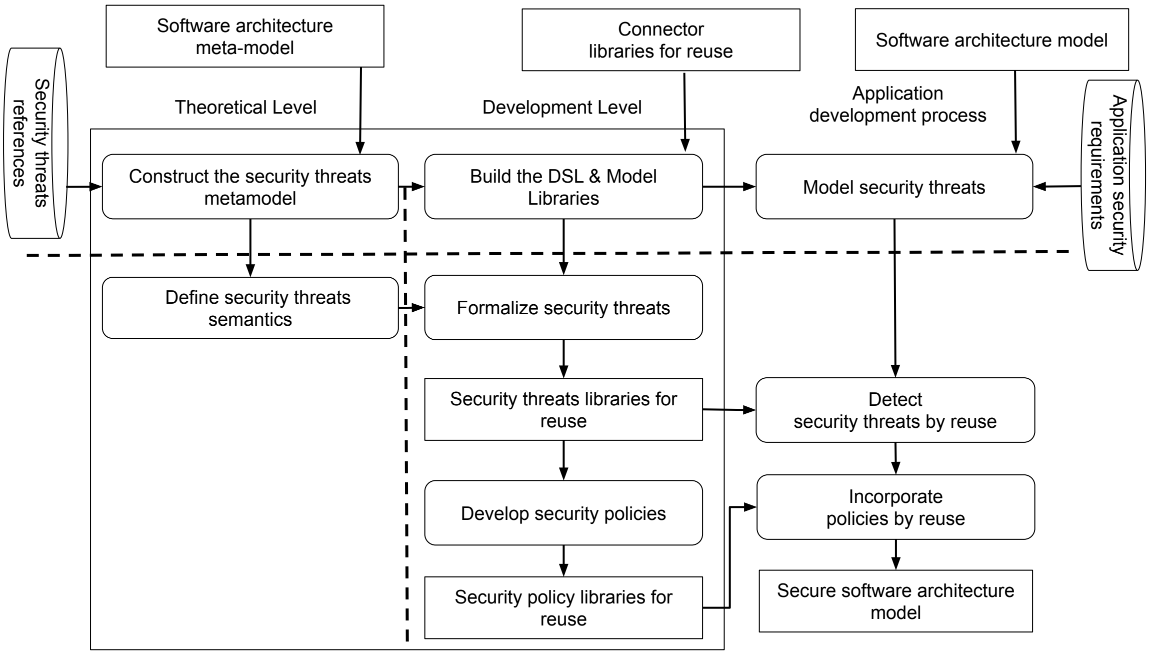

Software architecture meta-model
Connector libraries for reuse
Software architecture model
Security threats references
Theoretical Level
Development Level
Application development process
Application security requirements
Construct the security threats metamodel
Build the DSL & Model Libraries
Model security threats
Define security threats semantics
Formalize security threats
Security threats libraries for reuse
Detect security threats by reuse
Develop security policies
Incorporate policies by reuse
Security policy libraries for reuse
Secure software architecture model

Figure 5.1: Methodology for the creation of a design and analysis framework

In other words, *security properties* will be used to annotate the corresponding software architecture elements. Then, extending the logical specification of software architecture, we defined semantics to capture both legitimate and illegitimate message providers and consumers in the context of component-port-connector distributed systems. Within this technology-independent setting, we specify classes of threats based on the STRIDE threat classification so that they can be formalized to verify desirable properties of the components, ports, and connectors that will be used to elicit a set of security policies for the development of secure software architectures for distributed systems.

Development. After constructing the property meta-model and defining the security threat semantics upon the communication model (message passing connector), we developed (1) a DSL to model properties and (2) a formal modeling environment as an interpretation of the meta-model and the logical specification of the STRIDE security threats. Moreover, we provided a set formal model library for reuse to specify the security threats and policies. By doing so, we obtain a formal specification of some representative security threats for each STRIDE threat category. This gives us a set of reusable threat libraries (properties) capable of identifying security threats in a concrete software architecture model and for developing specifications of security solutions (policies) to mitigate these threats. The result is a set of abstract formal security solution modules that can be easily reused. This formalization step is required to leverage available tool support used in software and system modeling to enable straightforward instantiation and model checking capabilities to support the elicitation of an appropriate set of security policies to treat any detected threats.

5.4 Supporting security-by-design within the SDLC

The overall approach can easily fit well into various systems development life cycles (SDLCs) as a supplement to the requirements specification and architecture design phases, as shown in the right part of Figure 5.1. Conceptually, our approach is similar to the well-known approaches introduced in Section 2.5.1. We build a concrete architecture by instantiating the abstract architecture metamodel into a concrete model for an application-specific distributed software system, as presented in Section 4.3. Then, using the application-specific system security requirements, we use the security threat libraries to identify potential threats in the system model by checking that it satisfies the desired properties reflecting the security requirements. If we determine that the system model does not satisfy the security requirements, we use our developed libraries to incorporate security solutions (policies) to treat those threats in the software architecture model. We then verify the satisfaction of the security requirements in the updated software architecture model. In this way, we iterate over the requirements specification and architecture design to revise and improve the set of security policies that will help to mitigate the threats. As a result, we converge on a complete set of system security policies.

For simplicity, as shown in Figure 5.2, we illustrate the application of the approach within the Royce iterative waterfall SDLC [118]. The activities defined in Section 5.3 come as a supplement to the existing phase as follows. (1) The requirements specification is extended with the "Formalize the (New) Security Threats" and "Develop (New) Security Policies" activities. These activities concern the modeling framework development process and should only be done if a new threat or security policy is needed to be added in the framework, i.e., if a needed threat or security policy is not already defined in the framework libraries. (2) The architecture design is extended with the "Model Software Architecture" and "Detect Security Threats" 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 of the system, suggesting new security policies in the "Incorporate Security Policies" activity to protect against the detected threats. In this way, we can iterate over the requirements specification and architecture design phases

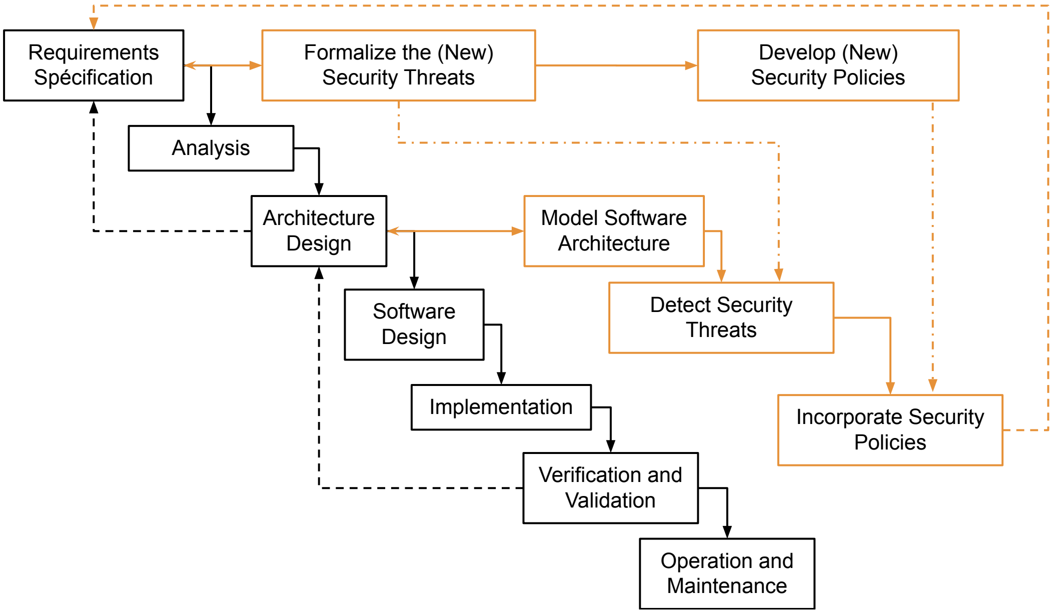

Requirements Spécification
Formalize the (New) Security Threats
Develop (New) Security Policies
Analysis
Architecture Design
Model Software Architecture
Software Design
Detect Security Threats
Implementation
Incorporate Security Policies
Verification and Validation
Operation and Maintenance

Figure 5.2: The proposed threat modeling approach within the Royce iterative waterfall SDLC

of the SDLC to revise and improve the architecture design of the system and mitigate the threats. As a result, the proposed approach enables the generation of formal artifacts early enough in the lifecycle to apply useful analysis within the design loop, thereby supporting the principles of security-by design.

5.5 Property view

The software architecture meta-model presented in Section 4.5 is extended with a *property view* for describing threats and policies in the form of categories to build property model libraries for reuse, as visualized in Figure 5.3. A *PropertyCategory* is a classification of properties. For instance, *Spoofing* and *Tampering* are defined as categories within the STRIDE library. These libraries are then used as external models to type the properties of the components and connectors. In addition, we define *mitigate* as a link between properties to capture relationships between policies and threats.

In the following, we present the precise definition of a set of threats using first-order logic and modal logic as a formalism that is abstract and technology-independent. This provides a more generic and understandable approach with mapping support to different existing property specification languages used in modeling and software development.

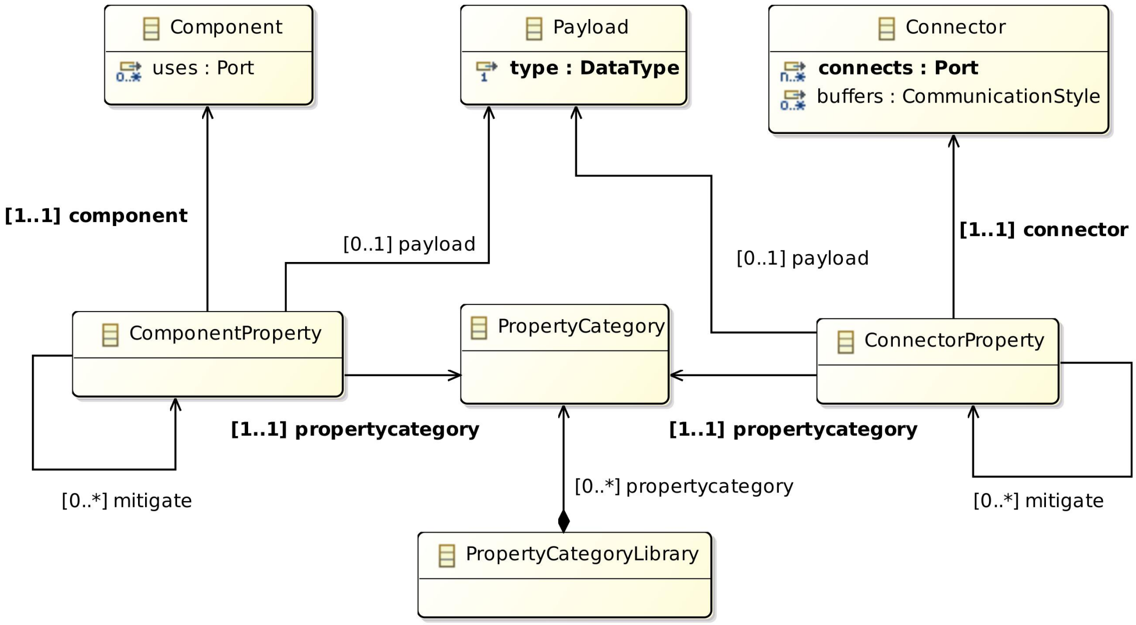


Figure 5.3: Property meta-model with *mitigate* relationships

5.5.1 Logical specification

In this section we extend the logical specification presented in Section 4.6.1. In the context of engineering secure systems, we define the following domain to capture the notion of both legitimate and illegitimate message providers (e.g., *inject*) and message consumers (e.g., *intercept*) of the system messages. These concepts are defined on top of the basic communication primitives (e.g., *send* and *receive*).

Sets

- $\mathcal{I}$ is the set of intervals
- $\mathcal{D}$ is the set of message payloads is extended with the following predicates:
  - has_freshness(d, i) indicates that $i \in \mathcal{I}$ is the interval of freshness of the payload $d \in \mathcal{D}$, i.e. the interval where the payload is considered relevant
- $\mathcal{M}$ 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 $d \in \mathcal{D}$

- $\mathcal{A}$ is the set of actions is extended with:
  - inject(m) denotes that component $c \in \mathcal{C}$ adds message $m \in \mathcal{M}$ into the system
  - intercept(m) denotes that component $c \in \mathcal{C}$ gets message $m \in \mathcal{M}$ from the system
  - set_src(m, s) denotes that component $c \in \mathcal{C}$ sets the declared source $s \in \mathcal{C}$ for message $m \in \mathcal{M}$
  - set_rcv(m, r) denotes that component $c \in \mathcal{C}$ sets the intended receiver $r \in \mathcal{C}$ for message $m \in \mathcal{M}$
  - set_pld(m, d) denotes that component $c \in \mathcal{C}$ sets the payload $d \in \mathcal{D}$ for message $m \in \mathcal{M}$
  - 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
  - get_rcv(m, r) denotes that component $c \in \mathcal{C}$ gets the intended receiver $r \in \mathcal{C}$ from message $m \in \mathcal{M}$, where r is not necessarily the actual receiver
  - get_pld(m, d) denotes that component $c \in \mathcal{C}$ gets the payload $d \in \mathcal{D}$ from message $m \in \mathcal{M}$, where d is not necessarily the true payload

Modalities

- $\mathbb{E}_c(a)$ is a predicate indicating that action $a \in \mathcal{A}$ is **$\mathbb{E}$nabled** for component $c \in \mathcal{C}$
- $\mathbb{Z}_c(a)$ is a predicate indicating that action $a \in \mathcal{A}$ is **authori$\mathbb{Z}$ed** for component $c \in \mathcal{C}$
- $\mathbb{T}_c(a)$ is a predicate indicating that component $c \in \mathcal{C}$ is **held accoun$\mathbb{T}$able** for action $a \in \mathcal{A}$
- $\mathbb{I}(a, i)$ is a predicate indicating that action $a \in \mathcal{A}$ will occur **within the $\mathbb{I}$nterval** $i \in \mathcal{I}$

Axioms

- $\forall\ a \in \mathcal{A},\ \forall\ c \in \mathcal{C}\ \cdot\ \mathbb{H}_c(a) \Rightarrow \mathbb{E}_c(a)$ is an axiom that states if an action a happen for a component c, then a is enabled for c
- $\forall\ m \in \mathcal{M},\ \forall\ c_1, c_2 \in \mathcal{C}\ \cdot\ \mathbb{H}_{c_2}(\mathit{inject}(m)) < \mathbb{E}_{c_1}(\mathit{intercept}(m))$ is an axiom that states if a component c_1 inject a message m, then before m was injected by a component c_2

- $\forall\ m \in \mathcal{M},\ \forall\ c_1, c_2 \in \mathcal{C}\ \cdot\ \mathbb{H}_{c_1}(\mathit{set\_rcv}(m, c_2))\ <\ \mathbb{H}_{c_1}(\mathit{inject}(m))$ is an axiom that states if a component $c1$ inject a message m, then before c_1 set the intended receiver of m as a component c_2

- $\forall\ m \in \mathcal{M},\ \forall\ c_1, c_2 \in \mathcal{C}\ \cdot\ \mathbb{H}_{c_1}(\mathit{set\_src}(m, c_2))\ <\ \mathbb{H}_{c_1}(\mathit{inject}(m))$ is an axiom that states if a component $c1$ inject a message m, then before c_1 set the source of m as a component c_2.

- $\forall\ m \in \mathcal{M},\ \forall\ c_1 \in \mathcal{C},\ \forall\ d \in\ \mathcal{D}\ \cdot\ \mathbb{H}_{c_1}(\mathit{set\_pld}(m, d))\ <\ \mathbb{H}_{c_1}(\mathit{inject}(m))$ is an axiom that states if an component $c1$ injected a message m, then before c_1 set the payload of m as a payload d

- $\forall\ m \in \mathcal{M},\ \forall\ c_1, c_2 \in \mathcal{C} \cdot \mathbb{H}_{c_1}(\mathit{intercept}(m))\ \wedge\ \mathit{has\_rcv}(m, r)\ <\ \mathbb{E}_{c_1}(\mathit{get\_rcv}(m, c_2))$ is an axiom that states if a component c_1 is able to get the intended receiver c_2 of message m , then before a component $c1$ intercepted a message m that contained a receiver c_2

- $\forall\ m \in \mathcal{M},\ \forall\ c_1, c_2 \in \mathcal{C} \cdot \mathbb{H}_{c_1}(\mathit{intercept}(m))\ \wedge\ \mathit{has\_src}(m, r)\ <\ \mathbb{E}_{c_1}(\mathit{get\_src}(m, c_2))$ is an axiom that states if a component c_1 is able to get the source c_2 of message m , then before an component $c1$ intercepted a message m that contained a source c_2

- $\forall\ m \in \mathcal{M},\ \forall\ c \in \mathcal{C},\ \forall\ d \in\ \mathcal{D}\ \cdot$
  $$\mathbb{H}_c(\mathit{intercept}(c, m))\ \wedge\ \mathit{has\_pld}(m, d)\ <\ \mathbb{E}_c(\mathit{get\_pld}(m, d))$$
  is an axiom that states if a component c_1 is able to get the payload d of message m, then before a component $c1$ intercepted a message m that contained a payload d

Macros

- $\text{send\_by}(m, s) \equiv \mathbb{H}_s(\text{inject}(m))\ \wedge\ \text{has\_src}(m, s)$

- $\text{send\_to}(m, c) \equiv \exists\ s \in \mathcal{C}\ \mid\ \text{send\_by}(m, s)\ \wedge\ \text{has\_rcv}(m, c)$

- $\text{send\_with}(m, d) \equiv \exists\ s \in \mathcal{C}\ \mid\ \text{send\_by}(m, s) \wedge \text{has\_pld}(m, d)$

5.5.2 STRIDE security threats

Our intent is to illustrate the approach described in Section 5.3 using representative threats in the context of a component-port-connector architecture model and message passing communication. In each of the following sections, we specify a representative property for each STRIDE category such that the violation of the specified property indicates the presence of the threat. This is not to say that satisfaction of the property guarantees that the threat is not present, i.e., each the property represents a sufficient condition for threat presence.

Spoofing. Spoofing refers to the impersonation of a component in the system for the purpose of misleading other system entities into falsely believing that an attacker is legitimate. Spoofing threats violates the authentication objectives of a system. In the context of message passing communication, spoofing threats take the form of message senders falsely claiming to be other system components, to entice other components to believe that the spoofed component is the originator of the message. Therefore, a spoofing threat can be identified by verifying whether the system ensures that all of the senders of message are authentic, i.e., *the sender of a message is always the originator of the message.* For components $c_1, c_2 \in \mathcal{C}$, we denote this representative property as $\mathit{SenderSpoofing}(c_1, c_2)$ which is specified for all messages $m \in \mathcal{M}$ as:

$$\text{send\_by}(m, c_1) \;<\; \mathbb{E}_{c_2}(\mathit{get\_src}(m, c_1)) \tag{5.1}$$

This property shows that every message that is received by a component c_2 that it believes was sent by component c_1 was actually sent by c_1 and not by any other component. In this way, there is no way in which a malicious component can pretend to be the sender of the message.

Example of Spoofing instantiation. Recalling our motivating college library website example (see Section 2.8.1), an example where the *Sender Spoofing* threat can be found is when a user is able to send a request to visualize a book page on the web server with the identity of another user. Consequently, the invalidity of the property $\mathit{SenderSpoofing}(\mathit{User}, \mathit{Webserver})$ would show that the *Sender Spoofing* threat exists in the system.

Tampering. Tampering refers to the unauthorized modification of data. Tampering threats violates the integrity objectives of a system. They often involve the modification of data in transit. Therefore, a tampering threat can be identified by verifying whether the message that was sent by a sender is the same message that was received by the receiver, i.e., *a message is not altered in transit.* For components $c_1, c_2 \in \mathcal{C}$, we denote this representative property as $\mathit{PayloadTampering}(c_1, c_2)$ which is specified for all messages $m \in \mathcal{M}$ and all payloads $d \in \mathcal{D}$ as:

$$(\text{send\_by}(m, c_1) \;\wedge\; \text{send\_with}(m, d)) \;<\; \mathbb{E}_{c_2}(\text{get\_pld}(m, d)) \tag{5.2}$$

This property shows that for every received message m with a payload d by a component c_2 from component c_1, there exists a message m that is sent with that exact payload d.

Example of Tampering instantiation. Coming back to our motivating college library website example, the *Payload Tampering* threat can be found, for example, when somebody is able to modify a request to visualize a book page from a user to the web server. Therefore, the invalidity of the *PayloadTampering*(*User*, *Webserver*) property would show that the *Payload Tampering* threat exists in the system.

Repudiation. Repudiation refers to a component claiming to have not performed an action that was in fact performed. Repudiation threats result from a lack of audit ability and accountability in the system. Mitigating repudiation threats requires an ability to separate legitimate claims from false claims made by system components. Very often, this involves constructing an audit log that records what happened during system operation and which components were involved. Therefore, a repudiation threat can be identified by verifying whether every sent or received message trace exists in the system, i.e., *for every sent or received message the system holds a component accountable for this action.* For components $c_1, c_2 \in \mathcal{C}$, we denote this representative property as *SendReceiveRepudiation*(c_1, c_2) which is specified for all messages $m \in \mathcal{M}$ as:

$$\begin{aligned} \mathbb{H}_{c_1}(\text{inject}(m)) \quad &\Rightarrow \mathbb{T}_{c_1}(\text{inject}(m)) \;\vee \\ \mathbb{H}_{c_2}(\text{intercept}(m)) &\Rightarrow \mathbb{T}_{c_2}(\text{intercept}(m)) \end{aligned} \tag{5.3}$$

This property shows that for every message m, sent by component c_1 or received by component c_2, the system respectively holds the sender c_1 accountable for the send action (inject) of the message m, or the receiver c_2 accountable for the receive action (intercept) of the message m.

Example of Repudiation instantiation. With respect to our motivating college library website example, an example where the *Send/Receive Repudiation* threat can be found is when the web server is able to make a request to the database and nobody is held accountable for this request. As a result, the invalidity of the *SendReceiveRepudiation* (*Webserver*, *Database*) property would show that the *Send/Receive Repudiation* threat exists in the system.

Information disclosure. Information disclosure refers to the unauthorized exposure of information to a component for which it is not intended. Information disclosure threats violate the confidentiality objectives of a system. With consideration to message passing communication, information disclosure threats occur when components other than those for which a message was intended are able to receive the sent message. Therefore, an information disclosure threat can be identified by verifying whether a component other than the intended receiver(s) is able to receive a message, i.e., *all messages are delivered only to the intended receiver(s).* For components $c_1, c_2 \in \mathcal{C}$, we denote this representative property as *PayloadDisclosure*

(c_1, c_2) which is specified for all messages $m \in \mathcal{M}$ and all components $c3 \in \mathcal{C}$ such that $c3 \neq c1 \neq c2$ as:

$$\neg(\text{send\_by}(m, c_1) \;\wedge\; \text{send\_to}(m, c_2)) \;<\; \mathbb{H}_{c_3}(\text{intercept}(m)) \tag{5.4}$$

This property shows that every message m received (intercepted) by a component c_3 cannot be sent by another component c_1 to another intended component c_2 at any previous point in time. Thus, there is no way in which the contents of the message can be disclosed to an unauthorized component.

Example of information disclosure instantiation. Recalling the college library website example, an example where the *Payload Disclosure* threat can be found is when somebody other than the web server is able to intercept and read the content of a request from a user visualizing a book page on the web server. Consequently, the invalidity of the property *PayloadDisclosure*(*User*, *Webserver*) would show that the *Payload Disclosure* threat exists in the system.

Denial of service. Denial of service refers to the unauthorized withholding of a service to system components. Denial of service threats violates the availability objectives of a system. In the context of message passing communication, denial of service threats may involve blocking the transmission of messages from senders to receivers such that the receiver never receives any of the messages before their freshness expires, i.e., before the deadline where the message becomes irrelevant is reached. Therefore, a denial of service threat can be identified by verifying whether sent messages are delayed, destroyed, or deleted in transit, preventing them from being received before the freshness of the message expires or from being received by the intended receiver at all.

For components $c_1, c_2 \in \mathcal{C}$, we denote this representative property as *PayloadStaleness* (c_1, c_2) which is specified for all messages $m \in \mathcal{M}$, all payloads $d \in \mathcal{D}$, and all intervals $i \in \mathcal{I}$ as:

$$\begin{gathered}\mathbb{H}_{c_1}(\text{inject}(m)) \;\wedge\; \text{has\_rcv}(m, c_2) \;\wedge\; \text{has\_pld}(d, m) \;\wedge\; \text{has\_freshness}(d, i) \Rightarrow \\ \mathbb{I}(\mathbb{E}_{c_2}(\text{intercept}(c_2)m), i)\end{gathered} \tag{5.5}$$

Put another way, this property states that every message m sent by a component c_1 intended for a component c_2 and that contains a payload d with a freshness interval i has the opportunity to be received at some point by the intended receiver c_2 in the interval i. This represents a typical *liveness* property of the system.

Example of denial of service instantiation. In the context of the college library website example, the *Payload Staleness* threat can be found, for example, when a user request to visualize a book page on the web server is unable to reach the server in an acceptable time. Consequently, the invalidity of the property *PayloadStaleness*(*User*, *Webserver*) would show that the *Payload Staleness* threat exists in the system.

Elevation of privilege. Elevation of privilege refers to the ability of a component to gain capabilities without proper authorization to have such capabilities. Elevation of privilege threats violate the authorization objectives of a system. Typically, an elevation of privilege threat involves a system component being able to perform an action for which they are not authorized according to the system access control policy. Very often, mitigating elevation of privilege attacks involves enforcing the system's access control policy. Therefore, an elevation of privilege threat can be identified by verifying whether it is possible for a component to perform any actions without having the proper authorizations, i.e., *every component that performs an action (send/receive) has the allowable permissions at the time the action is performed.* For components $c_1, c_2 \in \mathcal{C}$, we denote this representative property as *SendReceiveElevation*(c_1, c_2) which is specified for all messages $m \in \mathcal{M}$ as:

$$\begin{aligned}\mathbb{H}_{c_1}(\text{inject}(m)) &\Rightarrow \mathbb{Z}_{c_1}(\text{inject}(m)) \;\vee \\ \mathbb{H}_{c_2}(\text{intercept}(m)) &\Rightarrow \mathbb{Z}_{c_2}(\text{intercept}(m))\end{aligned} \tag{5.6}$$

This property shows that for every message m, sent by component c_1 or received by component c_2, there is an authorization for the sender c_1 or receiver c_2 with the corresponding permission to perform the action (inject or intercept) with respect to the specific message m that is being sent or received. In this way, the system verifies that the access control policy is enforced prior to enabling components to perform specific actions.

Example of elevation of privilege instantiation. Revisiting our motivating college library website example, the *Send/Receive Elevation* threat can be found, for example, when a user requests to visualize a book page on the web server without having the correct authorization to access this page. As a result, the invalidity of the property *SendReceiveElevation*(*User*, *Webserver*) would show that the *Send/Receive Elevation* threat exists in the system.

> *At this level of specification, we are able to specify the STRIDE set of security threats as properties of the system architecture model, using first-order and modal logic as an abstract and technology-independent formalism that can be interpreted in appropriate formal tooled languages.*

5.6 Formal specification and analysis in Alloy

In this section, we provide the specification and verification of a representative threat from each STRIDE threat category using *Alloy*. By operating as a counterexample generator, when the Alloy Analyzer identifies the violation of a property, it indicates the existence of a threat. In much the same way, by operating as a model checker, the Alloy Analyzer enables the use of a property as a security policy to indicate the absence of the threat. This enables us to construct models incrementally, allowing rapid iterations between modeling and analysis when writing a specification. For more formal definitions and examples on Alloy, the reader is referred to Section 2.6.2.

5.6.1 Formalizing the negative perspective of the property metamodel

We encode the set of actions described in Section 5.5.1 considering the notion of execution steps in the context of an asynchronous message-passing system as follows:

- $\mathcal{T}$ is the sequence of execution steps
- Each macro mc($param_1, \ldots, param_n$) is transformed to mc($param_1, \ldots, param_n, t$) denoting its relation to the execution step $t \in \mathcal{T}$. For example, injected(c, m) is transformed to injected(c, m, t) indicating that a sender $c \in \mathcal{C}$ added a message $m \in \mathcal{M}$ into the system before the step $t \in \mathcal{T}$

The *MsgPassing* concept is then extended with a set of attributes defined on the basis of these actions and their corresponding steps that should be logged when they occur.

Listing 5.1 depicts the presentation of the new concept called *MsgPassingConstraint* and the *Inject* action in Alloy.

```
sig MsgPassingConstraint extends MsgPassing {
  set_pld: Component -> Tick,
  set_rcv: Component -> Tick,
  set_src: Component -> Tick,
  get_pld: Component -> Tick,
  get_rcv: Component -> Tick,
  get_src: Component -> Tick
}{
  all c:Component, t:Tick | got_src.t = c => some m:MsgPassing, s:
     Component | get_src[c,m,s,t]
  ...}
pred E_inject[c:Component,m:MsgPassing, t:Tick] {
  some con:ConnectorMPS, p:c.uses
    | p in con.connects.t and p.kind = OUTPUT
    and m.payload.type = p.realizes.datatypes}
pred H_inject[c1:Component,m:MsgPassing, t:Tick] {
  E_inject[c1,m,t]
  c1.send[m.receiver,m.payload,t]}
pred injected[c:Component,m:MsgPassing, t1:Tick] {
  one t2:t1.prevs | inject[c,m,t2]}
```

Listing 5.1: Message data and example of action in Alloy

We encode set of the axioms describes in Section 5.5.1 as facts (i.e., always true for any model instance). Listing 5.2 depicts the corresponding Alloy code for one of the axioms.

```
fact {
  all m:MsgPassingConstraint, c1:Component, t1:Tick |
    H_inject[c1,m,t1] =>
        some c2:Component, t2:t1.prevs | H_set_src[c1,m,c2,t2]
}
```

Listing 5.2: Axiom in Alloy

Threats can now be defined as *properties* in terms of a system specification, i.e., in terms of enabled actions, messages, components and connectors.

We encode the property view as described in Section 5.5 by defining the corresponding concepts *ComponentProperty* and *ConnectorProperty* as abstract signatures in Alloy. For each one, the helper function *holds* is used to easily verify that the property holds in a given model. As we shall see, security threats and policies will be encoded as properties (see Listing 5.4).

```
abstract sig ComponentProperty {
  comp: one Component,
  payl: one Payload}
fun ComponentProperty.holds[c1:one Component,p: one Payload]: set ComponentProperty {
  { m:this { m.comp = c1 and m.payl = p }}}
abstract sig ConnectorProperty {
  conn: one Connector,
  payl: one Payload
}{ comp1 != comp2 }
fun ConnectorProperty.comp1 : one Component {{ c:Component | this.conn.portI in c.uses}}
fun ConnectorProperty.comp2 : one Component {{ c:Component | this.conn.portO in c.uses}}
fun ConnectorProperty.holds[c1,c2:one Component,p: one Payload]: set ConnectorProperty{
  { m:this { m.comp1 = c1 and m.comp2 = c2 and m.payl = p }}}
```

Listing 5.3: Property view concepts in Alloy

5.6.2 STRIDE security threats

We aim to build a set of reusable libraries for threat detection and treatment. To do so, we model the threat categories presented in Section 5.5 in Alloy, using constructions such as *Predicate*, *Assertion*, and *Fact*. Threats will be specified as constraints in the model. Remember that during the logical specification, each threat category is associated with a representative property such that the violation of the specified property indicates the presence of the threat. Therefore, each threat is associated with a property defined as a predicate to map the logical definition of the corresponding threat property to the Alloy model describing the targeted software architecture and communication system (e.g., *NotBeSpoofed*). Then, the presence of a threat, as a result of the violation of the property, is detected by the Alloy Analyzer through an assertion finding a counterexample. As a result, an appropriate security property is defined as a predicate (e.g., *spoofProof*) to codify a security policy to constrain the operation of the system and to protect against the corresponding threat.

Spoofing. The *Sender Spoofing* threat is considered within the connector and can be identified by checking whether the communication system provided by the connector ensures that all messages transmitted through this connector have authentic senders. The

NotBeSpoofed property is defined as a predicate according to the logical specification of the *Sender Spoofing* threat (Expression (5.1)) and the system computing model. Note that in Listing 5.4, while the *Sender Spoofing* threat is considered within the connector, we define the *NotBeSpoofed* property (Lines 3-6) with respect to two components (c1 and c2) representing the endpoints of the connector being considered. This enables us to conform to the abstract system computing model described in Section 5.5.1.

```
1 sig SenderSpoofing extends ConnectorProperty {}{
2   not NotBeSpoofed[comp1,comp2,payl]}
3 pred NotBeSpoofed[c1,c2:Component, p:Payload] {
4   let m={m:MsgPassing | m.payload = p}|
5       all t2:Tick-tick/first | some t1:t2.prevs |
6       E_get_src[c2,m,c1,t2] implies send_by[m,c1,t1] }
7 assert MPSCanBeSpoofed {
8   all c1,c2:Component, p:Payload |
9     | no SenderSpoofing.holds[c1,c2,d]}
```

Listing 5.4: Detection of spoofing

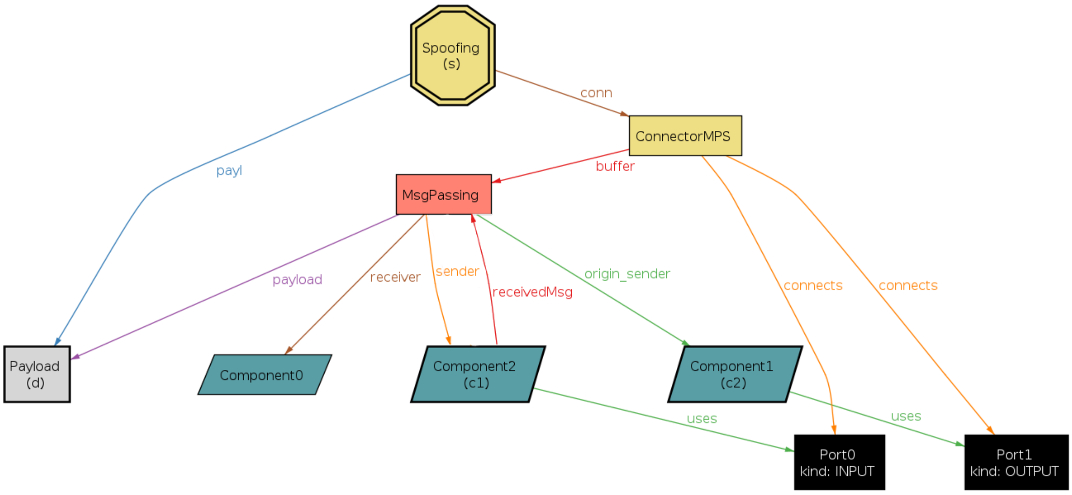


Figure 5.4: Spoofing counterexample provided by the Alloy Analyzer

The Alloy Analyzer detects a *Sender Spoofing* threat by finding a counterexample, as a violation of the *NotBeSpoofed* property. Figure 5.4 shows a model where the *Sender Spoofing* occurs. We proceed by defining a security policy as a predicate to protect against the *Sender Spoofing* threat. The idea is to ensure that no component can send a message by pretending to be some other component. The *spoofProof* property is defined as a predicate on the connector to ensure the authenticity of the sender of each message transmitted through this connector (Listing 5.5, Lines 1-4).

```
pred ConnectorMPS.spoofProof {
  all mp:MsgPassing, t:Tick |
    mp in this.buffer.t implies
      mp.sender = mp.origin_sender // The sender is authentic}
assert MPSCanNotBeSpoofed  {
  (all c:ConnectorMPS | c.spoofProof) implies
  (all c1,c2:Component, p:Payload |
    | no SenderSpoofing.holds[c1,c2,d])}
```

Listing 5.5: spoofProof property

According to the Alloy Analyzer, no counterexample was found. The satisfaction of the *spoofProof* property allows the fulfillment of the corresponding security policy to protect against the *Sender Spoofing* threat.

Tampering. The *Payload Tampering* threat is considered within the connector and can be identified by checking whether the communication system provided by the connector ensures that all messages transmitted through this connector are not altered in transit. The *NotBeAltered* property is defined as a predicate according to the logical specification of the *Payload Tampering* threat (Expression (5.2)) and the system computing model (Listing 5.4). We define this property (Lines 3-7) with respect to two components (`c1` and `c2`) representing the endpoints of the connector being considered for tampering threat. Once again, this is done to conform to the abstract system computing model described in Section 5.5.1.

```
1  sig PayloadTampering extends ConnectorProperty  {}{
2    not NotBeAltered[comp1,comp2,payl]}
3  pred NotBeAltered[c1,c2:Component, p:Payload] {
4    all p:Payload |  let m = {m:MsgPassing | m.payload = d}{
5        all t2:Tick-tick/first | some t1:t2.prevs |
6          (E_get_pld[c2,m,p,t2]) implies
7              (send_by[m,c1, t1] and send_with[m,p,t1])}
8  assert MPSCanBeAltered  {
9    all c1,c2:Component, p:Payload |
10     no PayloadTampering.holds[c1,c2,d}
```

Listing 5.6: Detection of tampering

The Alloy Analyzer detects a *Payload Tampering* threat by finding a counterexample, as a violation of the *NotBeAltered* property, showing a model where the tampering occurs. We proceed by defining a security policy as a predicate to protect against the *Payload*

Tampering threat. The idea is to ensure that no component can alter a message during transmission. The *tamperProof* property is defined as a predicate on the connector to ensure that the message that was sent by a sender is the same message that was received by the receiver through this connector (Listing 5.7, Lines 1-10).

```
1  pred  ConnectorMPS.tamperProof {
2    all t:Tick, mp:MsgPassing |
3      mp in this.buffer.t
4        implies (
5          no mp':MsgPassing-mp, t':t.prevs |
6          mp'.sender = mp.sender
7          and mp'.receiver = mp.receiver
8          and mp'.sent = mp.sent
9          and mp' in this.buffer.t'
10       )}
11 assert MPSDataCanNotBeAltered {
12   (all c:ConnectorMPS | c.tamperProof) implies
13   (all c1,c2:Component, p:Payload |
14     no PayloadTampering.holds[c1,c2,d])}
```

Listing 5.7: tamperProof property

According to the Alloy Analyzer, no counterexample was found. The satisfaction of the *tamperProof* property allows the fulfillment of the corresponding security policy to protect against the *Payload Tampering* threat.

Repudiation. The *Send/Receive Repudiation* threat is considered within the component and can be identified by checking whether the system holds a component accountable for each *inject* and *intercept* action that occurred. We define a new action *T_inject* (resp. *T_intercept*) as an abstract security mechanism to define the $\mathbb{T}_c(a)$ modality. The *NotBeRepudiated* property is defined as a predicate according to the logical specification of the *Send/Receive Repudiation* threat (Expression (5.3)) and the system computing model (Listing 5.8, Lines 3-9).

```
1  sig SendReceiveRepudiation extends ComponentProperty  {}{
2    not NotBeRepudiated[comp,payl]}
3  pred NotBeRepudiated[c:Component, p:Payload]{
4    all t1:Tick - tick/last, t2:Tick - tick/last |
5    let m = {m:MsgPassing | m.payload = p }{
6      (inject[c,m, t1] => all t1':t1.nexts | T_inject[c,m,t1'])
7      or (intercept[c,m, t2] => all t2':t2.nexts | T_intercept[c,m,t2'])}
8  }
```

```
assert MPSCanBeRepudiated {
    all c:Component, p:Payload |
      no SendReceiveRepudiation.holds[c,d]}
```

Listing 5.8: Detection of repudiation

The Alloy Analyzer detects a *Send/Receive Repudiation* threat by finding a counterexample, as a violation of the *NotBeRepudiated* property, showing a model where the repudiation occurs. We proceed by defining a security policy as a predicate to protect against the *Send/Receive Repudiation* threat. The idea is to ensure that no component can deny an action that was in fact performed. The *repudiationProof* property is defined as a predicate on the component to ensure that every sent or received message that the system got a trace of the corresponding action (Listing 5.9, Lines 1-5).

```
1  pred Component.repudiationProof {
2    all m:MsgPassing, t:Tick |
3      inject[this, m, t] implies Log.addEntry[m, t]
4    all m:MsgPassing, t:Tick |
5      intercept[this,m, t] implies Log.addEntry[m, t]
6  }
7  assert MPSCanNotBeRepudiated {
8    (all c:Component | c.repudiationProof) implies
9    (all c:Component, p:Payload |
10      no SendReceiveRepudiation.holds[c,d])}
```

Listing 5.9: repudiationProof property

According to the Alloy Analyzer, no counterexample was found. The satisfaction of the *repudiationProof* property allows the fulfillment of the corresponding security policy to protect against the *Send/Receive Repudiation* threat.

Information disclosure. The *Payload Disclosure* threat is considered within the connector and can be identified by checking whether the communication system provided by the connector ensures that all messages transmitted through this connector are delivered only to the intended receiver(s). The *NotBeIntercepted* property is defined as a predicate according to the logical specification of the *Payload Disclosure* threat (Expression (5.4)) and the system computing model. To conform to the abstract system computing model described in Section 5.5.1, we define the *NotBeIntercepted* property in Listing 5.10 (Lines 3-7) with respect to two components (`c1` and `c2`) representing the endpoints of the connector being considered for the *Payload Disclosure* threat in a manner similar to what was done for the *Sender Spoofing* and *Payload Tampering* threats above.

```
sig PayloadDisclosure extends ConnectorProperty {}{
  not NotBeIntercepted[comp1,comp2,payl]}
pred NotBeIntercepted[c1,c2:Component, p:Payload] {
  all c3:Component-c1-c2 | let m = {m:MsgPassing | m.payload = d } |
  all t2:Tick-tick/first | some t1:t2.prevs |
      H_intercept[c3,m,t2] implies
          not (send_by[m,c1,t1] and send_to[m,c2,t1])}
assert MPSCanBeIntercepted  {
    all c1,c2:Component, p:Payload |
      no PayloadDisclosure.holds[c1,c2,d]}
```

Listing 5.10: Detection of information disclosure

The Alloy Analyzer detects a *Payload Disclosure* threat by finding a counterexample, as a violation of the *NotBeIntercepted* property, showing a model where the *Payload Disclosure* occurs. We proceed by defining a security policy as a predicate to protect against the *Payload Disclosure* threat. The idea is to ensure that all messages transmitted through this connector cannot be intercepted by an undesired component. The *informationDisclosureProof* property is defined as a predicate on the connector to ensure that there is no component other than the intended receiver(s) that is able to receive a message sent through this connector (Listing 5.11, Lines 1-4).

```
1 pred  ConnectorMPS.informationDisclosureProof {
2   all mp:MsgPassing, t:Tick |
3   (mp in this.buffer.t)
4   implies this.connects.t = mp.origin_sender.uses + mp.receiver.uses}
5 assert MPSCanNotBeIntercepted  {
6     (all c:ConnectorMPS | c.informationDisclosureProof) implies
7     (all c1,c2:Component, p:Payload |
8         no PayloadDisclosure.holds[c1,c2,d])}
```

Listing 5.11: informationDisclosureProof property

According to the Alloy Analyzer, no counterexample was found. The satisfaction of the *informationDisclosureProof* property allows the fulfillment of the corresponding security policy to protect against the *Payload Disclosure* threat.

Denial of service. The *Payload Staleness* threat is considered within the connector and can be identified by checking whether the communication system provided by the connector ensures that every message that is sent through this connector will be eventually received by the intended receiver before the content of the message lost its freshness. The

NotBeReceivedStale property is defined as a predicate according to the logical specification of the *Payload Staleness* threat (Expression (5.5)) and the system computing model. We define the *NotBeReceivedStale* property in Listing 5.12 (Lines 3-5) with respect to two components (`c1` and `c2`) representing the endpoints of the connector being considered for the *Payload Staleness* threat to conform to the abstract system computing model described in Section 5.5.1.

```
1  sig PayloadStaleness extends ConnectorProperty  {}{
2    not NotBeReceivedStale[comp1,comp2,payl]}
3  pred NotBeReceivedStale[c1,c2:Component, d:Payload] {
4      all i:Interval, t1:Tick, t2:t1.nexts | let m = {m:MsgPassing | m.
           payload = d }{
5      inject[c1,m,t1] and m.receiver = c2 and d.freshness = i implies
           E_intercept[c2,m,t2] and  I_in[t2, d.freshness]}}
6  assert MPSCanBeStale  {
7       all c1,c2:Component, d:Payload |
8          no PayloadStaleness.holds[c1,c2,d]}
```

Listing 5.12: Detection of denial of service

The Alloy Analyzer detects a *Payload Staleness* threat by finding a counterexample, as a violation of the *NotBeReceivedStale* property, showing a model where the denial of service occurs. We proceed by defining a security policy as a predicate to protect against the *Payload Staleness* threat. The idea is to ensure that all messages transmitted through this connector are not being delayed, destroyed or deleted (i.e., dropped) in transit and made available to the receiver(s) before it content becomes stale.

The *staleProof* property is defined as a predicate on the connector to ensure that all messages transmitted through this connector have the opportunity to be received by the intended receiver(s) before their content loses their freshness (Listing 5.13, Lines 1-3).

```
1  pred  ConnectorMPS.staleProof {
2    all t:Tick - tick/last,  c:Component, mp:MsgPassing
3        | mp in this.buffer.t implies ((mp in this.buffer.(t.next) and mp.
             payload.freshness.end.lt[t.next]) or (mp.receiver).receive[c,mp
             .payload,t.next])}
4  assert MPSCanNotBeStale  {
5      (all c:ConnectorMPS | c.staleProof)
6      implies ( all c1,c2:Component, d:Payload |
7          no PayloadStaleness.holds[c1,c2,d])}
```

Listing 5.13: staleProof property

According to the Alloy Analyzer, no counterexample was found. The satisfaction of the *staleProof* property allows the fulfillment of the corresponding security policy to protect against the *Payload Staleness* threat.

Elevation of privilege. The *Send/Receive Elevation* threat is considered within the component and can be identified by checking that every component that performs an action (*inject/intercept*) is authorized. We define a new action *Z_inject* (resp. *Z_intercept*) as an abstract security mechanism to define the $\mathbb{Z}_c(a)$ modality. The *NoEop* property is defined as a predicate according to the logical specification of the *Send/Receive Elevation* threat (Expression (5.6)) and the system computing model (Listing 5.14, Lines 3-7).

```
1 sig SendReceiveElevation extends ComponentProperty {}{
2   not NoEop[comp, payl]}
3 pred NoEop[c:Component, p:Payload] {
4   all t1:Tick, t2:Tick |
5   let m = {m:MsgPassing | m.payload= p }{
6    (inject[c,m,t1] implies Z_inject[c,m])
7    or (intercept[c,m,t2] implies Z_intercept[c,m])}}
8 assert MPSCanBeEop  {
9     all c:Component, p:Payload | no SendReceiveElevation.holds[c,d]}
```

Listing 5.14: Detection of elevation of privilege

The Alloy Analyzer detects a *Send/Receive Elevation* threat by finding a counterexample, as a violation of the *NoEoP* property, showing a model where the *Send/Receive Elevation* occurs. We proceed by defining a security policy as a predicate to protect against the *Send/Receive Elevation* threat. The idea is to ensure that it is not possible for any system component to perform any actions without having the proper authorizations according to an access control list. The *EoPProof* property is defined as a predicate on the component to ensure that all the actions performed by this component are allowed (Listing 5.15, Lines 1-5).

```
1 pred Component.EoPProof {
2   all m:MsgPassing, t:Tick |
3     inject[this,m,t] implies Z_inject[this,m]
4   all m:MsgPassing, t:Tick |
5     intercept[this,m,t] implies Z_intercept[this,m]
6 assert MPSCanNotBeEop  {
7   (all c:Component | c.EoPProof) implies
8   (all c1,c2:Component, p:Payload | no SendReceiveElevation.holds[c,d])}
```

Listing 5.15: EoPProof property

According to the Alloy Analyzer, no counterexample was found. The satisfaction of the *EoPProof* property allows the fulfillment of the corresponding security policy to protect against the *Send/Receive Elevation* threat.

> *At this level of specification, we are able to: (1) formally represent an interpretation of the STRIDE set of threats described in Section 5.5 and develop a set of policies to treat them as properties of the architecture model in Alloy, (2) provide the properties as formal model libraries to support reuse, and (3) offer methods and functions to easily verify that the properties hold in a given architecture model fostering reuse.*

5.7 Tool support

We have implemented a prototype to support the proposed approach (Section 5.3) as an Eclipse plug-in. Our starting point is the software architecture metamodel presented in Section 5.5.1 as the metamodel for a software architecture DSL. 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.

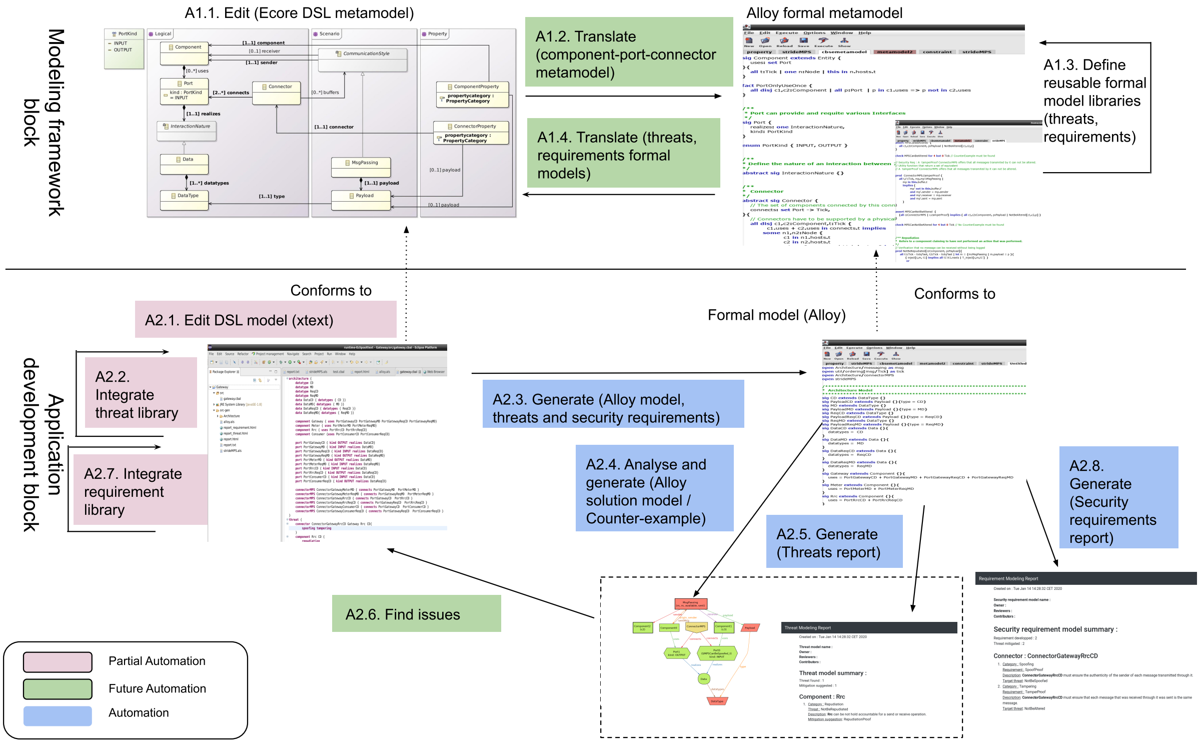

Modeling framework block
A1.1. Edit (Ecore DSL metamodel)
A1.2. Translate (component-port-connector metamodel)
Alloy formal metamodel
A1.3. Define reusable formal model libraries (threats, requirements)
A1.4. Translate (threats, requirements formal models)
Conforms to
Conforms to
Application development block
A2.1. Edit DSL model (xtext)
Formal model (Alloy)
A2.2. Integrate threat library
A2.3. Generate (Alloy model, threats and security requirements)
A2.4. Analyse and generate (Alloy solution model / Counter-example)
A2.5. Generate (Threats report)
A2.6. Find issues
A2.7. Integrate requirement library
A2.8. Generate (Security requirements report)
Partial Automation
Future Automation
Automation

Figure 5.5: Tool support architecture and artifacts of the approach

The architecture of the tool, as shown in Figure 5.5, is composed of two main blocks: (1) Modeling framework block and (2) Application development block. Each block is composed of a set of modules to support the corresponding set of activities (the numbers in parentheses correspond to the activity numbers in Figure 5.5).

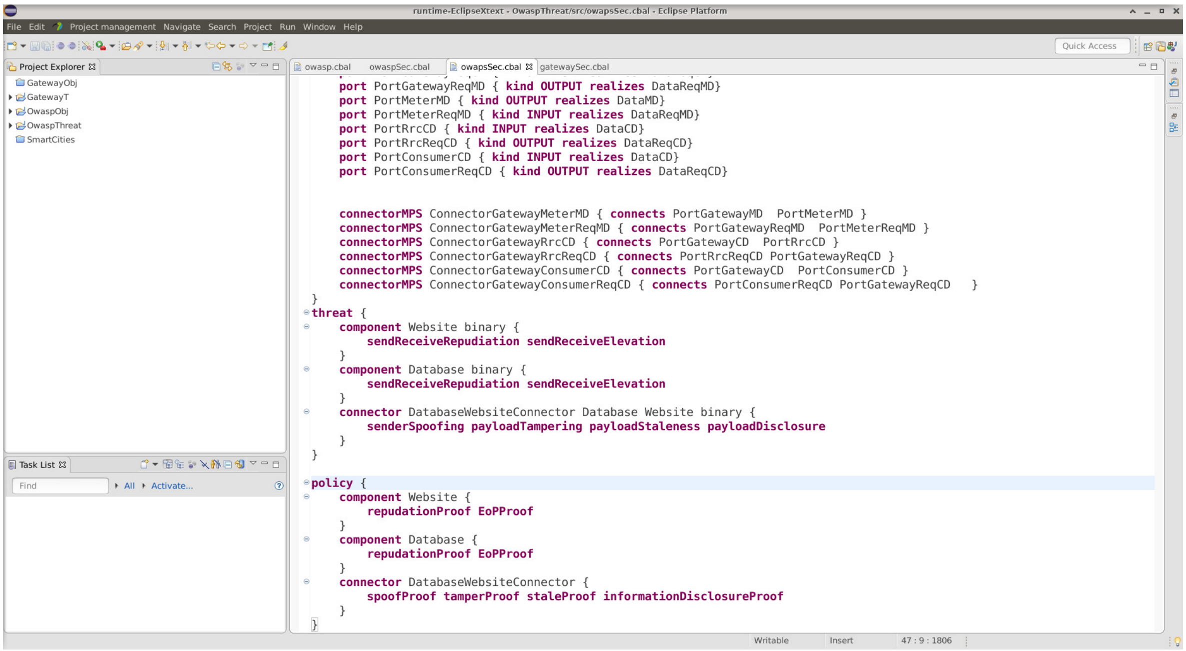
5.7.1 Modeling framework block

The first block is dedicated to supporting four activities. Activity *A1.1* is responsible for creating the DSL metamodel. The resulting metamodel is used to formally define (in Alloy) the static and behavioral semantics, based upon the DSL metamodel according to the procedure discussed in Section 5.6. We will name this definition the formal metamodel (*A1.2*). Furthermore, reusable threat model libraries and security policies model libraries are also defined as Alloy models according to the procedure discussed in Section 5.6.2 (*A1.3*). We will name this definition the formal model libraries. The last activity (*A1.4*) is the definition of a set of DSL model libraries from the Alloy specification of threats and policies (formal model libraries), using the *property view* of the DSL metamodel (Figure). Each threat property is associated with a set of security policies to mitigate it, during *A1.3*. As an example, we use `spoofing` as an instance of `PropertyCategory`, `senderSpoofing` as an instance of the `ConnectorProperty` within the *StrideThreat* library and `spoofProof` as an instance of the `ConnectorProperty` within the *SecurityPolicy* library. Finally, we set up that the `spoofProof` requirement *mitigates* the `senderSpoofing` threat. To support these four activities, we used EMF, Xtext to develop the DSL metamodel, and the Alloy Analyzer to encode the corresponding formal metamodel. No further implementation was required.

5.7.2 Application development block

The second block is dedicated to supporting eight activities. Activity *A2.1* allows the designer to model a software architecture (DSL model) conforming to the DSL metamodel, where *A2.2* allows to integrate the threats specification reusing the already developed threat model libraries (*A1.4*) as depicts in Figure5.6 for the college library web application.

Then, *A2.3* is the generation of a formal model (in Alloy) from a DSL model, through a transformation engine. The Alloy Analyzer is then invoked, with several iterations, to detect the targeted threat (*A2.4*) and the report on the detected threats may be generated as an HTML document (*A2.5*), such as visualized in Figure 5.8. The report is then analyzed (*A2.6*) and *A2.7* allows the designer to add the corresponding security policies

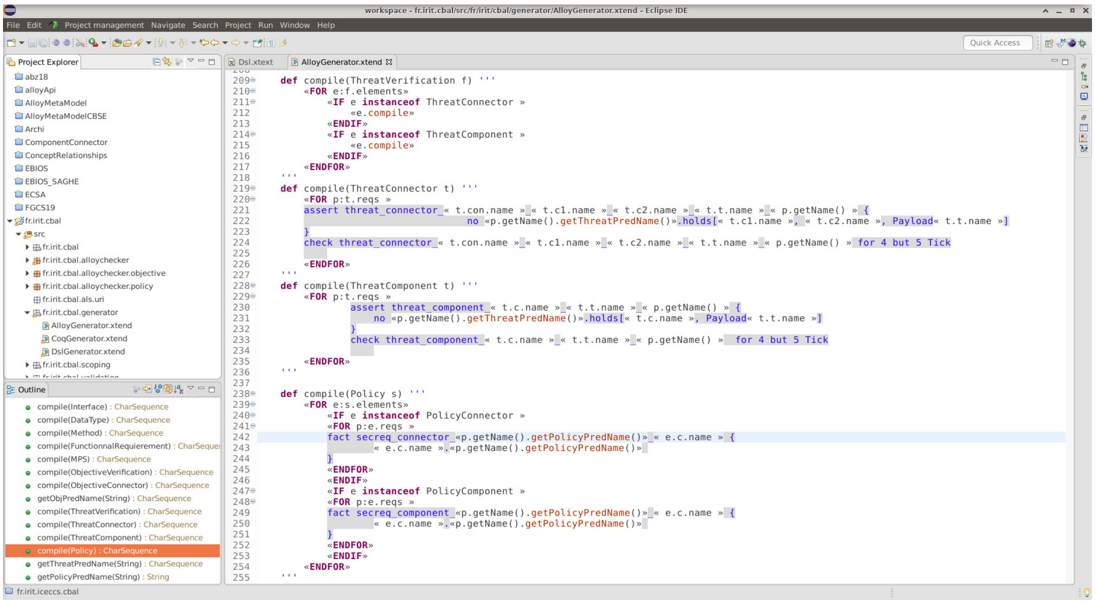


```
    port PortGatewayReqMD { kind OUTPUT realizes DataReqMD}
    port PortMeterMD { kind OUTPUT realizes DataMD}
    port PortMeterReqMD { kind INPUT realizes DataReqMD}
    port PortRrcCD { kind INPUT realizes DataCD}
    port PortRrcReqCD { kind OUTPUT realizes DataReqCD}
    port PortConsumerCD { kind INPUT realizes DataCD}
    port PortConsumerReqCD { kind OUTPUT realizes DataReqCD}


    connectorMPS ConnectorGatewayMeterMD { connects PortGatewayMD  PortMeterMD }
    connectorMPS ConnectorGatewayMeterReqMD { connects PortGatewayReqMD  PortMeterReqMD }
    connectorMPS ConnectorGatewayRrcCD { connects PortGatewayCD  PortRrcCD }
    connectorMPS ConnectorGatewayRrcReqCD { connects PortRrcReqCD PortGatewayReqCD }
    connectorMPS ConnectorGatewayConsumerCD { connects PortGatewayCD  PortConsumerCD }
    connectorMPS ConnectorGatewayConsumerReqCD { connects PortConsumerReqCD PortGatewayReqCD   }
}
threat {
    component Website binary {
        sendReceiveRepudiation sendReceiveElevation
    }
    component Database binary {
        sendReceiveRepudiation sendReceiveElevation
    }
    connector DatabaseWebsiteConnector Database Website binary {
        senderSpoofing payloadTampering payloadStaleness payloadDisclosure
    }
}

policy {
    component Website {
        repudationProof EoPProof
    }
    component Database {
        repudationProof EoPProof
    }
    connector DatabaseWebsiteConnector {
        spoofProof tamperProof staleProof informationDisclosureProof
    }
}
```

Figure 5.6: Definition of the security requirements using threats modeling for the college library web application

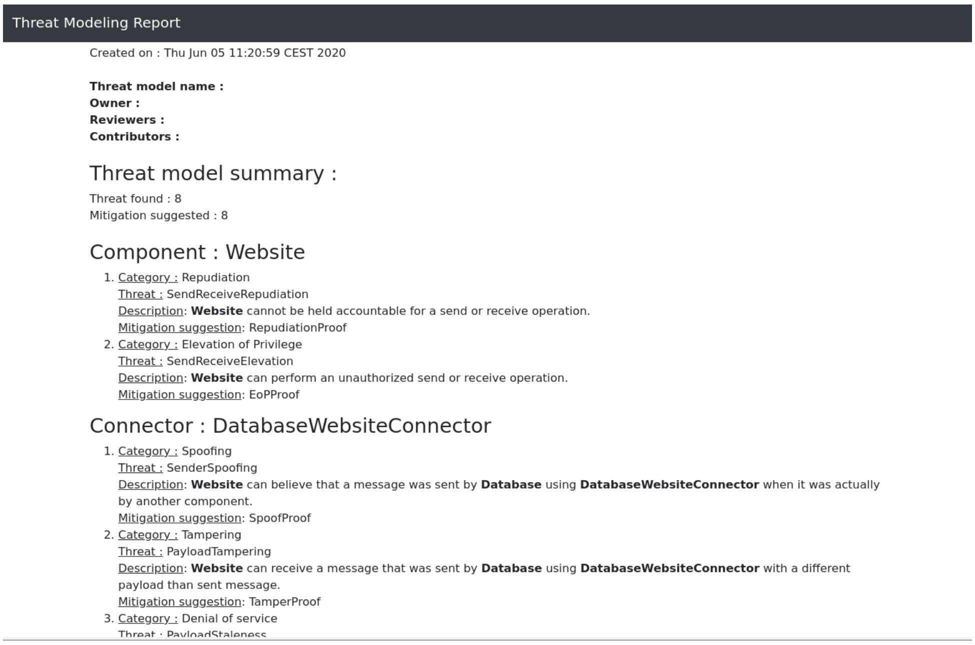
to the DSL model reusing the DSL model libraries and the *mitigation* relationship between properties (*A1.4*). The resulting DSL model is then transformed to a formal Alloy model (*A2.3*), where the formal definition of the corresponding security policy is automatically added in the produced Alloy software architecture model from their DSL definitions. At the end of mitigation (i.e., when no counterexample was found when the Alloy Analyzer is invoked), the security policies report is generated as an HTML document (*A2.8*), as visualized in Figure 5.9.

To support these eight activities, we developed a textual editor to model the architecture of the system using EMF and Xtext. In addition, the textual editor provides auto-completion for the manual integration of the threat library (*A2.2*), proposing the possible threat categories and properties for a software architecture elements, when the designer is typing the elements. We also developed two transformation engines to generate the formal Alloy model (*A2.3*) and the HTML reports using Xtend (*A2.5 and A2.8*). The transformation process takes advantage of the template feature provided by Xtend and consists of a set of transformation rules that are applied for each concept (using tree traversal) on a DSL model to generate a Alloy formal model. Figure 5.7 shows an example of such a rules for DSL Model to generate a corresponding Alloy formal model. The generation process allows to automatically incorporate the formal definition of the targeted threats from their DSL definition in the produced Alloy software architecture model as described in Listing 7.3. Finally, the Alloy Analyzer is used to verify the resulting models and to generate counterexamples.


workspace - fr.irit.cbal/src/fr/irit/cbal/generator/AlloyGenerator.xtend - Eclipse IDE
File Edit Project management Navigate Search Project Run Window Help
Quick Access
Project Explorer
abz18
alloyApi
AlloyMetaModel
AlloyMetaModelCBSE
Archi
ComponentConnector
ConceptRelationships
EBIOS
EBIOS_SAGHE
ECSA
FGCS19
fr.irit.cbal
src
fr.irit.cbal
fr.irit.cbal.alloychecker
fr.irit.cbal.alloychecker.objective
fr.irit.cbal.alloychecker.policy
fr.irit.cbal.als.uri
fr.irit.cbal.generator
AlloyGenerator.xtend
CoqGenerator.xtend
DslGenerator.xtend
fr.irit.cbal.scoping
Outline
compile(Interface) : CharSequence
compile(DataType) : CharSequence
compile(Method) : CharSequence
compile(FunctionnalRequierement) : CharSeque
compile(MPS) : CharSequence
compile(ObjectiveVerification) : CharSequence
compile(ObjectiveConnector) : CharSequence
getObjPredName(String) : CharSequence
compile(ThreatVerification) : CharSequence
compile(ThreatConnector) : CharSequence
compile(ThreatComponent) : CharSequence
compile(Policy) : CharSequence
getThreatPredName(String) : CharSequence
getPolicyPredName(String) : String
fr.irit.iceccs.cbal
Dsl.xtext
AlloyGenerator.xtend

```
    def compile(ThreatVerification f) '''
        «FOR e:f.elements»
            «IF e instanceof ThreatConnector »
                «e.compile»
            «ENDIF»
            «IF e instanceof ThreatComponent »
                «e.compile»
            «ENDIF»
        «ENDFOR»
    '''
    def compile(ThreatConnector t) '''
        «FOR p:t.reqs »
        assert threat_connector_« t.con.name »_« t.c1.name »_« t.c2.name »_« t.t.name »_« p.getName() » {
                            no «p.getName().getThreatPredName()».holds[« t.c1.name », « t.c2.name », Payload« t.t.name »]
        }
        check threat_connector_« t.con.name »_« t.c1.name »_« t.c2.name »_« t.t.name »_« p.getName() » for 4 but 5 Tick

        «ENDFOR»
    '''
    def compile(ThreatComponent t) '''
        «FOR p:t.reqs »
                assert threat_component_« t.c.name »_« t.t.name »_« p.getName() » {
                    no «p.getName().getThreatPredName()».holds[« t.c.name », Payload« t.t.name »]
                }
                check threat_component_« t.c.name »_« t.t.name »_« p.getName() »  for 4 but 5 Tick

        «ENDFOR»
    '''

    def compile(Policy s) '''
        «FOR e:s.elements»
            «IF e instanceof PolicyConnector »
            «FOR p:e.reqs »
            fact secreq_connector_«p.getName().getPolicyPredName()»_« e.c.name » {
                « e.c.name ».«p.getName().getPolicyPredName()»
            }
            «ENDFOR»
            «ENDIF»
            «IF e instanceof PolicyComponent »
            «FOR p:e.reqs »
            fact secreq_component_«p.getName().getPolicyPredName()»_« e.c.name » {
                « e.c.name ».«p.getName().getPolicyPredName()»
            }
            «ENDFOR»
            «ENDIF»
        «ENDFOR»
    '''
```

Figure 5.7: Transformations supporting the generation of an Alloy for threats from a DSL model using Xtend



Threat Modeling Report

Created on : Thu Jun 05 11:20:59 CEST 2020

Threat model name :
Owner :
Reviewers :
Contributors :

Threat model summary :

Threat found : 8
Mitigation suggested : 8

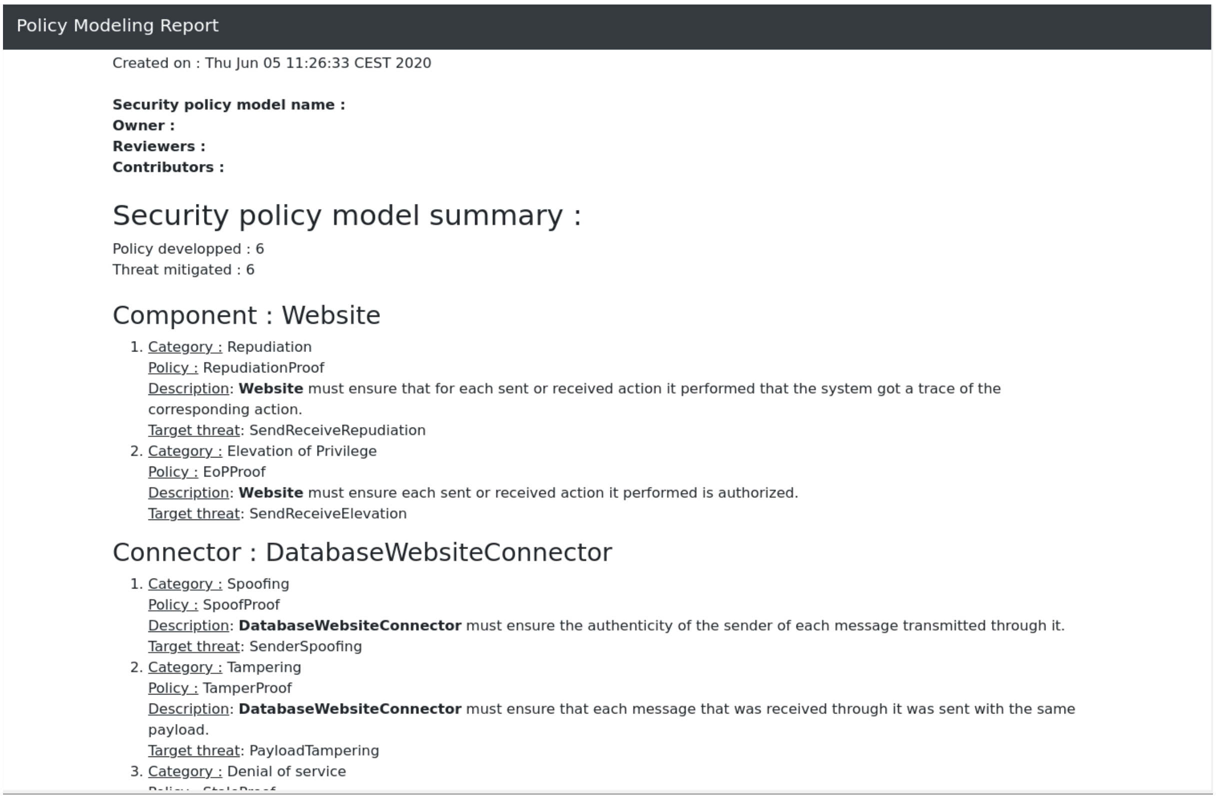
Component : Website

1. Category : Repudiation
   Threat : SendReceiveRepudiation
   Description: **Website** cannot be held accountable for a send or receive operation.
   Mitigation suggestion: RepudiationProof
2. Category : Elevation of Privilege
   Threat : SendReceiveElevation
   Description: **Website** can perform an unauthorized send or receive operation.
   Mitigation suggestion: EoPProof

Connector : DatabaseWebsiteConnector

1. Category : Spoofing
   Threat : SenderSpoofing
   Description: **Website** can believe that a message was sent by **Database** using **DatabaseWebsiteConnector** when it was actually by another component.
   Mitigation suggestion: SpoofProof
2. Category : Tampering
   Threat : PayloadTampering
   Description: **Website** can receive a message that was sent by **Database** using **DatabaseWebsiteConnector** with a different payload than sent message.
   Mitigation suggestion: TamperProof
3. Category : Denial of service
   Threat : PayloadStaleness

Figure 5.8: Threat report showing the identified threats for the college library web application



Policy Modeling Report

Created on : Thu Jun 05 11:26:33 CEST 2020

Security policy model name :
Owner :
Reviewers :
Contributors :

Security policy model summary :

Policy developped : 6
Threat mitigated : 6

Component : Website

1. Category : Repudiation
   Policy : RepudiationProof
   Description: **Website** must ensure that for each sent or received action it performed that the system got a trace of the corresponding action.
   Target threat: SendReceiveRepudiation
2. Category : Elevation of Privilege
   Policy : EoPProof
   Description: **Website** must ensure each sent or received action it performed is authorized.
   Target threat: SendReceiveElevation

Connector : DatabaseWebsiteConnector

1. Category : Spoofing
   Policy : SpoofProof
   Description: **DatabaseWebsiteConnector** must ensure the authenticity of the sender of each message transmitted through it.
   Target threat: SenderSpoofing
2. Category : Tampering
   Policy : TamperProof
   Description: **DatabaseWebsiteConnector** must ensure that each message that was received through it was sent with the same payload.
   Target threat: PayloadTampering
3. Category : Denial of service
   Policy : StaleProof

Figure 5.9: Security policy report showing the suggested security requirements to mitigate the identified security threats for the college library web application (Figure 5.8)

Putting all this together and using the developed tooled framework, we are able to: (1) model security requirements from the negative vision using the STRIDE security threats at the architecture design level using a domain-specific language, (2) generate an interpretation of the resulted software architecture model and properties in Alloy, (3) verify the satisfaction of a set of security requirements for a concrete application through reusable model of previously specified and verified security threat, (4) treat the detected threats using appropriate policy models, and (5) generate new artifacts as feedback on the software architecture model and its security.

5.8 Conclusion

We proposed an approach to threat specification, detection, and treatment in component-based software architecture models following two levels of specification: (1) logical specification of threats using first-order and modal logic as a technology-independent formalism; (2) interpretation of the logical specification in Alloy as a tooled language and (3) for each representative STRIDE security property, an appropriate security policy to treat the threat.

We keep the domain specifying threat categories at an abstract level to remain independent of any particular technology or implementation of the model aligned with the proposed approach. This can allow these formal specifications to apply to different implementations or architectures provided they support the notion of message-passing communication which fosters reuse. As a side effect, for each representative STRIDE threat, an appropriate property codifying the security policy protecting against the threat is defined and verified, where the violation of such a property indicates the presence of threat. The proposed approach can aid in developing reliable software systems. For example, it can help system designers to rework their designs to eliminate or mitigate the identified threats and/or to aid in selecting appropriate security and reliability controls.

Furthermore, we walked through an MDE-based prototype connected to a tooled formal language (Alloy) to support the proposed approach. An example of this tool suite is constructed using EMFT, Xtext, and Xtend and is currently provided in the form of Eclipse plug-ins. The combined formal modeling and MDE to specify threats and develop their targeted system security requirements allows to develop an accurate analysis, for evaluation and/or certification.

Chapter 6

Security objectives

Contents

6.1 Introduction

The current practice to express security statements from the positive perspective is given in the form of a set of desirable security properties using common taxonomies such as CIAA. Security solutions are then being introduced according to expected security properties.

In this chapter, we propose to use formal methods for the precise specification and analysis of security architecture requirements as properties of a modeled system. Starting from an informal description of a security objective (e.g., from a standard) in the context of component-based software architecture development, a logical specification of these objectives is proposed using an abstract system computing model followed by a more concrete specification of the system computing model and the objectives. Finally, a set of security policies are defined as abstract security mechanisms to be enforced within the system specification. They are represented in the form of assumptions of a modeled system to constrain its operation and to guarantee the corresponding security objectives. With regard to our contributions, we deal with ***C1***,***C2.2***, ***C3.2***,***C4.2***, ***C5.3*** and ***C5.4*** related to the Research Objective 1 (***RO1***), the security objectives concerns from ***C6*** and ***C7*** related to the Research Objective 2 (***RO2***) and the security objectives concerns from ***C8*** and ***C9*** related to the Research Objective 3 (***RO3***).

To evaluate our approach, we study a set of representative security objectives based on the CIAA quartet classification targeting the components and the communication links in component-port-connector architecture views [25]. We use MDE abstraction mechanisms to define and handle software architecture model, security objectives and policies through a meta-model that unifies those concepts. Moreover, we use MDE transformation mechanisms that can adapt and generate different artifacts and representations. Eclipse Modeling Framework Technology (EMFT) is used to build the support tools for our approach.

The remainder of the chapter is organized as follows. Section 6.2 discusses related work. Then, Section 6.3 describes the methodology to build a design and analysis framework for security architecture objectives. Section 6.4 presents how our approach could be integrated to existing system development life cycle. Section 6.5 describes the formalization of security objectives using first-order and modal logic following the CIAA classification. Then, Section 6.6 presents the interpretation of the property metamodel and the logical specification of the CIAA security objectives in Alloy. Section 6.7 presents an additional interpretation of the system computing model and a formalization and analysis of an example of confidentiality objective in Coq. It also includes a set of policies as architectural solutions. Section 6.8 describes the architecture of the tool suite. Finally, Section 6.9 concludes.

6.2 Related work

In system and software engineering, formal methods are used for the precise specification of the modeling artifacts across the development life cycle for validation purposes [112], particularly in the development of security-critical systems [71, 27]. Regarding the verification of security properties, early work discusses the verification of cryptographic protocols and is based on an abstract (term-based) representation of cryptographic primitives that can be automatically verified using model checking and theorem proving tools. One research line in this category is authentication logics, the first of these logics being the BAN Logic [20]. The Inductive Approach by Paulson [107] started another research line. The Security Modeling Framework (SeMF) developed by Fraunhofer SIT [34] rigorously defines security properties in terms of sequences of actions that are performed by a set of agents (e.g., client, server). Recently, in [45] the authors used SeMF for the precise specification of security patterns, starting from a pattern description language and providing a validation mechanism to enable the verification of security properties.

More relevant to our topic, works from authors in [76] used a logic-based representation for describing abstract security properties which were implemented and verified using Coq [15]. The ANR SELKIS project [72] offers a method for analyzing and designing secure Information Systems (IS) by addressing security requirements from the earliest abstraction stages of the development. The method is compliant with MDE and takes advantage of existing formal languages, such as B, enabling the use of formal verification techniques to ensure, on the one hand, the consistency and adequacy of security policies with respect to functional business specifications and, on the other hand, the consistency of the implementation with respect to the specification.

There has been a renewed interest in how to support the Twin Peaks model [9] in a wide range of aspects, such as theoretical frameworks for relating requirements and architecture, tools and techniques such as goal-oriented inference and uncertainty management, problem frames and service composition. There has also been approaches for applying the Twin Peaks model in the context of security [52]. Moreover, [50] used Software Cost Reduction (SCR) tables to specify and analyze security properties codifying system requirements.

A forefront approach is the work of [64] which defines a UML profile that authorizes the expression of information related to security using UML diagrams. The UMLsec profile has been established using three mechanisms of UML extensions which are stereotypes, tagged values attached to the stereotypes and constraints. These mechanisms define generic security properties (Confidentiality, Integrity, Data flow security, Access control, Auditability and Traceability). These properties are verified during the analysis of the design against an adversary model.

To the best of our knowledge, none of the above-described approaches is able to integrate the security solution validation into the MDE refinement process of the application. In contrast to other formal security engineering methods [71, 107, 27], our work is not following the attack nor the threat-based approaches. Its basis is a set of desired security properties and associated assumptions. With our approach it is possible to validate if properties like authenticity or confidentiality hold under given assumptions that have been represented basically by appropriate security policies. Our added value is that the analysis does not only check if the needed security policies exist, but also that they are correctly used to fulfill the security requirements. The side benefit in case a stated assumption does not hold is that possible consequences in regard to security properties can be estimated. The verification is conducted mostly within Alloy and the resulted verification artifacts will be utilized by the designer as input to the system development process.

6.3 Methodology for the creation of a design and analysis framework

In this section, we present an application of the methodology proposed in Section 3.3 to the development of secure software architecture, from the positive perspective (i.e., objectives). Particularly, we consider architecture security objectives modeling in the context of component-port-connector architectures and message passing communication. As depicted in Figure 6.1, the methodology is composed of several phases and activities. It supports the development of verified and reusable model libraries to represent security objectives violation and treatment for specifying secure software architectures of distributed systems.

Conceptual modeling. We begin by updating the *Property* package used for security threats introduced in the previous chapter, to capture security objectives (problem and solution) also as properties of a modeled system. Then, using the logical specification introduced in the context of threat modeling, we defined classes of security objectives based on the CIAA reference. In the context of this work, the security objectives are specified in terms of a set of desirable security properties, where security requirements as then introduced according to expected security objectives.

Development. After the update of the property meta-model and defining the security objectives semantics upon the communication model (message passing connector), we developed (1) a new DSL to model properties from the positive perspective and (2) a formal modeling environment as an interpretation of the meta-model and the logical specification of the CIAA security objectives using Alloy. We also developed an interpretation of the system computing model and an example of a confidentiality objective in Coq. Moreover,

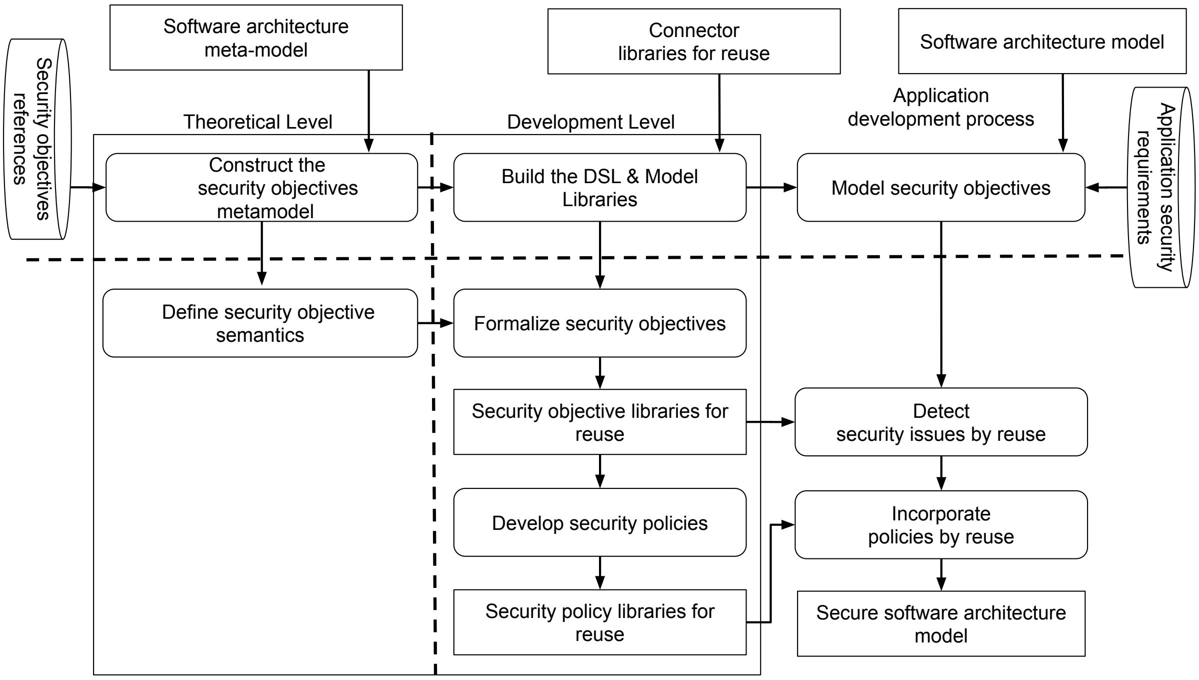

Software architecture meta-model
Connector libraries for reuse
Software architecture model
Security objectives references
Theoretical Level
Development Level
Application development process
Application security requirements
Construct the security objectives metamodel
Build the DSL & Model Libraries
Model security objectives
Define security objective semantics
Formalize security objectives
Security objective libraries for reuse
Detect security issues by reuse
Develop security policies
Incorporate policies by reuse
Security policy libraries for reuse
Secure software architecture model

Figure 6.1: The proposed approach development process for security objectives

we provided a set formal model library for reuse to specify the security objectives and policies. By doing so, we obtain a formal specification of some representative security objectives for each CIAA objective category. This gives us a set of reusable security objectives libraries (properties) capable of identifying the violation of the desired security objective in a concrete software architecture model and for developing specifications of security solutions (policies) to satisfy them. The result is a set of abstract formal security solution modules that can be easily reused. This formalization step is required to leverage available tool support used in software and system modeling and to enable straightforward instantiation and model checking capabilities to support the elicitation of an appropriate set of security policies to treat any detected security issues.

6.4 Supporting security-by-design within the SDLC

The overall approach can easily fit well into various systems development life cycles (SDLCs) as a supplement to the requirements specification and architecture design phases, as shown in the right part of Figure 6.1. Starting from the software architecture model obtained from the approach presented in Section 4.3, we use the security objectives libraries to identify potential security issues in the system model by checking that it satisfies the desired properties. If the model does not satisfy the properties, we use our developed libraries to suggest new security policy to treat the identified security issues.

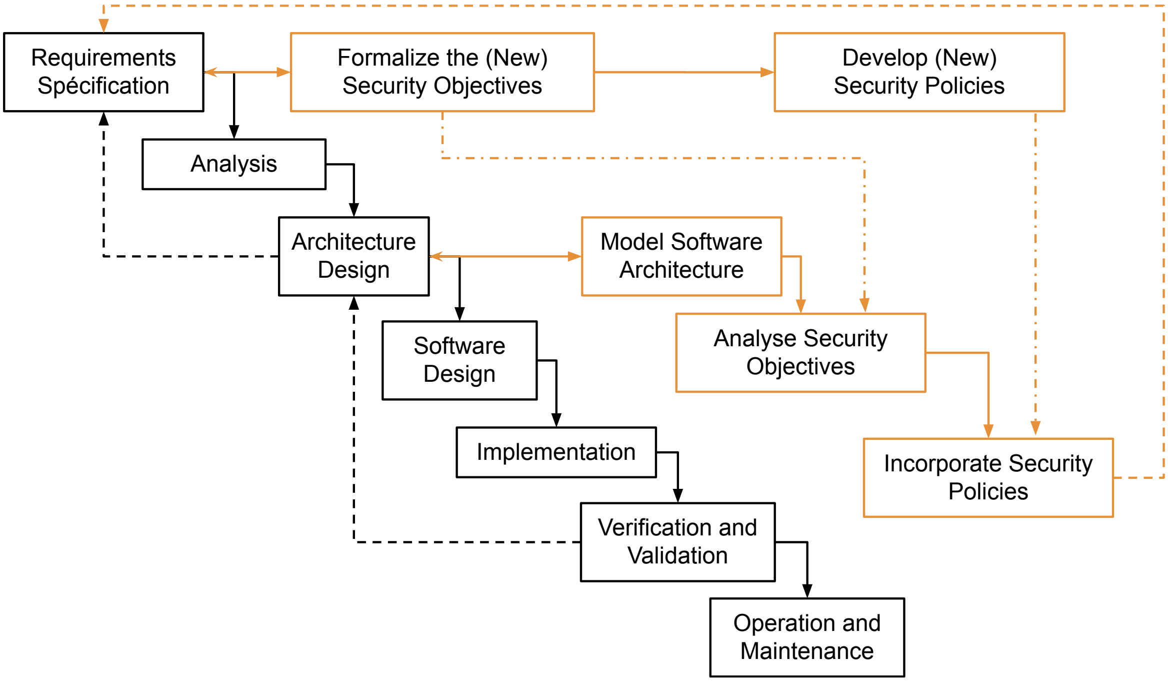

Requirements Spécification
Formalize the (New) Security Objectives
Develop (New) Security Policies
Analysis
Architecture Design
Model Software Architecture
Software Design
Analyse Security Objectives
Implementation
Incorporate Security Policies
Verification and Validation
Operation and Maintenance

Figure 6.2: The proposed approach within the Royce iterative waterfall SDLC

For simplicity, as shown in Figure 6.2, we illustrate the application of the approach within the Royce iterative waterfall SDLC [118]. The activities defined in Section 6.3 come as a supplement to the existing phase as follows: (1) The requirements specification is extended with the "Formalize the (New) Security Objectives" and "Develop (New) Security Policies" activities. These activities concern the modeling framework development process and should only be done if a new security objective or policy is needed to be added in the framework, i.e., if a needed objective or policy is not already defined in the framework libraries. (2) The architecture design is extended with the "Model Software Architecture" and "Analyse Security Objectives" 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 of the system, suggesting new security policies in the "Incorporate Security Policies" activity to satisfy the desired security objectives. 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 and fulfill the security objectives. As a result, the proposed approach enables the generation of formal artifacts early enough in the lifecycle to apply useful analysis within the design loop, thereby supporting the principles of security-by design.

6.5 Property view

Given the component-port-connector architectural model, we explore the formalization of representative security objectives based on the CIAA classification that target the communication links and components in such architectures.

We use the property meta-model presented in the beginning of Section 5.5 to describe the security objectives and policies as categories to build property model libraries for reuse, as visualized in Figure 6.3. For instance, *Confidentiality* and *Integrity* are defined as categories within the CIAA library. In addition, we define *satisfy* as a link between properties to capture relationships between objectives and policies.

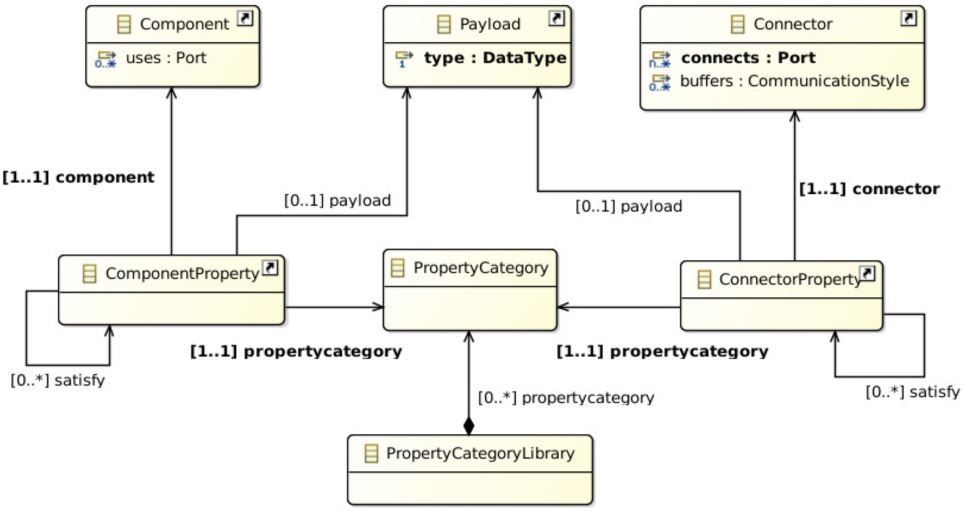


Figure 6.3: Property meta-model with *satisfy* relationships

In the following section, we present the precise definition of a set of security objectives using first-order logic and modal logic as a formalism that is abstract and technology independent in order to provide a more generic and understandable approach with mapping support to different existing properties languages used in modeling and software development.

6.5.1 Logical specification

We use the logical specification domains introduced to study threats (see Section 5.6.1).

6.5.2 CIAA security objectives

In each of the following paragraphs, we specify a representative property for each CIAA category.

Confidentiality. According to the ISO/IEC 27000:2018 standard [56], *confidentiality* denotes the property that information is not made available or disclosed to unauthorized individuals, entities, or processes. In the context of message passing communications within software architectures, confidentiality allows that the transmitted message can be obtained by other components (i.e., components can receive it) but only allowed components can get the actual content of the payload. In other words, if a component c_3 other than c_1 and c_2 is able to get the payload d of a message m, then c_1 didn't send the message m with c_2 as intended receiver before that. For components $c_1, c_2 \in \mathcal{C}$, we denote this representative property as *PayloadConfidentiality*(c_1, c_2) which is specified for all messages $m \in \mathcal{M}$, for all payloads $d \in \mathcal{D}$ and all components $c3 \in \mathcal{C}$ such that $c3 \neq c1 \neq c2$ as:

$$\mathbb{E}_{c_3}(\text{get\_pld}(m,d)) \Rightarrow \neg(\mathbb{H}_{c_1}(\text{inject}(m) \ \wedge \ \text{has\_rcv}(m,c_2)) \ < \ \mathbb{E}_{c_3}(\text{get\_pld}(m,d)) \quad (6.1)$$

Example of Confidentiality instantiation. In the context of the college library website example, the *Payload Confidentiality* objective must be satisfied, for example, when the web server makes a request using a message m to the database, then only the database should be able to get the payload of this request. Therefore, the validity of the property *PayloadConfidentiality*(*Webserver*, *Database*, m) would show that the *Payload Confidentiality* objective is fulfilled in the system.

Integrity. According to the ISO/IEC 27000:2018 standard [56], *integrity* refers to the accuracy and completeness of the information. In the context of message passing communications within software architectures, integrity means that no message payload can be altered. Therefore, an integrity property can be identified by verifying whether the system ensures that every received message is authentic, i.e., every message intercepted by an authorized receiver provides the accurate payload to this receiver. In other words, for any message m and payload d, if a component c_2 is able to get the payload d of message m and m was sent by c_1, then d must be the original payload of m. For components $s, r \in \mathcal{C}$, we denote this representative property as *PayloadIntegrity*(c_1, c_2) which is specified for all messages $m \in \mathcal{M}$ and for all payloads $d \in \mathcal{D}$ as:

$$\mathbb{H}_{c_1}(inject(m)) \; < \; \mathbb{E}_{c_2}(get\_pld(m,d)) \Rightarrow \; \mathbb{H}_{c_1}(set\_pld(m,d)) \; < \; \mathbb{H}_{c_1}(inject(m)) \quad (6.2)$$

Example of Integrity instantiation.

In the context of the college library website example, the *Payload Integrity* objective must be satisfied, for example, when the web server makes a request using a message m to the database, then this request should be accurate. As a result, the validity of the property *PayloadIntegrity*(*Webserver*, *Database*, m) would show that the *Payload Integrity* objective is fulfilled in the system.

Availability. According to the ISO/IEC 27000:2018 standard [56], *availability* denotes the property of a system function of being accessible and usable on demand by an authorized entity. In the context of message passing communications within software architectures, every sent message is received by the intended receiver. Therefore, an availability property can be identified by verifying whether the system ensures that all messages are not destroyed or deleted in transit, and they are eventually received by the intended receiver. In other words, for any message m, if a component c_1 injects the message m with c_2 as the intended receiver, then finally c_2 will be able to intercept m. For components $s, r \in \mathcal{C}$, we denote this representative property as *MessageAvailability*(c_1, c_2) which is specified for all messages $m \in \mathcal{M}$ as:

$$\mathbb{H}_{c_2}(inject(m)) \; \wedge \; has\_rcv(m, c_2) \; \rightsquigarrow \; \mathbb{E}_{c_2}(intercept(m)) \quad (6.3)$$

Example of Availability instantiation. Coming back to our motivating college library website example, the *Message Availability* objective must be satisfied, for example, when the web server makes a request using a message m to the database, then this request should be always reach the database. Consequently, the validity of the property *MessageAvailability* (*Webserver*, *Database*, m) would show that the *Payload Authenticity* objective is fulfilled in the system.

Authenticity. According to the ISO/IEC 27000:2018 standard [56], *authenticity* denotes the property that an entity is what it claims to be. In the context of message passing communications within software architectures, a receiver of a message is able to identify the true sender of the corresponding payload of this message. Therefore, an authenticity property can be identified by verifying whether the system ensures that all of the senders of message are authentic, i.e., the sender of a message is always the originator of the message. 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_2 is the accurate sender of m. For components $s, r \in \mathcal{C}$, we denote this representative property as $\mathit{MessageAuthenticity}(s, r, m)$ which is specified for all messages $m \in \mathcal{M}$ as:

$$\mathbb{H}_{c_1}(\mathit{inject}(m)) \; < \; \mathbb{E}_{c_2}(\mathit{get\_src}(m, c_1)) \tag{6.4}$$

Example of Authenticity instantiation. With respect to our motivating college library website example, the *Message Authenticity* objective must be satisfied, for example, when the database receives request using a message m from the web server, then the database should be sure it is in fact the web server that sent this request. Therefore, the validity of the property $\mathit{MessageAuthenticity}(\mathit{Webserver}, \mathit{Database}, m)$ would show that the *Payload Authenticity* objective is fulfilled in the system.

At this level of specification, we are able to specify the CIAA set of security objectives as properties of the system architecture model, using first-order and modal logic as an abstract and technology-independent formalism that can be interpreted in appropriate formal tooled languages.

6.6 Formal specification and analysis in Alloy

In this section, we provide the specification and verification of a representative security objective for each CIAA security objective category using *Alloy*. By operating as a counterexample generator, when the Alloy Analyzer identifies the violation of a property, it indicates the non-fulfillment of the objective. In much the same way, by operating as a model checker, the Alloy Analyzer enables the use of a property as a policy to indicate the fulfillment of the security objective. This enables us to construct models incrementally, allowing rapid iterations between modeling and analysis when writing a specification. For more formal definitions and examples on Alloy, the reader is referred to Section 2.6.2.

6.6.1 Formalizing the positive perspective of the property metamodel

The encoding in Alloy of the logical specification of the security objectives follows the one used for threats (see Section 5.6.1). As we shall see, security objectives and policies will be encoded as properties (see Listing 6.1).

6.6.2 CIAA security objectives

We aim to build a set of reusable libraries for security objectives verification and treatment. To do so, we model the objective categories presented in Section 6.5 in Alloy, using constructions such as *Predicate*, *Assertion*, and *Fact*. Remember that during the logical specification, each security objective category is associated with a representative property such that the violation of the specified property indicates the not satisfaction of the objective. Therefore, each security objective is associated with a security property defined as a predicate to map the logical definition of the corresponding security property to the Alloy model describing the targeted software architecture and communication system (e.g., *confidentialityNotHold*). Then, the violation of a security objective, as a result of the violation of the property, is detected by the Alloy Analyzer through an assertion finding a counterexample. As a result, an appropriate security policy is defined as a predicate (e.g., *restrictiveGetPld*) to codify a security mechanism to constrain the operation of the system and to guarantee the satisfaction of corresponding security objectives.

Confidentiality. The *PayloadConfidentiality* property is considered within the connector and can be identified by checking whether the communication system provided by the connector ensures that all information transmitted through this connector is delivered only to the intended receiver(s). The PayloadConfidentiality property is defined in this respect as a predicate according to the logical specification of the PayloadConfidentiality (see Equation (6.5.2)) and the system computing model.

```
sig PayloadConfidentiality extends ConnectorProperty  {}{
  payloadConfidentiality[comp1,comp2,payl]
}
pred payloadConfidentiality[c1,c2:Component, d:Payload] {
  all c3:Component-c1-c2 | let m = {m:MsgPassing | m.payload = d } |
  all t:Tick | E_get_pld[c3,m,d,t] implies
    (no t2:Tick-tick/first | no t1:t2.prevs | (H_inject[c1,m,t1] and m.receiver = c2)
       implies E_get_pld[c3,m,d,t2])
}
assert confidentialityNotHold {
  all c1,c2: Component, m:MsgPassing |  payloadConfidentiality[c1,c2,m]
}
```

Listing 6.1: Confidentiality property

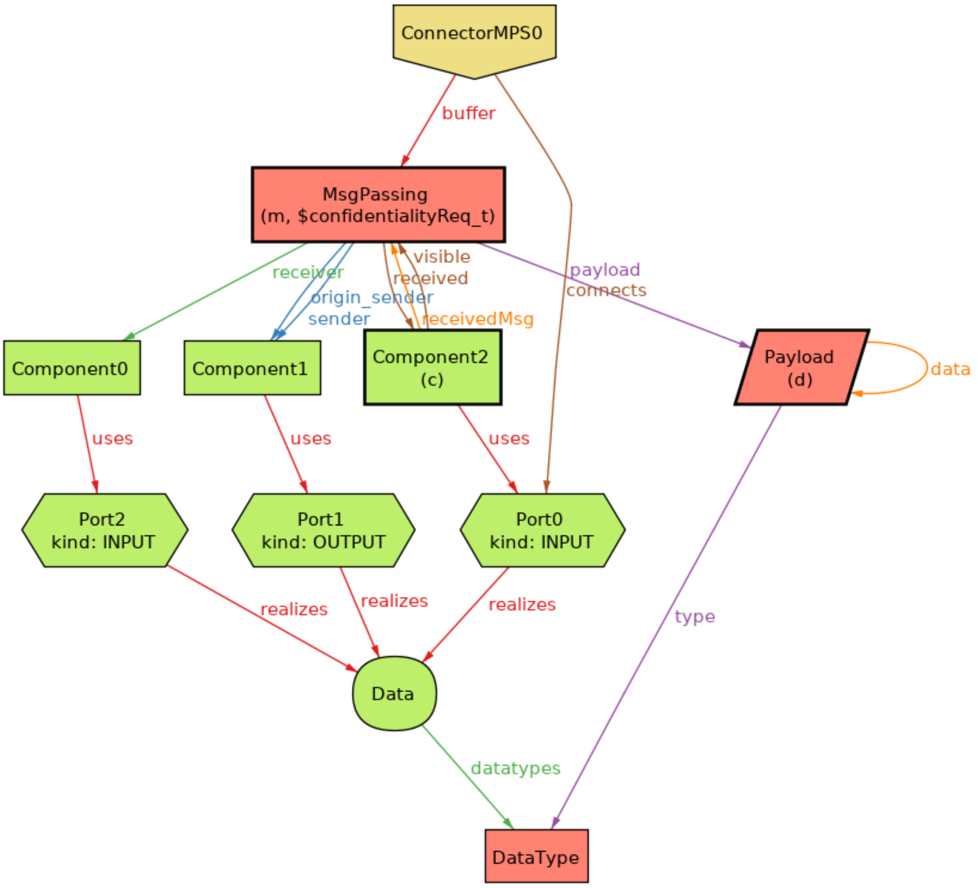


Figure 6.4: Confidentiality counterexample provided by the Alloy Analyzer

The Alloy Analyzer detects the violation of the confidentiality property by finding a counterexample executing the *confidentialityNotHold* assertion (see Figure 6.4). Then, we proceed by defining the security policy named *restrictiveGetPld* that constrains the *inject* and *get_pld* operations using the *AllowedGetPld*. It is as an abstract security mechanism to satisfy the confidentiality property by identifying the set of components that are able to get the payload. To evaluate the impact on using the defined policy, we introduce the *RestrictiveGetPld* property as a predicate on the connector level that ensures that all messages going through it respects this policy. It guarantees that no component other than the intended receiver(s) is able to get the payload of a message sent through this connector.

```
sig AllowedGetPld {
  msg: one MsgPassing,
  comp: set Component
}
```

```
pred restrictiveGetPld[m:MsgPassing] {
  all c:Component, t:Tick |
    E_inject[c,m,t] implies some al:AllowedGetPld |
        al.msg = m and  al.comp = m.receiver
  all c:Component, t:Tick, d:Payload |
    E_get_pld[c,m,d, t] implies some al:AllowedGetPld |
        al.msg = m and c in al.comp
}
pred ConnectorMPS.RestrictiveGetPld {
  all m:MsgPassing, t:Tick |
    m in this.buffer.t implies restrictiveGetPld[m]
}
assert confidentialityHold {
   (all c:ConnectorMPS | c.RestrictiveGetPld ) implies
      all c1,c2:Component, d:Payload | payloadConfidentiality[c1,c2,d]
}
```

Listing 6.2: Confidentiality policy

Integrity. The *PayloadIntegrity* property is considered within the connector and can be identified by checking whether the communication system provided by the connector ensures that all messages transmitted through this connector are not altered in transit. The PayloadIntegrity property is defined in this respect as a predicate according to the logical specification of the PayloadIntegrity property (see Equation (6.5.2)) and the system computing model.

```
sig PayloadIntegrity extends ConnectorProperty  {}{
  payloadIntegrity[comp1,comp2,payl]
}
pred payloadIntegrity[c1,c2:Component, d:Payload]  {
   let m = {m:MsgPassing | m.payload = d } |
    all t2:Tick-tick/first | some t1:t2.prevs |
     (H_inject[c1,m,t1] implies  E_get_pld[c2,m,d,t2]))
     implies (all t2:Tick-tick/first | some t1:t2.prevs |
        (set_pld[c1,m,d,t2] implies H_inject[c1,m,t1]))
}
assert integrityNotHold {
  all c1,c2: Component, d:Payload | payloadIntegrity[c1, c2, d] }
```

Listing 6.3: Integrity property

The Alloy Analyzer detects the violation of the integrity property by finding a counterexample. Then, we proceed by defining the security policy named *restrictiveSetPld* that constrains the *inject* and *set_pld* operations using the *AllowedSetPld*. It is an abstract security mechanism to satisfy the integrity property by defining the set of components who are able to write/modify the payload. The *RestrictiveSetPld* property is defined as a predicate on the connector that ensures that all messages going through it respects this policy. It guarantees that the payload that was sent by a sender is the same payload that was received by the receiver through this connector.

```
sig AllowedSetPld {
  msg: one MsgPassing,
  comp: one Component
}
pred restrictiveSetPld[m:MsgPassing] {
    all c:Component, t:Tick, d:Payload |
      E_set_pld[c,m,d, t] implies some al:AllowedSetPld |
          al.msg = m and al.comp = c
    all c:Component, t:Tick |
      E_inject[c,m,t] implies some al:AllowedSetPld |
          al.msg = m and al.comp = c
}
pred ConnectorMPS.RestrictiveSetPld {
  all m:MsgPassing, t:Tick |
    m in this.buffer.t implies restrictiveSetPld[m]
}
assert integrityHold  {
   (all c:Connector | c.RestrictiveSetPld) implies
      all c1,c2: Component, d:Payload |  payloadIntegrity[c1,c2,d]
}
```

Listing 6.4: Integrity policy

Availability. The *MessageAvailability* property is considered within the connector and can be identified by checking whether the communication system provided by the connector ensures that all information transmitted through this connector is not lost until the intended receiver(s) intercepts the message. The MessageAvailability property is defined in this respect as a predicate according to the logical specification of the MessageAvailability (see Equation (6.5.2)) and the system computing model.

```
sig MessageAvailability extends ConnectorProperty  {}{
  messageAvailability[comp1,comp2,payl]
}
pred messageAvailability [c1,c2:Component, d:Payload] {
  let m = {m:MsgPassing | m.payload = d } |
  all t1:Tick | some t2:t1.nexts |
      H_inject[c1,m,t1] and m.receiver = c2 implies E_intercept[c2,m,t2]
}
assert availabilityNotHold {
  all c1,c2: Component, d:Payload | messageAvailability[c1,c2, d]
}
```

Listing 6.5: Availability property

The Alloy Analyzer detects the violation of the Availability property by finding a counterexample.Then, we proceed by defining the security policy named *lossFreeInject* that constrains the operations on the buffer using the *AllowedIntercept*. It is an abstract security mechanism to satisfy this property by keeping every injected message in the buffer until an appropriate *intercept* operation is executed. The *LossFreeInject* property is defined as a predicate on the connector that ensures that all messages going through it respects this policy. It guarantees that no message is lost until the intended receiver(s) is able to intercept the message.

```
sig AllowedIntercept {
  msg: one MsgPassing,
  comp: set Component
}
pred lossFreeInject[m:MsgPassing] {
    all c1:Component, t1:Tick | let c2=m.receiver |
      inject[c1,m,t1] implies
        some al:AllowedIntercept, con:Connector | all t2:t1.nexts |
        m not in con.buffer.t2 iff al.msg = m and al.comp = c2
        and H_intercept[c2,m,t2]
}
pred ConnectorMPS.LossFreeInject {
  all m:MsgPassing, t:Tick |
    m in this.buffer.t implies lossFreeInject[m]
}
assert availabilityHold {
  (all c:Connector | c.LossFreeInject )  implies
      all c1,c2: Component, d:Payload | messageAvailability[c1,c2,d]
}
```

Listing 6.6: Availability policy

Authenticity. The *MessageAuthenticity* property is considered within the connector and can be identified by checking whether the communication system provided by the connector ensures that all messages transmitted through this connector have authentic senders. The MessageAuthenticity property is defined in this respect as a predicate according to the logical specification of the MessageAuthenticity property (see Equation (6.5.2)) and the system computing model.

```
sig MessageAuthenticity extends ConnectorProperty  {}{
  messageAuthenticity[comp1,comp2,payl]
}
pred messageAuthenticity[c1,c2:Component, d:Payload]  {
  let m = {m:MsgPassing | m.payload = d } |
  all t2:Tick-tick/first | some t1:t2.prevs |
      E_get_src[c2,m,c1,t2] implies H_inject[c1,m,t1]
}
assert authencityNotHold {
  all c1,c2: Component, d:Payload | messageAuthenticity[c1,c2,d]
}
```

Listing 6.7: Authenticity property

The Alloy Analyzer detects the violation of the authenticity property by finding a counterexample.Then, we proceed by defining the security policy named *restrictiveSetSrc* that constrains the *inject* and *set_src* operations using the *AllowedSetSrc*. It is an abstract security mechanism to satisfy the authenticity property by defining the set of components who are able to set up the payload. The *RestrictiveSetSrc* property is defined as a predicate on the connector that ensures that all messages going through it respects this policy. It guarantees that no component can pretend to send a message as some other component.

```
sig AllowedSetSrc {
  msg: one MsgPassing,
  comp: one Component
}
pred restrictiveSetSrc[m:MsgPassing] {
    all c,s:Component, t:Tick |
      E_set_src[c,m,s,t] implies some al:AllowedSetSrc |
          al.msg = m and al.comp = s
    all c:Component, t:Tick |
      E_inject[c,m,t] implies some al:AllowedSetSrc |
          al.msg = m and al.comp = c
}
pred ConnectorMPS.RestrictiveSetSrc {
  all m:MsgPassing, t:Tick |
    m in this.buffer.t implies restrictiveSetSrc[m] }
```

```
assert authencityHold {
  (all c:Connector | c.RestrictiveSetSrc ) implies
      all c1,c2: Component, d:Payload |  messageAuthenticity[c1,c2,d]
}
```

Listing 6.8: Authenticity policy

At this level of specification, we are able to: (1) formally represent an interpretation of the CIAA set of objectives described in Section 5.5.2 and develop a set of policies to treat them as properties of the architecture model in Alloy, (2) provide the properties as formal model libraries to support reuse, and (3) offer methods and functions to easily verify that the properties hold in a given architecture model fostering reuse.

6.7 Formal specification and analysis in Coq

Our goal is to use Coq as a proof assistant (1) to specify the security objectives as an interpretation of the logical definition of the system architecture (see Section 5.5.1), the security objectives and policies as properties (see Section 6.6); (2) the formulation of theorems stating relations between properties; and (3) to support proving these relations semi-automatically using the proof assistant.

In this section, we provide the specification and verification of a representative security objective for the confidentiality category using *Coq*. By operating as a proof assistant, when the Coq IDE checks the validity of a property as a policy to indicate the fulfillment of the security objective. For more formal definitions and examples on Coq, the reader is referred to Section 2.6.2 and Appendix B.1.

6.7.1 Formalizing the software component meta-model and properties meta-model

A software architecture metamodel as described in Section 4.5 and the set of actions described in Section 5.5.1 are mapped to our Coq metamodel as follows.

- Component $\mathcal{C}$, Message $\mathcal{M}$, Payload $\mathcal{D}$ are defined as *Type*.
- The set of actions $\mathcal{A}$ are expressed as inductive types, where each action act($param_1$, $\ldots, param_n$) $\in \mathcal{A}$ is transformed in constructor act : param_1 → ... → param_n → Action. For example, inject(m) is transformed to inject : Message → Action indicating that a sender adds a message $m \in \mathcal{M}$ into the system.

- Σ is the set of actions in $\mathcal{A}$ performed by the components in $\mathcal{C}$.
- The system's behavior can then be formally described by B, where $B \subseteq \Sigma^*$

Listing 6.9 depicts the interpretation of these concepts in Coq.

```
Parameter Component: Type.
Parameter Message: Type.
Parameter Payload : Type.
Parameter Pld: Message → Payload.
Parameter Src: Message → Component.
Parameter Rcv: Message → Component.

Inductive Action: Type :=
    inject :  Message → Action
  | intercept :  Message → Action
  | get_rcv :  Message → Component → Action
  | get_src :  Message → Component → Action
  | get_pld :  Message → Payload → Action
  | set_rcv :  Message → Component → Action
  | set_src :  Message → Component → Action
  | set_pld :  Message → Payload → Action.

Inductive CompoundAction : Type :=
    BasicAction (c:Component) (a : Action)
  | Any
  | UnionAction (a1 a2 : CompoundAction)
  | SequenceAction (a1 a2 : CompoundAction)
  | RepeatAction (a: CompoundAction)
  | SkipAction.

Definition Behavior := nat → (Component ∗ Action).
Notation "c 'does' a" := (BasicAction c a) (at level  23,  no associativity ).
```

Listing 6.9: Coq metamodel

Then, we define the atom and formula as inductive *Type*. The base logic connector (atomic formula, false, implies, until) are also defined. Listing 6.10 depicts the corresponding definitions and used notations.

```
Inductive Atom :=
| has_src  (m: Message) (c: Component)
| has_rcv  (m: Message) (c: Component)
| has_pld  (m: Message) (d: Payload)
| E (ag: Component) (a: Action).
```

```
Inductive Formula :=
| AtomicFormula (a:Atom)
| Is⊥
| Implies (f1: Formula) (f2: Formula)
| Until (f1: Formula) (A: CompoundAction) (f2: Formula).
Coercion AtomicFormula: Atom >→ Formula.
```

Listing 6.10: Coq atom & formula specification

Next, we specify the logical connectives and modal operators presented in Section 6.5 on top of the already defined formulas. Listing 6.11 depicts some examples of the definitions and notations in Coq.

```
Definition Not p := Implies p Is⊥.
Definition Is⊤ := Not Is⊥.
...
Definition Leadsto f1 A f2 := Globally Any (Implies f1 (Eventually A f2)).
Definition AllA (ca: CompoundAction) (a: Formula) := Leadsto Is⊤ ca a.
...
Definition H ag a := ExA (ag does a) Is⊤.
Definition Precede p q := Not (Until (Not p) Any q).
...
Notation ”p ⇒ q” := (Implies p q) (at level 14, right associativity ).
Notation ”p −→ q” := (Leadsto p Any q) (at level 49, right associativity).
Notation ”p << q” := (Precede p q) (at level 15, no associativity).
...
Notation ”! p” := (Not p) (at level 9, right associativity ).
```

Listing 6.11: Interpretation of logical connectives and modal operators in Coq

Then, we specify notations $\mathrm{B} \vdash \mathrm{f}$ that express the satisfaction of a formula f for a behavior B (See Listing 6.12).

```
Parameter asat: (Component ∗ Action) → Atom → Prop.
Parameter compat: Behavior → nat → CompoundAction → Prop.
Fixpoint sat B f : Prop :=
  match f with
   AtomicFormula a → asat (B 0) a
 | Is⊥ → ⊥
 | Implies f1 f2 → (sat B f1) → (sat B f2)
 | Until f1 A f2 → ∃ t,
    sat (fun i → B (t+i)) f2 ∧
    (∀ t’, t’ < t → sat (fun i → B (t’+i)) f1) ∧
    compat B t A
  end.
```

```

Notation ”B ⊢f” := (sat B f) (at level 50, no associativity ).
```

Listing 6.12: Coq behavior satisfaction

We encode the axioms described in Section 5.5.1 and specify lemmas for the previous defined operator of our logic (logical connectives, modal operators) to build logical reasoning. For example, we define Lemma *satImpl_elim* that shows if a behavior B satisfies p *implies* q, then B satisfies p and B satisfies q (see Listing 6.13).

```
Axiom axiomHImpE : ∀ B c1 p, B ⊢(H c1 p) ⇒ (E c1 p).
...
Lemma satImpl_elim: ∀ B p q, (B ⊢p ⇒ q) → ((B ⊢p) → (B ⊢q)).
Proof.
  intros.
  apply H0.
  apply H1.
Qed.
...
```

Listing 6.13: Coq Axioms & Lemmas

Security objectives can now be defined in terms of a system specification, i.e., in terms of actions, messages, components, etc.

6.7.2 Confidentiality in Coq

As already mentioned in Section 6.6.2, an example of a relation between properties is that restrictiveGetPld [m:MsgPassing] under certain assumptions implies PayloadConfidentiality [c1,c2:Component, d:Payload]. In this section we use the formal definitions provided in Section 6.6 to formally prove that under the assumption that the policy is an abstract security mechanism identifying components who are able to get the payload, restrictiveGetPld (p: AllowedGetPld) m implies PayloadConfidentiality B c1 c2. This is our first example of using Coq in the context of security-by design but serves to show how we use Coq in our work.

Confidentiality of the content of the payload. *PayloadConfidentiality* property for component $c1$ and $c2$ is defined in this respect as a predicate according to the logical specification of the *PayloadConfidentiality* (Equation 6.1) and the system computing model as introduced in the beginning of this section.

```
Definition PayloadConfidentiality B c1 c2 :=
  ∀ c3 m d, not (In c3 [c1; c2]) → (B ⊢Eventually Any (E c3 (get_pld m d))) →
    not (B ⊢(H c1 (inject  m) && has_rcv m c2) << (E c3 (get_pld m d))).
```

Listing 6.14: Coq PayloadConfidentiality property definition

Security policy. *restrictiveGetPld* property constrains the *inject* and *get_pld* operations using *AllowedGetPld*. It is as an abstract security mechanism to satisfy the confidentiality property by identifying components who are able to get the payload.

```
Record AllowedGetPld : Type := {
  msg: Message ;
  comp : Component;
}.

Definition restrictiveGetPld (p:AllowedGetPld) m B :=
  (∀  c, (B ⊢Eventually Any (E c (inject  m)) → (msg p = m ∧ Rcv m = comp p)))
    ∧
  (∀  c d, (B ⊢Eventually Any (E c (get_pld m d))) → (msg p = m ∧ c = comp p)).

Definition connectorRestritiveGetPld p B :=
  ∀ c  m, (B ⊢Eventually Any (E c (intercept m))) → restrictiveGetPld p m B.
```

Listing 6.15: Coq restrictiveGetPld policy definition

We can now proceed with the proof.

Proof goal. We want to prove that if the connector *con* between $c1$ and $c2$ enforces the *allowedGetPld* policy (hypothesis *connectorRestritiveGetPld*), then the confidentiality objective ($PayloadConfidentiality$) between $c1$ and $c2$ holds, i.e., no $c3$ component is able to get the payload of a message m send by $c1$ to $c2$ through connector *con*.

Proof overview. In order to show this, we use proof by contradiction. We assume that *connectorRestritiveGetPld* (the policy is deployed) and B ⊢(H c1 (inject m)&& has_rcv m c2)<< E c3 (get_pld m d) (the negation of the final implication of our proof goal $PayloadConfidentiality$) and we derive a contradiction. The proof consists of the following steps:

Step 1. We define the initial hypotheses for all B behavior; $c1$, $c2$, $c3$ components; p *allowedGetPld* policy; m a message and d a payload. Unfolding the definitions and specializing some of them, we obtain the following. The corresponding Coq code is depicted in Listing 6.16.

1. *connector_restritive_get_pld* (specialize for $c3$ and m):

$$\begin{aligned}
&\text{B} \vdash \text{Eventually Any (E c3 (intercept m))} \rightarrow \\
&\quad (\ \forall\ \text{c} : \text{Component, B} \vdash \text{Eventually Any (E c (inject m))} \\
&\qquad\quad \rightarrow \text{msg p} = \text{m} \wedge \text{Rcv m} = \text{comp p}) \wedge \\
&\quad (\ \forall\ (\text{c} : \text{Component})(\text{d} : \text{Data}), \text{B} \vdash \text{Eventually Any (E c (get\_pld m d))} \\
&\qquad\quad \rightarrow \text{msg p} = \text{m} \wedge \text{c} = \text{comp p})
\end{aligned}$$

2. *c3_not_in_c1_c2* (from *PayloadConfidentiality*) :

$$\neg\,\text{In c3 [c1; c2]}$$

3. *c3_eventually_enable_to_get_d* (from *PayloadConfidentiality*) :

$$\text{B} \vdash \text{Eventually Any (E c3 (get\_pld m d))}$$

4. *c1_injected_m_to_c2* (negation of the final implication of *PayloadConfidentiality* (proof by contradiction)) :

$$\text{B} \vdash \text{(H c1 (inject m) \&\& has\_rcv m c2)} << \text{E c3 (get\_pld m d)}$$

```
Theorem confidentialityHold:
  ∀ B c1 c2 p, connectorRestritiveGetPld p B → PayloadConfidentiality B c1 c2.
Proof.
  intros B c1 c2 p connector_restritive_get_pld .
  unfold PayloadConfidentiality, connectorRestritiveGetPld, restrictiveGetPld in ∗.
  intros c3 m d c3_not_in_c1_c2 c3_eventually_enable_to_get_d c1_injected_m_to_c2.
  specialize (connector_restritive_get_pld c3 m).
```

Listing 6.16: Coq confidentiality proof step 1

Step 2. In this step, we show that, as we know, the behavior B entails that eventually $c3$ is able to get the payload d of message m, then it means that a behavior B

entails that eventually $c3$ intercepted this message before. Starting from the hypothesis *c3_eventually_enable_to_get_d* :

$$\text{B} \vdash \text{Eventually Any (E c3 (get\_pld m d))}$$

We deduce *c3_eventually_intercept_m* :

$$\text{B} \vdash \text{Eventually Any (H c3 (intercept m))}$$

Therefore, we show that the behavior B satisfies that eventually $c3$ intercepted this message.

Step 3. We show that as the behavior B satisfies eventually $c3$ intercepted the message m, then it means that the behavior B satisfies eventually $c3$ was also able to intercept this message. Starting from the newly obtained hypothesis *c3_eventually_intercept_m* :

$$\text{B} \vdash \text{Eventually Any (H c3 (intercept m))}$$

We obtain the new hypothesis *c3_eventually_e_intercept_m* :

$$\text{B} \vdash \text{Eventually Any (E c3 (intercept m))}$$

Step 4. We apply the hypothesis from the *Step 1 connector_restritive_get_pld* in the new hypothesis *c3_eventually_e_intercept_m* obtained in the previous step in order to simplify it. From this we then deduce *restritive_get_pld*.

$$\begin{array}{l}
(\ \forall\ \text{c : Component,} \\
\quad \text{B} \vdash \text{Eventually Any (E c (inject m))} \\
\qquad \rightarrow \text{msg p} = \text{m} \wedge \text{Rcv m} = \text{comp p)} \\
\wedge \\
(\ \forall\ \text{(c : Component)(d : Data),} \\
\quad \text{B} \vdash \text{Eventually Any (E c (get\_pld m d))} \\
\qquad \rightarrow \text{msg p} = \text{m} \wedge \text{c} = \text{comp p)}
\end{array}$$

Step 5. In this step, we simplify again the hypothesis *restritive_get_pld* and show that $c3$ and m must be the same, respectively, as the component, denoted by *comp p*, and the message, denoted by *msg p*, of the policy p. The obtained terms are then substituted in the rest of the hypotheses.

$$
\begin{aligned}
&(\ \forall\ \text{c}\ :\ \text{Component},\\
&\text{B} \vdash \text{Eventually Any (E c (inject m))} \rightarrow \text{msg p} = \text{m} \wedge \text{Rcv m} = \text{comp p})\\
&\wedge\\
&(\forall\ (\text{c}\ :\ \text{Component})(\text{d} : \text{Data}),\\
&\text{B} \vdash \text{Eventually Any (E c (get\_pld m d))} \rightarrow \text{msg p} = \text{m} \wedge \text{c} = \text{comp p})
\end{aligned}
$$

We obtain the following two terms by simplification :

- msg p = m
- c3 = comp p

We substitute these terms in our hypothesis. From this we then deduce the following updated hypothesis:

- *c3_not_in_c1_c2* (from *Step 1*) :

$$\neg\, \text{In (comp p)[c1; c2]}$$

- *c3_eventually_intercept_m* (from *Step 2*) :

$$\text{B} \vdash \text{Eventually Any (H (comp p)(intercept (msg p)))}$$

- *c1_injected_m_to_c2* (from *Step 1*):

$$
\begin{aligned}
&\text{B} \vdash \text{(H c1 (inject (msg p)) \&\& has\_rcv (msg p)c2)} <<\\
&\quad \text{E (comp p)(get\_pld (msg p) d)}
\end{aligned}
$$

- *eventually_e_inj_m_impl_valid_policy* (from current Step) :

$$
\begin{gathered}
\text{B} \vdash \text{Eventually Any (E c1 (inject (msg p)))} \rightarrow\\
\text{msg p} = \text{msg p} \wedge \text{Rcv (msg p)} = \text{comp p}
\end{gathered}
$$

Step 6. We show that, since the behavior *B* satisfies that eventually the component denoted by *comp p* of the policy *p* is able to get the payload *d* from the message denoted by *msg p* of the policy *p*, the behavior *B* satisfies that eventually the component *comp p* is the one that intercepts the message *msg p*. Starting from hypothesis *c3_eventually_enable_to_get_d*:

$$\text{B} \vdash \text{Eventually Any (E (comp p)(get\_pld (msg p)d))}$$

We get *eventually_h_comp_p_intercept_msg_p* :

$$\text{B} \vdash \text{Eventually Any (H (comp p)(intercept (msg p)))}$$

Step 7. In this step, we show that since the behavior *B* satisfies that eventually the component *comp p* is the one that intercepts the message *msg p*, then the behavior *B* satisfies that eventually the component *c*1 injected the message *msg p*. Starting from hypothesis *eventually_h_comp_p_intercept_msg_p*:

$$\text{B} \vdash \text{Eventually Any (H (comp p)(intercept (msg p)))}$$

We obtain *ev_c1_inject_msgP*:

$$\text{B} \vdash \text{Eventually Any (H c1 (inject (msg p))}$$

Step 8. We show that since the behavior *B* satisfies that eventually *c*1 injected the message *msg p*, the behavior *B* also satisfies that eventually *c*1 was able to inject *msg p*. Starting from *ev_c1_inject_msgP*:

$$\text{B} \vdash \text{Eventually Any (H c1 (inject (msg p)))}$$

We show that we have *E_c1_inject_msgP*:

$$\text{B} \vdash \text{Eventually Any (E c1 (inject (msg p)))}$$

Step 9. In this step, we show that through some simplifications from the previous hypotheses, the component *comp p* declared in the policy *p* must be the receiver *Rcv* (*msg p*) of the message *msg p* declared in the policy *p*. Then, we rewrite the corresponding terms in all our hypotheses. Starting from hypothesis *eventually_e_inj_m_impl_valid_policy* :

$$\text{B} \vdash \text{Eventually Any (E c1 (inject (msg p)))} \rightarrow \text{msg p} = \text{msg p} \wedge \text{Rcv (msg p)} = \text{comp p}$$

We obtain by simplification *policyRight* :

$$\text{Rcv (msg p)= comp p}$$

We rewrite terms using *policyRight* in our hypotheses. From this, we obtain the following updated hypotheses:

- *c3_not_in_c1_c2* (from *Step 5*):

$$\neg \text{ In (Rcv (msg p))[c1; c2]}$$

- *c3_eventually_enable_to_get_d* (from *Step 5*):

$$\text{B} \vdash \text{Eventually Any (E (Rcv (msg p))(get\_pld (msg p) d))}$$

- *c1_injected_m_to_c2* (from *Step 5*) :

$$\begin{aligned}&\text{B} \vdash \text{(H c1 (inject (msg p)) \&\& has\_rcv (msg p)c2)}\\&<< \text{E (Rcv (msg p))(get\_pld (msg p)d)}\end{aligned}$$

- *eventually_h_comp_p_intercept_msg_p* (from *Step 6*) :

$$\text{B} \vdash \text{Eventually Any (H (Rcv (msg p))(intercept (msg p)))}$$

Step 10. In this step, we show that *c*2 must also be the receiver *Rcv* (*msg p*) of the message *msg p* of the policy *p*, which in turn contradicts the previous result and conclude the proof. Starting from hypothesis *c1_injected_m_to_c2* :

$$\text{B} \vdash \text{(H c1 (inject (msg p)) \&\& has\_rcv (msg p)c2)} << \text{E (Rcv (msg p))(get\_pld (msg p)d)}$$

We obtain *ev_hasRcv_msg2_c2*:

$$\text{c2 = Rcv (msg p)}$$

We substitute these terms in our hypotheses. From this, we deduce the following updated hypothesis:

- $c3\_not\_in\_c1\_c2$ (from *Step 9*) :

$$\neg \text{In (Rcv (msg p))[c1; Rcv (msg p)]}$$

Hypothesis $c3\_not\_in\_c1\_c2$ shows a contradiction, hence, as Coq IDE shows, the proof is complete. **Q.E.D.**

At this level of specification, we are able to: (1) formally represent an interpretation of the confidentiality security objective described in Section 5.5.2 and develop an appropriate policy as properties of the architecture model in Coq, (2) provide the properties as formal model libraries to support reuse, and (3) offer methods and functions to easily check the validity of a property as a policy to indicate the fulfillment of the security objective in a given architecture model, fostering reuse.

6.8 Tool Support

We have implemented two prototypes to support the proposed approach (Section 6.3) as an Eclipse plug-in. The first prototype uses Alloy and the second uses Coq as the support tool for analysis. As presented in 2.6.2, each tool offers its advantages and disadvantages. Alloy offers a better automation, therefore, as we will see in what follows, better reuse and generation support for our approach (e.g., report generation) but verification is bound to scope. Whereas, Coq offers proof for verification, as a consequence, the verification is not limited to a scope, but allows less automation. Consequently, it impacts the reuse and automation support part of our approach and requires more skills (e.g., proof analysis) for the designer.

In both cases, our starting point is the software architecture metamodel presented in Section 6.5 as the metamodel for a software architecture DSL. 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.

6.8.1 Automated formal tool : Alloy

The architecture of the tool using Alloy as the formal tool, as shown in Figure 6.5, is composed of two main blocks: (1) Modeling framework block and (2) Application development block. Each block is composed of a set of modules to support the corresponding set of activities (the numbers in parentheses correspond to the activity numbers in Figure 6.5).

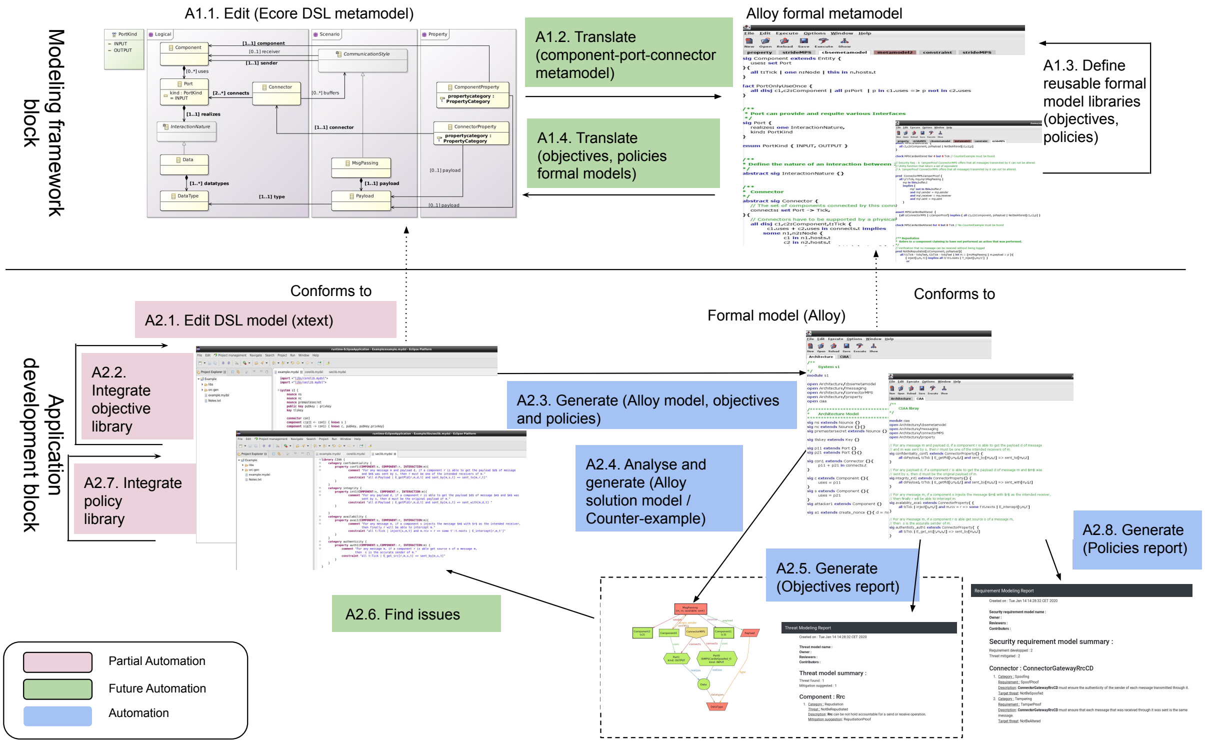

Modeling framework block
A1.1. Edit (Ecore DSL metamodel)
A1.2. Translate (component-port-connector metamodel)
A1.4. Translate (objectives, policies formal models)
Alloy formal metamodel
A1.3. Define reusable formal model libraries (objectives, policies)
Conforms to
Conforms to
Application development block
A2.1. Edit DSL model (xtext)
Formal model (Alloy)
A2.2. Integrate objective library
A2.7. Integrate policy library
A2.3. Generate (Alloy model, objectives and policies)
A2.4. Analyse and generate (Alloy solution model / Counter-example)
A2.8. Generate (Policies report)
A2.5. Generate (Objectives report)
A2.6. Find issues
Threat Modeling Report
Requirement Modeling Report
🟪 Partial Automation
🟩 Future Automation
🟦 Automation

Figure 6.5: Tool support for security objectives using Alloy architecture and artifacts of the approach

6.8.1.1 Modeling framework block

The first block is dedicated to supporting four activities. Activity *A1.1* is responsible for creating the DSL metamodel. The resulting metamodel is used to formally define (in Alloy) the static and behavioral semantics, based upon the DSL metamodel according to the procedure discussed in Section 6.6. We will name this definition the formal metamodel (*A1.2*). Furthermore, reusable objective model libraries and policy model libraries are also defined as Alloy models according to the procedure discussed in Section 6.6.2 (*A1.3*). We will name this definition the formal model libraries. The last activity (*A1.4*) is the definition of a set of DSL model libraries from the Alloy specification of objectives and policies (formal model libraries), using the *property view* of the DSL metamodel (Figure 6.3). Each objective property is associated with a set of policy to satisfy it, during *A1.3*. As an example, we use `confidentiality` as an instance of `PropertyCategory`, `payloadConfidentiality` as an instance of the `ConnectorProperty` within the *CIAAObjective* library and `restrictedGetPld` as an instance of the `ConnectorProperty` within the *SecurityPolicy* library. Finally, we set up that the `restrictedGetPld` policy *satisfy* the `payloadConfidentiality` objective.

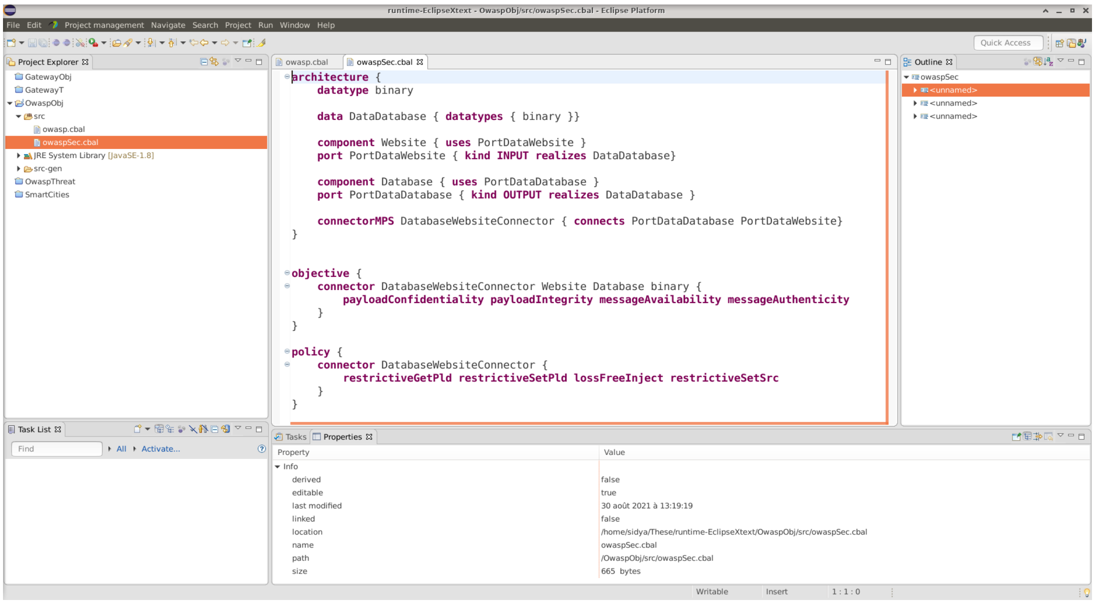
To support these four activities, we used EMF, Xtext to develop the DSL metamodel, and the Alloy Analyzer to encode the corresponding formal metamodel. No further implementation was required.

6.8.1.2 Application development block

The second block is dedicated to supporting eight activities. Activity *A2.1* allows the designer to model a software architecture (DSL model) conforming to the DSL metamodel, where *A2.2* allows to integrate the objectives specification reusing the already developed objective model libraries (*A1.4*). Figure 6.6 shows the model for the example the college library web application.

Then, *A2.3* is the generation of a formal model (in Alloy) from a DSL model, through a transformation engine. The Alloy Analyzer is then invoked, with several iterations, to detect the violation or satisfaction of objective (*A2.4*) and the report on the violated and satisfy objectives may be generated as an HTML document (*A2.5*), such as the one visualized in Figure 6.8. The report is then analyzed (*A2.6*) and *A2.7* allows the designer to add the corresponding security policies to the DSL model reusing the DSL model libraries and the *satisfy* relationship between properties (*A1.4*). The resulting DSL model is then transformed to a formal Alloy model (*A2.3*), where the formal definition of the corresponding security policy is automatically added in the produced Alloy software architecture model from their DSL definitions. At the end, when all objectives are satisfied (i.e., when no counterexample was found when the Alloy Analyzer is invoked), the policy report is generated as an HTML document (*A2.8*), as visualized in Figure 6.9.

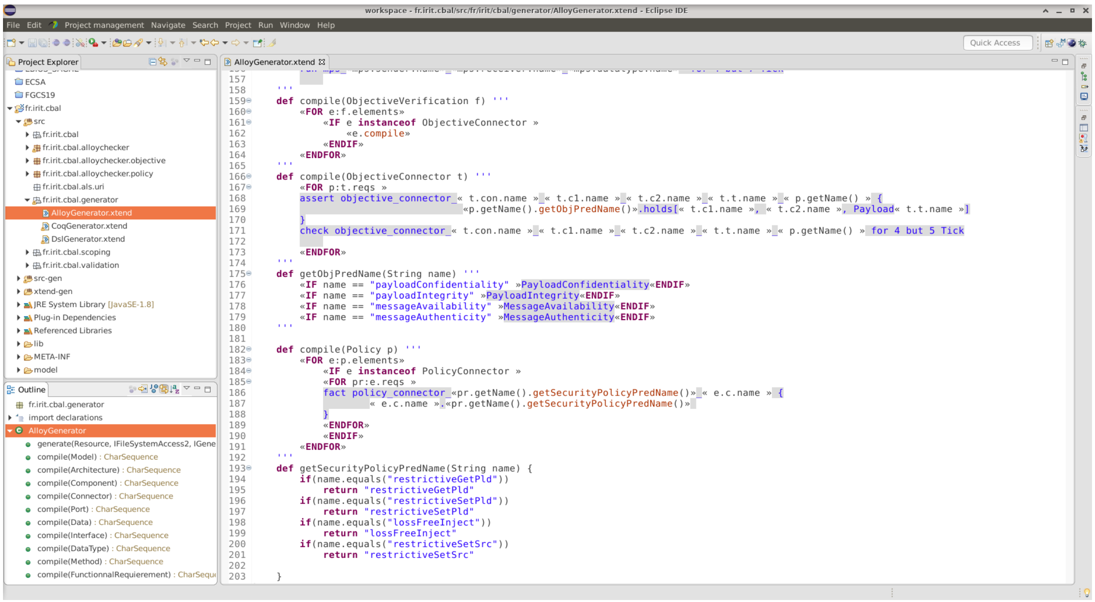
To support these eight activities, we developed a textual editor to model the architecture of the system using EMF and Xtext. In addition, the textual editor provides auto-completion for the manual integration of the objective library (*A2.2*), proposing the possible objective categories and properties for a software architecture elements, when the designer is typing the elements. We also developed two transformation engines to generate the formal Alloy model (*A2.3*) and the HTML reports using Xtend (*A2.5 and A2.8*). The transformation process takes advantage of the template feature provided by Xtend and consists of a set of transformation rules that are applied for each concept (using tree traversal) on a DSL model to generate an Alloy formal model. Figure 6.7 shows an example of such rules to transform concept from our DSL Model to an Alloy formal model. The generation process allows to automatically incorporate the formal definition of the targeted objectives from their DSL definition in the produced Alloy software architecture model. Finally, the Alloy Analyzer is used to verify the resulting models and to generate counterexamples.



```
architecture {
    datatype binary

    data DataDatabase { datatypes { binary }}

    component Website { uses PortDataWebsite }
    port PortDataWebsite { kind INPUT realizes DataDatabase}

    component Database { uses PortDataDatabase }
    port PortDataDatabase { kind OUTPUT realizes DataDatabase }

    connectorMPS DatabaseWebsiteConnector { connects PortDataDatabase PortDataWebsite}
}


objective {
    connector DatabaseWebsiteConnector Website Database binary {
        payloadConfidentiality payloadIntegrity messageAvailability messageAuthenticity
    }
}

policy {
    connector DatabaseWebsiteConnector {
        restrictiveGetPld restrictiveSetPld lossFreeInject restrictiveSetSrc
    }
}
```

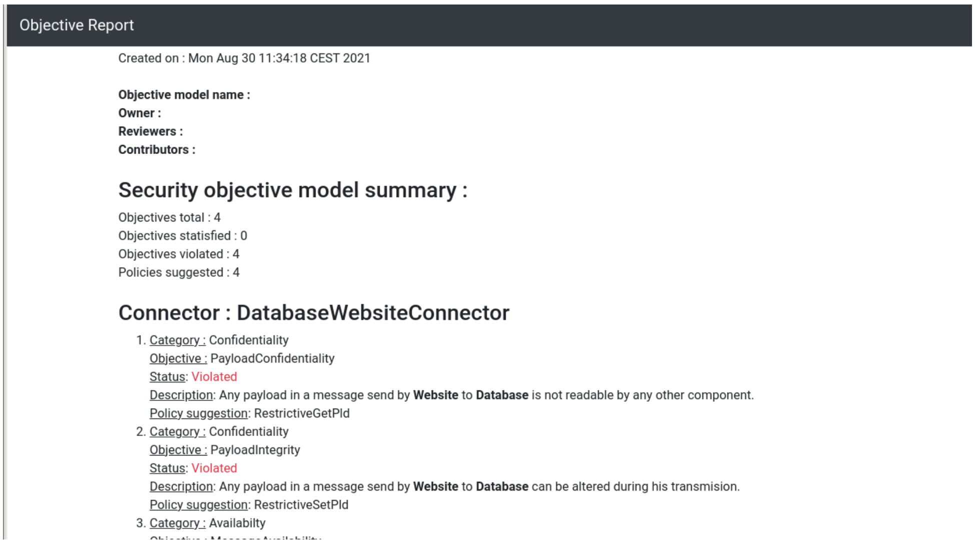
Property	Value
Info	
derived	false
editable	true
last modified	30 août 2021 à 13:19:19
linked	false
location	/home/sidya/These/runtime-EclipseXtext/OwaspObj/src/owaspSec.cbal
name	owaspSec.cbal
path	/OwaspObj/src/owaspSec.cbal
size	665 bytes

Figure 6.6: Definition of the security requirements using security objectives modeling for the college library web application


workspace - fr.irit.cbal/src/fr/irit/cbal/generator/AlloyGenerator.xtend - Eclipse IDE
File Edit Project management Navigate Search Project Run Window Help
Quick Access
Project Explorer
ECSA
FGCS19
fr.irit.cbal
src
fr.irit.cbal
fr.irit.cbal.alloychecker
fr.irit.cbal.alloychecker.objective
fr.irit.cbal.alloychecker.policy
fr.irit.cbal.als.uri
fr.irit.cbal.generator
AlloyGenerator.xtend
CoqGenerator.xtend
DslGenerator.xtend
fr.irit.cbal.scoping
fr.irit.cbal.validation
src-gen
xtend-gen
JRE System Library [JavaSE-1.8]
Plug-in Dependencies
Referenced Libraries
lib
META-INF
model
Outline
fr.irit.cbal.generator
import declarations
AlloyGenerator
generate(Resource, IFileSystemAccess2, IGene
compile(Model) : CharSequence
compile(Architecture) : CharSequence
compile(Component) : CharSequence
compile(Connector) : CharSequence
compile(Port) : CharSequence
compile(Data) : CharSequence
compile(Interface) : CharSequence
compile(DataType) : CharSequence
compile(Method) : CharSequence
compile(FunctionnalRequirement) : CharSequ
AlloyGenerator.xtend

```
    '''
    def compile(ObjectiveVerification f) '''
        «FOR e:f.elements»
            «IF e instanceof ObjectiveConnector »
                «e.compile»
            «ENDIF»
        «ENDFOR»
    '''
    def compile(ObjectiveConnector t) '''
        «FOR p:t.reqs »
        assert objective_connector_« t.con.name »_« t.c1.name »_« t.c2.name »_« t.t.name »_« p.getName() » {
                                    «p.getName().getObjPredName()».holds[« t.c1.name », « t.c2.name », Payload« t.t.name »]
        }
        check objective_connector_« t.con.name »_« t.c1.name »_« t.c2.name »_« t.t.name »_« p.getName() » for 4 but 5 Tick

        «ENDFOR»
    '''
    def getObjPredName(String name) '''
        «IF name == "payloadConfidentiality" »PayloadConfidentiality«ENDIF»
        «IF name == "payloadIntegrity" »PayloadIntegrity«ENDIF»
        «IF name == "messageAvailability" »MessageAvailability«ENDIF»
        «IF name == "messageAuthenticity" »MessageAuthenticity«ENDIF»
    '''

    def compile(Policy p) '''
        «FOR e:p.elements»
            «IF e instanceof PolicyConnector »
            «FOR pr:e.reqs »
            fact policy_connector_«pr.getName().getSecurityPolicyPredName()»_« e.c.name » {
                    « e.c.name ».«pr.getName().getSecurityPolicyPredName()»
            }
            «ENDFOR»
            «ENDIF»
        «ENDFOR»
    '''
    def getSecurityPolicyPredName(String name) {
        if(name.equals("restrictiveGetPld"))
            return "restrictiveGetPld"
        if(name.equals("restrictiveSetPld"))
            return "restrictiveSetPld"
        if(name.equals("lossFreeInject"))
            return "lossFreeInject"
        if(name.equals("restrictiveSetSrc"))
            return "restrictiveSetSrc"

    }
```

Figure 6.7: Transformations supporting the generation to Alloy for security objectives from a DSL model using Xtend



Objective Report

Created on : Mon Aug 30 11:34:18 CEST 2021

Objective model name :
Owner :
Reviewers :
Contributors :

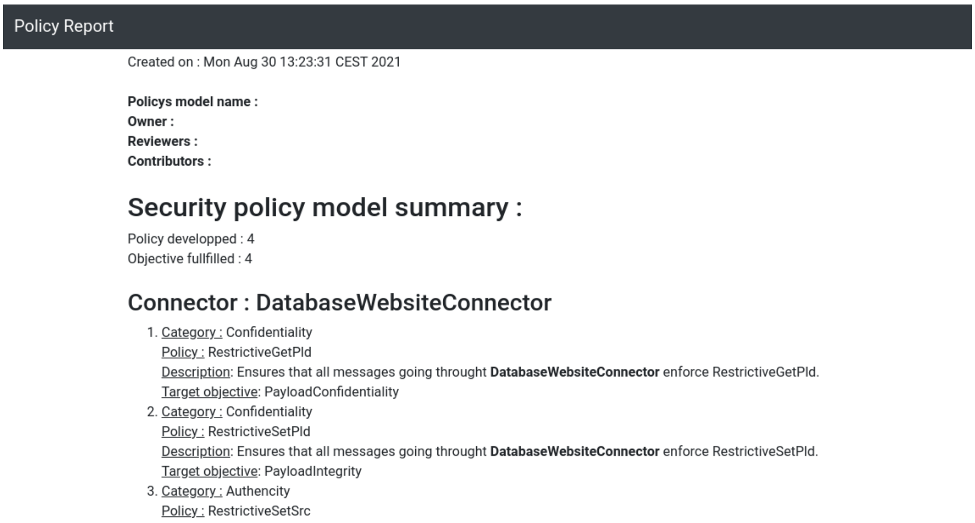
Security objective model summary :

Objectives total : 4
Objectives statisfied : 0
Objectives violated : 4
Policies suggested : 4

Connector : DatabaseWebsiteConnector

1. Category : Confidentiality
   Objective : PayloadConfidentiality
   Status: Violated
   Description: Any payload in a message send by **Website** to **Database** is not readable by any other component.
   Policy suggestion: RestrictiveGetPld
2. Category : Confidentiality
   Objective : PayloadIntegrity
   Status: Violated
   Description: Any payload in a message send by **Website** to **Database** can be altered during his transmision.
   Policy suggestion: RestrictiveSetPld
3. Category : Availabilty
   Objective : MessageAvailability

Figure 6.8: Objective report showing the violated and satisfied security objectives for the college library website application



Policy Report

Created on : Mon Aug 30 13:23:31 CEST 2021

Policys model name :
Owner :
Reviewers :
Contributors :

Security policy model summary :

Policy developped : 4
Objective fullfilled : 4

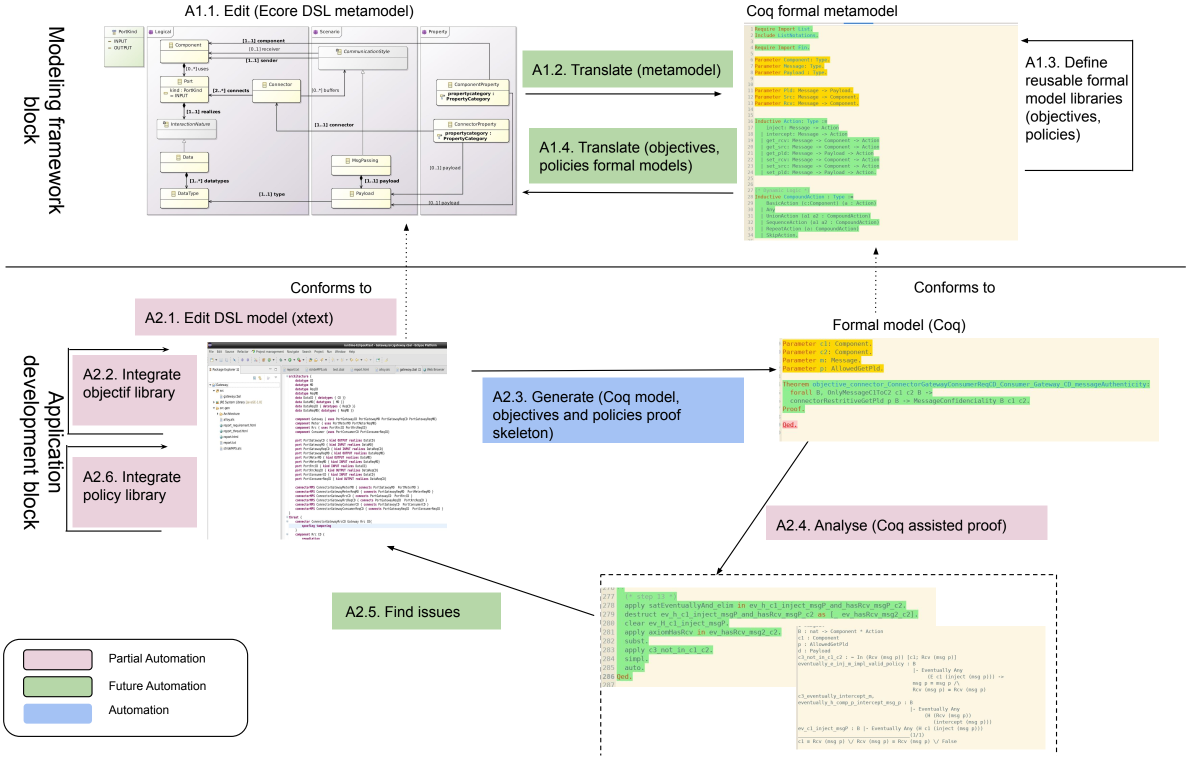
Connector : DatabaseWebsiteConnector

1. Category : Confidentiality
   Policy : RestrictiveGetPld
   Description: Ensures that all messages going throught **DatabaseWebsiteConnector** enforce RestrictiveGetPld.
   Target objective: PayloadConfidentiality
2. Category : Confidentiality
   Policy : RestrictiveSetPld
   Description: Ensures that all messages going throught **DatabaseWebsiteConnector** enforce RestrictiveSetPld.
   Target objective: PayloadIntegrity
3. Category : Authencity
   Policy : RestrictiveSetSrc

Figure 6.9: Policy report showing the policies applied to fulfill security objectives for the college library website application

Putting all this together and using the developed tooled framework, we are able to: (1) model security requirements from the positive vision using the CIAA security objectives at the architecture design level using a domain-specific language, (2) generate an interpretation of the resulted software architecture model and properties in Alloy, (3) verify the satisfaction of a set of security requirements for a concrete application through reusable model of previously specified and verified security objectives, (4) treat the detected properties violation using appropriate policy models, and (5) generate new artifacts as feedback on the software architecture model and its security.

6.8.2 Proof assistant : Coq



```
Require Import List.
Include ListNotations.

Require Import Fin.

Parameter Component: Type.
Parameter Message: Type.
Parameter Payload : Type.


Parameter Pld: Message -> Payload.
Parameter Src: Message -> Component.
Parameter Rcv: Message -> Component.


Inductive Action: Type :=
    inject: Message -> Action
  | intercept: Message -> Action
  | get_rcv: Message -> Component -> Action
  | get_src: Message -> Component -> Action
  | get_pld: Message -> Payload -> Action
  | set_rcv: Message -> Component -> Action
  | set_src: Message -> Component -> Action
  | set_pld: Message -> Payload -> Action.


(* Dynamic Logic *)
Inductive CompoundAction : Type :=
    BasicAction (c:Component) (a : Action)
  | Any
  | UnionAction (a1 a2 : CompoundAction)
  | SequenceAction (a1 a2 : CompoundAction)
  | RepeatAction (a: CompoundAction)
  | SkipAction.
```

```
Parameter c1: Component.
Parameter c2: Component.
Parameter m: Message.
Parameter p: AllowedGetPld.

Theorem objective_connector_ConnectorGatewayConsumerReqCD_Consumer_Gateway_CD_messageAuthenticity:
  forall B, OnlyMessageC1ToC2 c1 c2 B ->
  connectorRestritiveGetPld p B -> MessageConfidenciality B c1 c2.
Proof.

Qed.
```

```
  (* step 13 *)
  apply satEventuallyAnd_elim in ev_h_c1_inject_msgP_and_hasRcv_msgP_c2.
  destruct ev_h_c1_inject_msgP_and_hasRcv_msgP_c2 as [_ ev_hasRcv_msg2_c2].
  clear ev_H_c1_inject_msgP.
  apply axiomHasRcv in ev_hasRcv_msg2_c2.
  subst.
  apply c3_not_in_c1_c2.
  simpl.
  auto.
Qed.
```

```
B : nat -> Component * Action
c1 : Component
p : AllowedGetPld
d : Payload
c3_not_in_c1_c2 : ~ In (Rcv (msg p)) [c1; Rcv (msg p)]
eventually_e_inj_m_impl_valid_policy : B
                                       |- Eventually Any
                                            (E c1 (inject (msg p))) ->
                                       msg p = msg p /\
                                       Rcv (msg p) = Rcv (msg p)
c3_eventually_intercept_m,
eventually_h_comp_p_intercept_msg_p : B
                                      |- Eventually Any
                                           (H (Rcv (msg p))
                                              (intercept (msg p)))
ev_c1_inject_msgP : B |- Eventually Any (H c1 (inject (msg p)))
______________________________________(1/1)
c1 = Rcv (msg p) \/ Rcv (msg p) = Rcv (msg p) \/ False
```

Figure 6.10: Tool support for security objectives using Coq architecture and artifacts of the approach

The architecture of the tool support using Coq as the formal tool, as shown in Figure 6.10, is composed of two main blocks: (1) Modeling framework block and (2) Application development block. Each block is composed of a set of modules to support the corresponding set of activities (the numbers in parentheses correspond to the activity numbers in Figure 6.10).

6.8.2.1 Modeling framework block

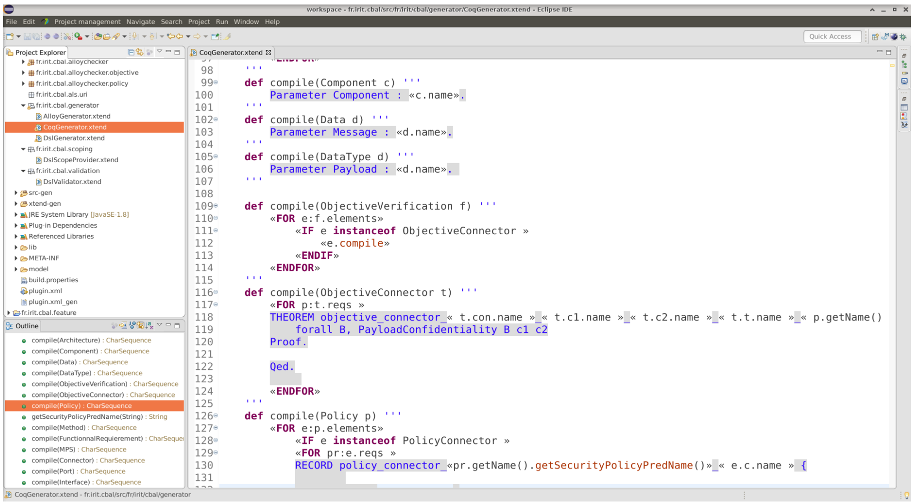
The first block is dedicated to supporting four activities. Activity *A1.1* is responsible for creating the DSL metamodel. The resulting metamodel is used to formally define (in Coq) the static and behavioral semantics, based upon the DSL metamodel according to the procedure discussed in Section 6.7. We will name this definition the formal metamodel (*A1.2*). Furthermore, reusable objective model libraries and policy model libraries are also defined as Coq models according to the procedure discussed in Section 6.7.2 (*A1.3*). We will name this definition the formal model libraries. The last activity (*A1.4*) is the definition of a set of DSL model libraries from the Alloy specification of objectives and policies (formal model libraries), using the *property view* of the DSL metamodel (Figure 6.3). Each objective property is associated with a set of policy to satisfy it, during *A1.3*.

To support these four activities, we used EMF, Xtext to develop the DSL metamodel, and Coq to encode the corresponding formal metamodel. No further implementation was required.

6.8.2.2 Application development block

The second block is dedicated to supporting six activities. Activity *A2.1* allows the designer to model a software architecture (DSL model) conforming to the DSL metamodel, where *A2.2* allows to integrate the objectives specification reusing the already developed objective model libraries (*A1.4*). Then, *A2.3* is the generation of a formal model (in Coq) from a DSL model, through a transformation engine. Then, the designer can use the Coq IDE as proof assistant and complete the analyse the model and ensure the satisfaction of the required objectives (*A2.4*). From the results of the analysis, the designer can deduce the security issues (*A2.5*). Then in activity *A2.6*, the designer can add the corresponding security policies to the DSL model reusing the DSL model libraries. The resulting DSL model is then transformed to a formal Coq model (*A2.3*), where the formal definition of the corresponding security policy is automatically added in the produced Coq software architecture model from their DSL definitions. Finally, when all objectives are satisfied, i.e., when the designer did and validated all the required proofs, the system is considered secure.

To support these six activities, we developed a textual editor to model the architecture of the system using EMF and Xtext. In addition, the textual editor provides auto-completion for the manual integration of the objective library (*A2.2*), proposing the possible objective categories and properties for a software architecture elements, when the designer is typing the elements. We also developed two transformation engines to generate the formal Coq model (*A2.3*). The transformation process takes advantage of the template feature provided by Xtend and consists of a set of transformation rules that are applied for each concept (using tree traversal) on a DSL model to generate a Coq formal model. Figure 6.11 shows examples of such a rule to transform concept from our DSL Model to a Coq formal model. The generation process allows to incorporate the formal definition of the targeted objectives from their DSL definition in the produced Coq software architecture model. Finally, the Coq IDE is used to complete and verify the resulting models using proofs.


workspace - fr.irit.cbal/src/fr/irit/cbal/generator/CoqGenerator.xtend - Eclipse IDE
File Edit Project management Navigate Search Project Run Window Help
Quick Access
Project Explorer
fr.irit.cbal.alloychecker
fr.irit.cbal.alloychecker.objective
fr.irit.cbal.alloychecker.policy
fr.irit.cbal.als.uri
fr.irit.cbal.generator
AlloyGenerator.xtend
CoqGenerator.xtend
DslGenerator.xtend
fr.irit.cbal.scoping
DslScopeProvider.xtend
fr.irit.cbal.validation
DslValidator.xtend
src-gen
xtend-gen
JRE System Library [JavaSE-1.8]
Plug-in Dependencies
Referenced Libraries
lib
META-INF
model
build.properties
plugin.xml
plugin.xml_gen
fr.irit.cbal.feature
Outline
compile(Architecture) : CharSequence
compile(Component) : CharSequence
compile(Data) : CharSequence
compile(DataType) : CharSequence
compile(ObjectiveVerification) : CharSequence
compile(ObjectiveConnector) : CharSequence
compile(Policy) : CharSequence
getSecurityPolicyPredName(String) : String
compile(Method) : CharSequence
compile(FunctionnalRequierement) : CharSeque
compile(MPS) : CharSequence
compile(Connector) : CharSequence
compile(Port) : CharSequence
compile(Interface) : CharSequence
CoqGenerator.xtend
CoqGenerator.xtend - fr.irit.cbal/src/fr/irit/cbal/generator

```
    '''
    def compile(Component c) '''
        Parameter Component : «c.name».
    '''
    def compile(Data d) '''
        Parameter Message : «d.name».
    '''
    def compile(DataType d) '''
        Parameter Payload : «d.name».
    '''

    def compile(ObjectiveVerification f) '''
        «FOR e:f.elements»
            «IF e instanceof ObjectiveConnector »
                «e.compile»
            «ENDIF»
        «ENDFOR»
    '''
    def compile(ObjectiveConnector t) '''
        «FOR p:t.reqs »
        THEOREM objective_connector_« t.con.name »_« t.c1.name »_« t.c2.name »_« t.t.name »_« p.getName()
            forall B, PayloadConfidentiality B c1 c2
        Proof.

        Qed.

        «ENDFOR»
    '''
    def compile(Policy p) '''
        «FOR e:p.elements»
            «IF e instanceof PolicyConnector »
            «FOR pr:e.reqs »
            RECORD policy_connector_«pr.getName().getSecurityPolicyPredName()»_« e.c.name » {

```

Figure 6.11: Transformations supporting the generation to Coq for security objectives from a DSL model using Xtend

Putting all this together and using the developed tooled framework, we are able to: (1) model security requirements from the positive vision using the confidentiality security objective at the architecture design level using a domain-specific language, (2) generate an interpretation of the resulted software architecture model and properties in Coq, (3) verify the satisfaction of a set of security requirements for a concrete application through reusable models, (4) treat the validity of security objectives using appropriate policy models, and (5) generate new artifacts as feedback on the software architecture model and its security.

6.9 Conclusion

In this chapter, we proposed an approach to security objectives specification, verification, and treatment in component-based software architecture models following two levels of specifications : (1) logical specification of security objectives in the form of security properties of the system model using first-order logic, as an abstract and technology-independent formalism; (2) an interpretation of the logical specification in Alloy as tooled languages, and (3) for each representative CIAA security property, an appropriate security policy to make the security property hold. In addition, we presented our first experiment of using Coq, as a proof assistant, in the context of security-by design to check the validity of a property as a policy to indicate the fulfillment of the security objective.

We keep the domain specifying security requirement categories at an abstract level to remain independent of any particular technology or implementation of the model aligned with the approach described in Section 6.3. This can allow these formal specifications to be applied to different implementations or architectures supporting the notion of message-passing communication which fosters reuse.

Furthermore, we walked through an MDE-based prototype connected to a tooled formal language (Alloy and Coq) to support the proposed approach. An example of this tool suite is constructed using EMFT, Xtext, and Xtend and is currently provided in the form of Eclipse plug-ins. The combined formal modeling and MDE to specify security objectives and develop their targeted system security requirements allows to develop an accurate analysis, for evaluation and/or certification.

Chapter 7

Evaluation of the contributions

Contents

7.1 Introduction

This chapter assesses the feasibility of the contributions of our work. We first report an industrial case study performed in the metrology domain Section 7.2, followed by discussions on the contributions and feasibility of the proposed approaches (Section 7.3). The case study enables us to determine whether the approach leads to a simplification of the engineering process steps, whereas the discussions highlight the benefits and potential applications of the proposed approach, as well as the potential for its generalization and extension. With regard to our contributions, we deal particularly with ***C7*** related to the research objective ***RO2***.

7.2 Case study

We use smart meter gateway systems to exemplify the proposed approaches. The aim of this case study is providing a methodology to improve existing approaches to engineering secure systems using MDE and formal techniques. We evaluate the proposed approach in the construction of a methodology that is adapted for engineering secure systems focusing on the architecture design, from the negative and positive perspectives.

7.2.1 Expressing the architecture of the smart meter gateway

We start by defining the component types and the interfaces and connectors using the software architecture meta-model concepts of our DSL model.

Listing 7.1 depicts the DSL model specification of the smart meter gateway application architecture described in Figure 2.10, with respect to the functional requirements. Lines 2-5 specify the data types required to represent the consumption data, meter data, requested consumption data, and requested meter data, respectively. Lines 6-9 specify the data components of the component-port-connector meta-model (see Section 4.6.2). Similarly, Lines 10-13 specifies the connectors, Lines 14-23 specify the ports, and Lines 24-29 specify the connectors for the smart meter gateway application according to the component-port-connector meta-model.

```
 1  architecture {
 2    datatype CD
 3    datatype MD
 4    datatype ReqCD
 5    datatype ReqMD
 6    data DataCD { datatypes { CD }}
 7    data DataMD{ datatypes { MD }}
 8    data DataReqCD { datatypes { ReqCD }}
 9    data DataReqMD{ datatypes { ReqMD }}
10    component Gateway { uses PortGatewayCD PortGatewayMD PortGatewayReqCD
         PortGatewayReqMD}
11    component Meter { uses PortMeterMD PortMeterReqMD}
12    component Rrc { uses PortRrcCD PortRrcReqCD}
13    component Consumer {uses PortConsumerCD PortConsumerReqCD}
14    port PortGatewayCD { kind OUTPUT realizes DataCD}
15    port PortGatewayMD { kind INPUT realizes DataMD}
16    port PortGatewayReqCD { kind INPUT realizes DataReqCD}
17    port PortGatewayReqMD { kind OUTPUT realizes DataReqMD}
18    port PortMeterMD { kind OUTPUT realizes DataMD}
19    port PortMeterReqMD { kind INPUT realizes DataReqMD}
20    port PortRrcCD { kind INPUT realizes DataCD}
21    port PortRrcReqCD { kind OUTPUT realizes DataReqCD}
22    port PortConsumerCD { kind INPUT realizes DataCD}
23    port PortConsumerReqCD { kind OUTPUT realizes DataReqCD}
24    connectorMPS ConnectorGatewayMeterMD { connects PortGatewayMD
         PortMeterMD }
25    connectorMPS ConnectorGatewayMeterReqMD { connects PortGatewayReqMD
         PortMeterReqMD }
26    connectorMPS ConnectorGatewayRrcCD { connects PortGatewayCD PortRrcCD
```

```
    }
  connectorMPS ConnectorGatewayRrcReqCD { connects PortRrcReqCD
      PortGatewayReqCD }
  connectorMPS ConnectorGatewayConsumerCD { connects PortGatewayCD
      PortConsumerCD }
  connectorMPS ConnectorGatewayConsumerReqCD { connects
      PortConsumerReqCD PortGatewayReqCD   }
}
```

Listing 7.1: A smart meter gateway application using our DSL Model

7.2.2 Security analysis

In the next step, the architect can specify security requirements upon the architecture model. The security characteristics of the *Smart Meter Gateway* can be easily extracted from the protection profile of the gateway of a smart metering system [18]. In this study, we focus only on the protection of the *consumption data* asset (with some revisited information). The consumption data represents the billing-relevant part of the meter data.

This asset faces the following threats:

1. Spoofing, tampering, elevation of privilege, repudiation: comparable to a classic meter and its security requirements

2. Information disclosure: due to privacy concerns

3. Denial of Service: loss of data during the communication hinders proper operation

The designer can do an analysis using either the negative view using threat modeling, or the positive view using objective modeling, or both in a complementary way. In the following, we do the analysis for the *Smart Meter Gateway* first using the negative view, and then the positive view.

7.2.2.1 Negative perspective (security threats)

The architect can reuse the previously defined DSL security threat libraries (e.g., *senderSpoofing*) to seek threats on the concrete architecture (see Listing 7.2). According to the smart meter architecture and its security characteristics, we consider the connectors that are transmitting consumption data (i.e., *ConnectorGatewayRrcCD, ConnectorGatewayRrcReqCD, ConnectorGatewayConsumerCD* and *ConnectorGatewayConsumerReqCD*) and components that are injecting/intercepting consumption data (i.e., *Gateway*, *Consumer*, and *Rrc*). For example, Lines 2-3 specifies that the *ConnectorGatewayRrcCD* connector faces spoofing, tampering, denial-of-service, and information disclosure threats. Similarly, Lines 10-11 specifies that the *Gateway* component faces repudiation and elevation of privilege threats.

```
 1  threat {
 2    connector ConnectorGatewayRrcCD Gateway Rrc CD {
 3      senderSpoofing payloadTampering payloadStaleness payloadDisclosure }
 4    connector ConnectorGatewayRrcReqCD Rrc Gateway CD {
 5      senderSpoofing payloadTampering payloadStaleness payloadDisclosure }
 6    connector ConnectorGatewayConsumerCD Gateway Consumer CD {
 7      senderSpoofing payloadTampering payloadStaleness payloadDisclosure }
 8    connector ConnectorGatewayConsumerReqCD Consumer Gateway CD {
 9      senderSpoofing payloadTampering payloadStaleness payloadDisclosure }
10    component Gateway CD {
11      sendReceiveRepudiation sendReceiveElevation }
12    component Rrc CD {
13      sendReceiveRepudiation sendReceiveElevation }
14    component Consumer CD {
15      sendReceiveRepudiation sendReceiveElevation }
16  }
```

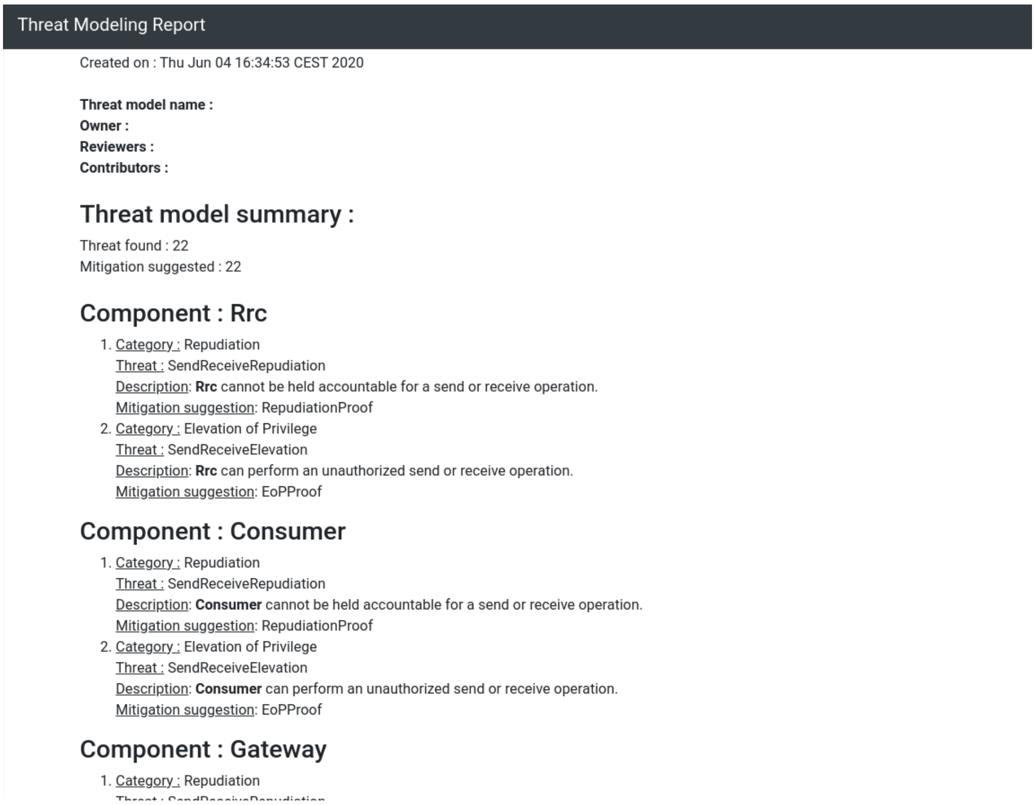
Listing 7.2: Threat detection on consumption data asset

The next step is to use the model transformation engine to generate the corresponding Alloy model, where the formal specification of threats are incorporated into the generated model reusing the threat Alloy model libraries (Listing 7.3). In the generated Alloy model shown in Listing 7.3, the model transformation engine specifies the necessary datatypes, data, components, ports (Lines 1-12) corresponding to the DSL model depicted in Listing 7.1. Furthermore, model transformation engine specifies uses the threat detection specification depicted in Listing 7.2 to specify assertions indicating the presence of threats within the model. For example, Lines 14-16 assert and check that there are no spoofing threats in the *ConnectorGatewayRrcCD* connector with respect to the consumption data. Similarly, Lines 17-19 assert and check that there are no tampering threats, Lines 20-23 assert and check that there are no denial-of-service threats, and Lines 24-26 assert and check that there are no information disclosure threats in the *ConnectorGatewayRrcCD* connector with respect to the consumption data. Taken together, Lines 14-26 in Listing 7.3 correspond to the threat detection specification on the consumption data shown in Listing 7.2, Lines 2-3. Upon verification, the generated threat modeling report (Figure 7.1) shows that several threats are detected for the considered connectors and components.

```
sig CD extends DataType {}
sig PayloadCD extends Payload {}{
  type = CD }
...
sig DataCD extends Data {}{
  datatypes =  CD }
...
sig Gateway extends Component {}{
  uses = PortGatewayCD + PortGatewayMD + PortGatewayReqCD +
      PortGatewayReqMD }
...
sig PortGatewayCD extends Port {}{
  realizes = DataCD }
...
assert threat_connector_ConnectorGatewayRrcCD_
    Gateway_Rrc_CD_senderSpoofing {
  no SenderSpoofing.holds[Gateway, Rrc, PayloadCD]}
check threat_connector_ConnectorGatewayRrcCD_
    Gateway_Rrc_CD_senderSpoofing for 3 but 5 Tick
assert threat_connector_ConnectorGatewayRrcCD_
    Gateway_Rrc_CD_payloadTampering {
  no PayloadTampering.holds[Gateway, Rrc, PayloadCD]}
check threat_connector_ConnectorGatewayRrcCD_
    Gateway_Rrc_CD_payloadTampering for 3 but 5 Tick
assert threat_connector_ConnectorGatewayRrcCD_
    Gateway_Rrc_CD_payloadDropping {
  no PayloadDropping.holds[Gateway, Rrc, PayloadCD]}
check threat_connector_ConnectorGatewayRrcCD_
    Gateway_Rrc_CD_payloadDropping for 3 but 5 Tick
assert threat_connector_ConnectorGatewayRrcCD_
    Gateway_Rrc_CD_payloadDisclosure {
  no PayloadDisclosure.holds[Gateway, Rrc, PayloadCD]
}
check threat_connector_ConnectorGatewayRrcCD_
    Gateway_Rrc_CD_payloadDisclosure for 3 but 5 Tick
...
assert threat_component_Gateway_CD_sendReceiveElevation {
  no SendReceiveElevation.holds[Gateway, PayloadCD]}
check threat_component_Gateway_CD_sendReceiveElevation for 3 but 5 Tick

assert threat_component_Gateway_CD_sendReceiveRepudiation {
  no SendReceiveRepudiation.holds[Gateway, PayloadCD]}
```

```
check threat_component_Gateway_CD_sendReceiveRepudiation for 3 but 5 Tick
...
```

Listing 7.3: Generated Alloy model for the threat detection on consumption data asset



Threat Modeling Report

Created on : Thu Jun 04 16:34:53 CEST 2020

Threat model name :
Owner :
Reviewers :
Contributors :

Threat model summary :

Threat found : 22
Mitigation suggested : 22

Component : Rrc

1. Category : Repudiation
   Threat : SendReceiveRepudiation
   Description: **Rrc** cannot be held accountable for a send or receive operation.
   Mitigation suggestion: RepudiationProof
2. Category : Elevation of Privilege
   Threat : SendReceiveElevation
   Description: **Rrc** can perform an unauthorized send or receive operation.
   Mitigation suggestion: EoPProof

Component : Consumer

1. Category : Repudiation
   Threat : SendReceiveRepudiation
   Description: **Consumer** cannot be held accountable for a send or receive operation.
   Mitigation suggestion: RepudiationProof
2. Category : Elevation of Privilege
   Threat : SendReceiveElevation
   Description: **Consumer** can perform an unauthorized send or receive operation.
   Mitigation suggestion: EoPProof

Component : Gateway

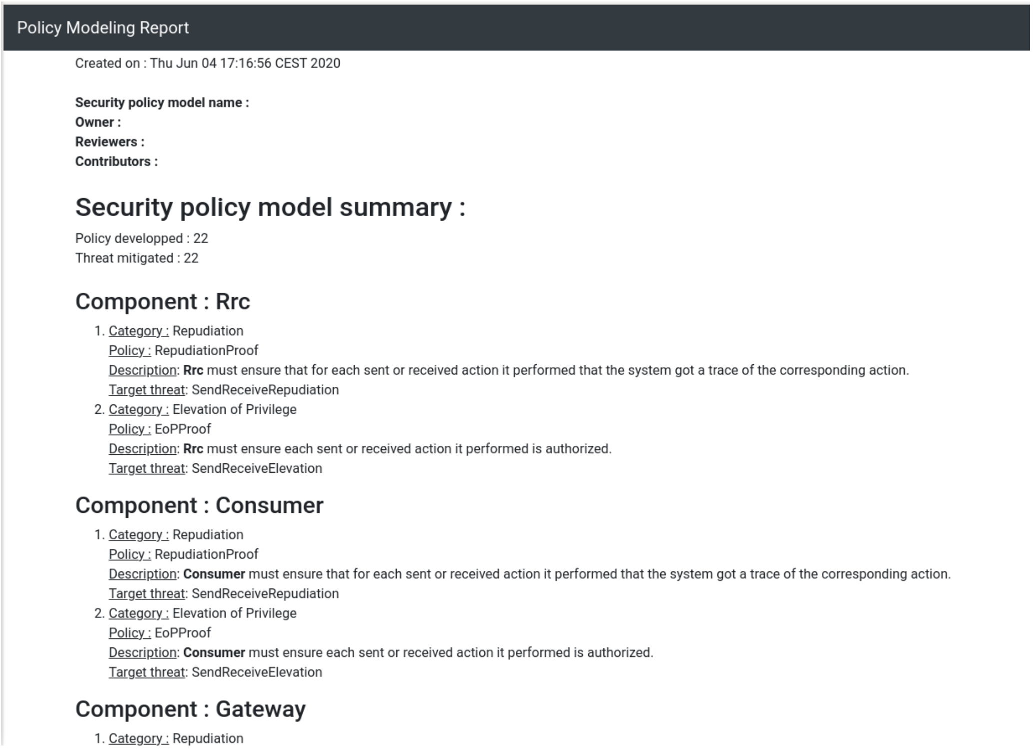
1. Category : Repudiation
   Threat : SendReceiveRepudiation

Figure 7.1: Threat report (Detected threats in the Gateway application)

Then, as depicted in Listing 7.2, the architect revisits the set of policies reusing the previously developed DSL security policy libraries with the help of the generated modeling report that suggests adequate security policies associated with the identified threats (e.g., *spoofProof*) in order to mitigate them. The final verification generates a threat modeling report that shows no more threats are found, meaning that the set of suggested security policies are complete regarding the identified threats. Then, a policy modeling report (Figure 7.2) that sums up this set of security policies is generated. This check indicates that the system model is secure with respect to the consumption data asset.

```
policy {
  connector ConnectorGatewayRrcCD {
    spoofProof tamperProof staleProof informationDisclosureProof }

  connector ConnectorGatewayRrcReqCD {
    spoofProof tamperProof staleProof informationDisclosureProof }
  connector ConnectorGatewayConsumerCD {
    spoofProof tamperProof staleProof informationDisclosureProof }
  connector ConnectorGatewayConsumerReqCD {
    spoofProof tamperProof staleProof informationDisclosureProof }
  component Gateway {
    repudationProof EoPProof }
  component Rrc {
    repudationProof EoPProof }
  component Consumer {
    repudationProof EoPProof }
}
```

Listing 7.4: Specification of policy of the gateway application for mitigate threats



Policy Modeling Report

Created on : Thu Jun 04 17:16:56 CEST 2020

Security policy model name :
Owner :
Reviewers :
Contributors :

Security policy model summary :

Policy developped : 22
Threat mitigated : 22

Component : Rrc

1. Category : Repudiation
   Policy : RepudiationProof
   Description: **Rrc** must ensure that for each sent or received action it performed that the system got a trace of the corresponding action.
   Target threat: SendReceiveRepudiation
2. Category : Elevation of Privilege
   Policy : EoPProof
   Description: **Rrc** must ensure each sent or received action it performed is authorized.
   Target threat: SendReceiveElevation

Component : Consumer

1. Category : Repudiation
   Policy : RepudiationProof
   Description: **Consumer** must ensure that for each sent or received action it performed that the system got a trace of the corresponding action.
   Target threat: SendReceiveRepudiation
2. Category : Elevation of Privilege
   Policy : EoPProof
   Description: **Consumer** must ensure each sent or received action it performed is authorized.
   Target threat: SendReceiveElevation

Component : Gateway

1. Category : Repudiation

Figure 7.2: Policy report (Policies applied to mitigate the detected threats in the Gateway application)

7.2.2.2 Positive perspective (security objectives)

The architect can reuse the previously defined DSL security objectives libraries (e.g., *payloadConfidentiality*) to seek the satisfaction on the concrete architecture (see Listing 7.5). According to the smart meter architecture and its security characteristics, we consider the connectors that are transmitting consumption data (i.e., *ConnectorGatewayRrcCD, ConnectorGatewayRrcReqCD, ConnectorGatewayConsumerCD* and *ConnectorGateway-ConsumerReqCD*) and components that are injecting/intercepting consumption data (i.e., *Gateway*, *Consumer*, and *Rrc*). For example, Lines 2-4 specifies that the *ConnectorGatewayRrcCD* connector must ensure confidentiality, integrity, availability, and authenticity objectives.

```
 1 objective {
 2   connector ConnectorGatewayRrcCD Gateway Rrc CD{
 3     payloadConfidentiality payloadIntegrity messageAvailability
         messageAuthenticity
 4   }
 5   connector ConnectorGatewayRrcReqCD Rrc Gateway CD {
 6     payloadConfidentiality payloadIntegrity messageAvailability
         messageAuthenticity
 7   }
 8   connector ConnectorGatewayConsumerCD Gateway Consumer CD{
 9     payloadConfidentiality payloadIntegrity messageAvailability
         messageAuthenticity
10   }
11   connector ConnectorGatewayConsumerReqCD Consumer Gateway CD{
12     payloadConfidentiality payloadIntegrity messageAvailability
         messageAuthenticity
13   }
14 }
```

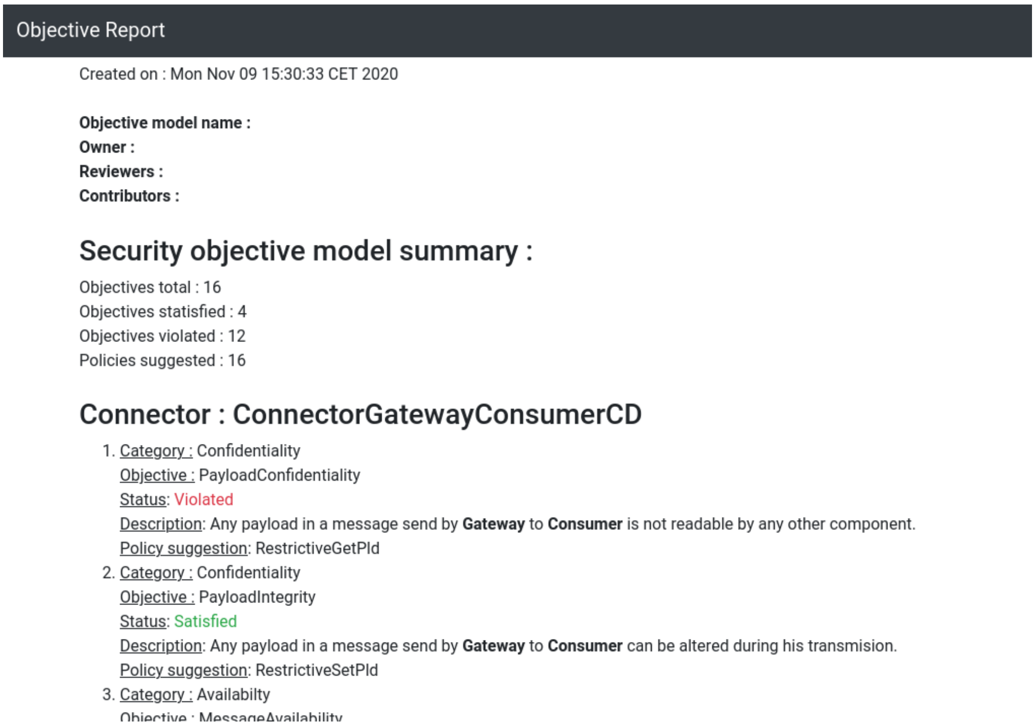
Listing 7.5: Objective satisfaction on consumption data asset

The next step is to use the model transformation engine to generate the corresponding Alloy model, where the formal specification of objectives are incorporated into the generated model reusing the objectives Alloy model libraries (Listing 7.5). In the generated Alloy model shown in Listing 7.1, the model transformation engine specifies the necessary datatypes, data, components, and ports (Lines 1-12) corresponding to the DSL model depicted in Listing 4.7.1. Furthermore, model transformation engine specifies uses the objectives validation specification depicted in Listing 7.6 to specify assertions ensure the satisfaction of objectives within the model. For example, Lines 17-19 assert and check that the confidentiality objective is satisfied in the *ConnectorGatewayRrcCD* connector with respect to the consumption data. Similarly, Lines 20-22 assert and check that the integrity objective is respected, Lines 23-25 assert and check that the availability objective is valid, and Lines 27-28 assert and check that the authenticity objective is satisfied in the *ConnectorGatewayConsumerReqCD* connector with respect to the consumption data. Upon verification, the generated objective modeling report (Figure 7.3) shows that several objectives are violated (i.e., are not satisfied) for the considered connectors.

```
 1 sig CD extends DataType {}
 2 sig PayloadCD extends MsgPassing {}{payload.type = CD}
 3 ...
 4 sig DataCD extends Data {}{
 5   datatypes =  CD }
 6 ...
 7 sig Gateway extends Component {}{
 8   uses = PortGatewayCD + PortGatewayMD + PortGatewayReqCD +
        PortGatewayReqMD }
 9 sig Meter extends Component {}{
10   uses = PortMeterMD + PortMeterReqMD }
11 ...
12 sig ConnectorGatewayMeterMD extends ConnectorMPS {}{
13     all t:Tick | PortGatewayMD + PortMeterMD in connects.t  }
14 sig ConnectorGatewayMeterReqMD extends ConnectorMPS {}{
15     all t:Tick | PortGatewayReqMD + PortMeterReqMD in connects.t }
16 ...
17 assert objective_connector_ConnectorGatewayRrcCD_
      Gateway_Rrc_CD_payloadConfidentiality {
18     PayloadConfidentiality.holds[Gateway, Rrc, PayloadCD] }
19 check objective_connector_ConnectorGatewayRrcCD_
      Gateway_Rrc_CD_payloadConfidentiality for 4 but 5 Tick
20 assert objective_connector_ConnectorGatewayRrcCD_
      Gateway_Rrc_CD_payloadIntegrity {
21   PayloadIntegrity.holds[Gateway, Rrc, PayloadCD] }
22 check objective_connector_ConnectorGatewayRrcCD_
      Gateway_Rrc_CD_payloadIntegrity for 4 but 5 Tick
23 assert objective_connector_ConnectorGatewayRrcCD_
      Gateway_Rrc_CD_messageAvailability {
24     MessageAvailabilit.holds[Gateway, Rrc, PayloadCD] }
25 check objective_connector_ConnectorGatewayRrcCD_
      Gateway_Rrc_CD_messageAvailability for 4 but 5 Tick
26 ...
27 assert objective_connector_ConnectorGatewayConsumerReqCD_
      Consumer_Gateway_CD_messageAuthenticity { MessageAuthenticity.holds[
      Consumer, Gateway, PayloadCD] }
```

```
check objective_connector_ConnectorGatewayConsumerReqCD_
   Consumer_Gateway_CD_messageAuthenticity for 4 but 5 Tick
```

Listing 7.6: Generated Alloy model for the objective validation on consumption data asset



Objective Report

Created on : Mon Nov 09 15:30:33 CET 2020

Objective model name :
Owner :
Reviewers :
Contributors :

Security objective model summary :

Objectives total : 16
Objectives statisfied : 4
Objectives violated : 12
Policies suggested : 16

Connector : ConnectorGatewayConsumerCD

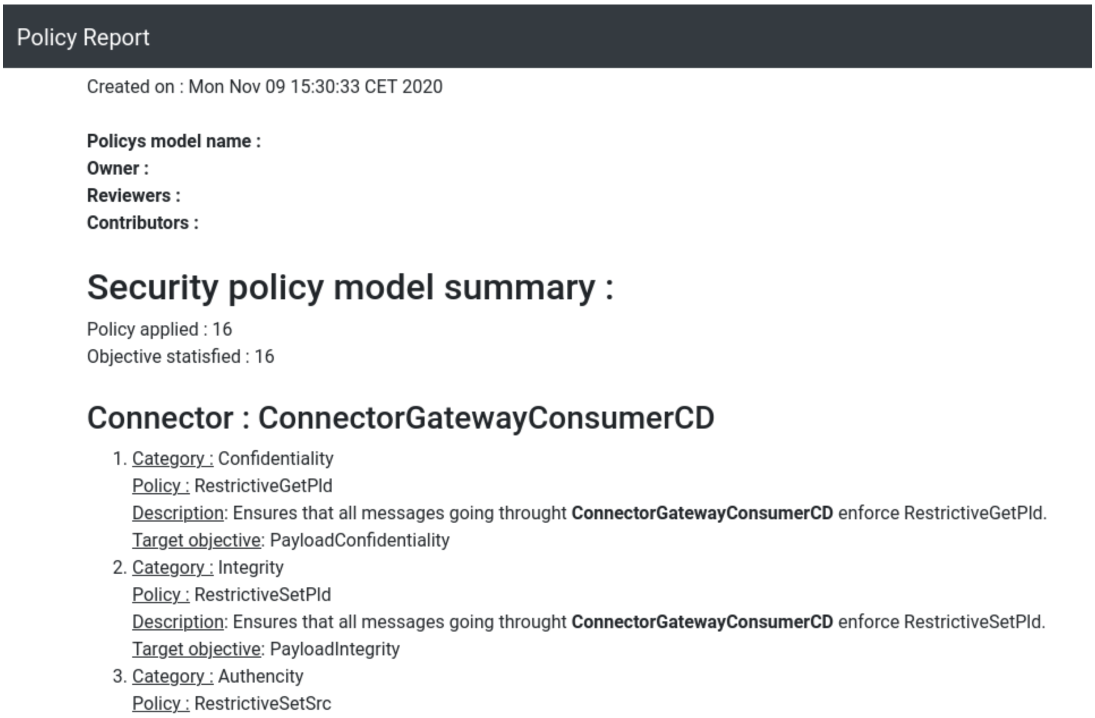
1. Category : Confidentiality
   Objective : PayloadConfidentiality
   Status: Violated
   Description: Any payload in a message send by **Gateway** to **Consumer** is not readable by any other component.
   Policy suggestion: RestrictiveGetPld
2. Category : Confidentiality
   Objective : PayloadIntegrity
   Status: Satisfied
   Description: Any payload in a message send by **Gateway** to **Consumer** can be altered during his transmision.
   Policy suggestion: RestrictiveSetPld
3. Category : Availabilty
   Objective : MessageAvailability

Figure 7.3: Objective report (Violated and satisfied objectives in the Gateway application)

Then, as depicted in Listing 7.4, the architect revisits the set of requirements reusing the previously developed DSL policy libraries with the help of the generated modeling report that suggests adequate policy to fulfill identified violated objective (e.g., *restriveSetPld*) in order to satisfy them. The final verification generates an objective modeling report that shows all the security objectives are satisfied, meaning that the set of suggested policy is complete regarding the identified security objectives. Then, a policy modeling report that sums up this set of policy is generated (Figure 7.4). This check indicates that the system model is secure with respect to the consumption data asset.

```
policy {
  connector ConnectorGatewayRrcCD {
    restrictiveGetPld restrictiveSetPld lossFreeInject restrictiveSetSrc
  }
  connector ConnectorGatewayRrcReqCD {
    restrictiveGetPld restrictiveSetPld lossFreeInject restrictiveSetSrc
  }

  connector ConnectorGatewayConsumerCD {
    restrictiveGetPld restrictiveSetPld lossFreeInject restrictiveSetSrc
  }
  connector ConnectorGatewayConsumerReqCD {
    restrictiveGetPld restrictiveSetPld lossFreeInject restrictiveSetSrc
  }
}
```

Listing 7.7: Specification of policy of the gateway application for security objectives validation



Policy Report

Created on : Mon Nov 09 15:30:33 CET 2020

Policys model name :
Owner :
Reviewers :
Contributors :

Security policy model summary :

Policy applied : 16
Objective statisfied : 16

Connector : ConnectorGatewayConsumerCD

1. Category : Confidentiality
   Policy : RestrictiveGetPld
   Description: Ensures that all messages going throught **ConnectorGatewayConsumerCD** enforce RestrictiveGetPld.
   Target objective: PayloadConfidentiality
2. Category : Integrity
   Policy : RestrictiveSetPld
   Description: Ensures that all messages going throught **ConnectorGatewayConsumerCD** enforce RestrictiveSetPld.
   Target objective: PayloadIntegrity
3. Category : Authencity
   Policy : RestrictiveSetSrc

Figure 7.4: Policy report (Policies applied to satisfy the objectives in the Gateway application)

7.3 Discussions

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

7.3.1 Applications of the proposed approach

The proposed approach yields a number of useful capabilities for system architects and designers. First, it enables them to find security bugs early in the development of a system. It is well known that correcting software errors and defects, especially those related to system security, at late stages of development is very costly. By employing the proposed approach, system architects and designers can find and take corrective actions before any lines of code are written. Second, it enables them to better understand their security requirements. Good threat modeling approaches can help to ask, "Is this really a requirement?" Ultimately, the proposed approach helps system architects and designers to study the interplay between security requirements, threats, and mitigations. By employing the proposed approach, system architects and designers can determine whether they have a complete set of security requirements that can effectively address the known threats in their systems. Furthermore, the generated threat and objective reports can support security evaluation and assurance activities by providing evidence that systematic analyses have been performed at the Architecture Design phase of the SDLC and that identified threats have been adequately mitigated through the introduction of new security requirements that are satisfied by the modeled software architecture. The need to support security assurance at the architecture design phase has been discussed in [127].

7.3.2 Generalization of the proposed approach

We have demonstrated the practical application of the proposed approach in the context of the Royce waterfall model (SDLC) where we used Microsoft's STRIDE threat classification and CIAA objective classification, first-order and modal logic as a technology-independent specification language, and Alloy as a language with automated tool support for model checking and verification. However, as described in Chapter 3 the proposed approach can be generalized, enabling it to be applied in alternative SDLCs and using other threat or objective classification references, technology-independent specification languages, and model verification tools.

For example, we can envision the proposed approach to be applied in another SDLC, such as an Agile process with additional domain specific modeling tools, where CAPEC or another domain-specific threat model (e.g., IoT-based big data environment [139], MPSoC-based embedded applications [132], etc.) is used as the threat classification reference, temporal logic is used as a technology-independent specification language, and UPPAAL is used as a language with automated tool support for model checking and verification. Such a realization of the proposed approach would simply require a new definition of the DSL using appropriate domain-specific modeling environments, and the formalization of the threats, objectives and the metamodel in the chosen specification languages (e.g., temporal logic and UPPAAL). After applying the approach with the accompanying tool support, the resulting security policies can then be incorporated into stories or scenarios as part of the Agile development process in future sprints.

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

7.3.3 Tool support : automated tool and proof complementarity

In Section 2.6.2, we argue our choices of Alloy and Coq as formal tool for the development of the support tool for our proposed approach. We presented the two selected tools separately for the security objectives in Section 6.8. However, we could imagine a tool suite that combines both tools. In fact, as we stated earlier, Alloy offer issues detection (counter-examples) and better automation, whereas Coq offer a better but more complex verification (proof). Thus, it might be possible to get the best of both. The idea would be to start the design with Alloy as formal tool using its advantage for automation allowing rapid design iterations and issues detection; and, then use Coq as a complement where there is either: a possible doubt about the Alloy scope's sufficiency; or a very critical part in the design that can gain from additional verification through proof validation. As a result, in this vision the two formal tools would be complementary in one approach. Of course, we only suggest here the starting idea; a more precise study should be conducted to determine the feasibility.

Chapter 8

Conclusion & future works

Contents

8.1 Summary and contributions

Security is a quality attribute that is becoming increasingly critical in current software-intensive systems and is thus gaining substantial attention by the research and industrial community. In our work, we proposed an approach to create methodological tool support for security modeling and analysis in the context of component-based software architecture, following a negative (i.e., security threats) and positive (i.e., security objectives) perspectives. The proposed approach for engineering secure systems is dependent on the software architecture model, the underlying communication model, threat and security objectives models as first-class citizens to specify applications within a particular domain and focuses on the problem of software system architecture engineering using a design philosophy that fosters reuse.

This approach may significantly reduce the cost of engineering a system because it enables security issues to be addressed early in the system development process (architecture design) while simultaneously relieving the developer of the technical details. We begin by specifying a conceptual model of the target system and the desired concerns and proceed by the definition of a logical specification of the system architecture, the computing model and concerns as properties using first-order and modal logic as an abstract and technology-independent formalism. We then develop modeling languages and interpretations in tooled formal languages that are appropriate for the targeted design and analysis activities. The results of these efforts are subsequently employed to specify and define a security threat and objectives properties (formal) models and solution (e.g., in the form of a policy). These results illuminate how to build a bridge between these models.

Developing an application using model-based development processes and reusing existing (formal) property model libraries requires finding and tailoring suitable properties to a form that is appropriate for the targeted development environment. The integration of our approach in existing SDLC enables a domain engineer to reuse the resultant model libraries that have been specified and verified and then adapted for a given engineering environment (development platform) to develop a domain-specific application.

In Chapter 3, we proposed a methodology for the creation of a design and analysis framework 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 4, we described a formal framework to support the rigorous design of software architectures focusing on the communication aspects at the architecture level. The framework is based on a component-port-connector meta-model describing the high-level concepts of distributed software architecture supporting message passing communication (answering part of ***RO1***). Then, we formally specified and verified a software architecture based on a set of reusable models, namely connectors (answering part of ***RO2***).

In Chapter 5 and Chapter 6, we respectively proposed an approach to *threat specification, detection, and treatment* and *security objectives specification, detection, and treatment* in component-based software architecture models via reusable security threat, objectives and policies formal model libraries. First, we presented a process of development of reusable formal model libraries for the specification and verification of security threats and objectives by a security expert (answering part of ***RO1***). Second, we presented a process of secure architectural design by an architect based on the libraries previously specified to specify and design secure software architectures (answering part of ***RO2***). Our solution is based on formal techniques for the purposes of precise specification and verification of security aspects as properties of a modeled system.

Furthermore, for each studied architecture concerns (message passing communication, security threats and security objectives), we walked through an MDE-based prototype connected to a tooled formal language to support the proposed approach (answering ***RO3***). An example of this tool suite is constructed using EMFT, Xtext, and Xtend and is currently provided in the form of Eclipse plug-ins that are accompanied by the Alloy Analyser and Coq IDE. The combined formal modeling and MDE to specify security concerns and develop their targeted system security policies allows to develop an accurate analysis, for evaluation and/or certification.

In Chapter 7, we illustrate the application of the proposed approach and the developed tool model the secure software architecture of *Smart Meter Gateway System*, reusing the developed formal model libraries for communication, and security threats and objectives (answering part of ***RO2***). We believe that the result of this part is of practical interest. Basically, our framework and the resulted tool support, aid to illuminate the key ideas to use MDE and formal techniques for the design and analysis of software architectures and security in unison.

8.2 Limitations and future works

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

1. *Risk assessment.* Risk assessment can be improved by automating the identification of system threats. This is possible by reusing the work of software threat analysis. In fact, software threat analysis is used to detect threats at the software architecture design level according to formalized rules. The formalization of rules describes the context in which a certain threat may exploit an absence of one or more security mechanisms, their weakness or their incorrect integration. Therefore, at the level of system architecture, the formalization of system threats may be simplified to the non-existence of security mechanism.

2. *Patterns.* As stated in Section 1.1, one of our next goal is to use this work to improvement our Pattern Based System Engineering (PBSE) framework [48] considering security and safety requirements within software architectures. We plan to transform our PBSE pattern modeling concepts to Alloy specifications to ensure semantic validation. We plan to use our security property modeling approach during the specification and classification of security patterns such as those used in [122], and we intend to add more formality to ensure semantic validation transforming existing Pattern Based System Engineering (PBSE) concepts [32, 48] to Alloy and/or Coq specifications. In addition, we aim at refining our modeling framework with properties and reasoning of the Security Modeling Framework (SeMF) [45]. Our

starting point is modeling security patterns in Alloy from [51]. Moreover, some timing and/or other resource constraints can also be enforced to verify the architecture models.

3. *Tool support.* Another objective for the near future is to continue the development of the tool support described in Section 4.8, Section 5.7 and Section 6.8.

    (a) *Automation.* We would like to improve manual and semi-automated parts of the tool with more automation. For example, we plan to develop the interfaces with the Alloy Analyzer to transfer and to highlight the results of the verification into the DSL model through backward transformation (Alloy model to DSL model) i.e., an automatic (or systematic) incorporation of the results of analysis (report) into the DSL software architecture model.

    (b) *Reuse.* We plan to improve the tool support to cover more aspects of the Krueger [69] fundamental reuse issue. For instance, we seek to handle the storage and the selection of the reusable models [44].

4. *Assessment.* In the near future, we plan to conduct an experiment in which we will present the approach and the solution of our case study to collect feedback from industry practitioners through a survey. In particular, we wanted to assess the perception of using formal techniques coupled with the modeling approaches to engineering secure systems. Duplicate the study to address secure software system development in other domains and perform the survey with other subjects (e.g., students) have been also considered as future work.

5. *Approach Generalization.* We will seek new opportunities to apply the proposed approach to other domains. This task requires an instantiation of the complete software engineering tool and method and an evaluation of the experiences of many users across many domains. We would like to enhance the proposed approaches for the integration with other model-based approaches, architecture models, security models (e.g., the ENISA threat taxonomy [31]), formal techniques (e.g., CTL, NuSMV, Event-B), and software development life cycle (e.g., Agile, Devops).

6. *Interplay of the negative and positive security perspectives.* An important next step is to investigate the interplay of concerns in order to better understand how they may be impacted by one another. This type of investigation is important because it can help us understand how, even if we have a protection (e.g., a policy) in place to protect the system against a threat, this protection may be ineffective if an

adversary is able to pivot to bypass other protections that are in place. Alternatively, a protection may be inadequate because other functional concerns, such as the computation time overhead added by a protection, may impact a system safety requirement. To that end, we intend to investigate: threats interplay, interplay between security objectives and threats, and other relevant non-functional concerns.

(a) *Threats interplay.* With the proposed approach, it may also be possible to further study the relationship between threats that have been detected in the software architecture model to determine whether the existence of one threat facilitates the existence of another. There are a number of methods to implement an attack to realize a given threat, and additionally, an attack is usually the sum of different actions accomplished by the adversary, as in the case of an attack tree. As we have seen in the past, adversaries often will exploit one vulnerability (which leads to a threat) in order to exploit another vulnerability to escalate their reach in the system. This idea is often referred to as *pivoting* or a multilayered attack. For example, an adversary may spoof an identity in order to tamper with information that they would otherwise not have access to. In this case, the fact that it is possible for the adversary to realize the spoofing threat leads to the realization of the tampering threat. Thus, if we are able to determine these relationships, it can allow us to target our mitigation strategies by eliminating the spoofing threat which will subsequently eliminate the tampering threat. The formal setting of the proposed approach provides many essential mechanisms that will enable this kind of reasoning to aid in identifying the root causes leading to the existence or emergence of detected threats. Such a capability is expected to have a major impact on the time and resources required to treat threats in software systems. Of course, we would like to conduct similar investigations for the security objectives.

(b) *Security Objectives & Threats.* Another interesting point would be to investigate the interaction between objectives and threats. As a starting point, we could formalize the links between the STRIDE classification and the objectives classification provided by OWASP [106]. This way, we could expand the meta-model with new associations that are similar to *satisfy* relationships between objectives and policies and *mitigate* relationships between policies and threats. It would allow for a greater complementary between the proposed positive view (objective) and negative view (threat) in our approach. Intuitively, a threat could *violate* an objective. Fulfilling an objective could *protect* against one or multiple threat. As a result, a more comprehensive approach could be proposed.

(c) *Others non-functional concerns.* In addition, interplay with other non-functional aspects such as performance [110] or safety [67] could be considered to manage threats & objectives and to select the appropriate countermeasures. The study interplay of Security and Dependability (S&D) was started in our team in [43] but in a semi-formal way.

8.3 Perspectives

Perspectives emerging from this thesis are manifold. These perspectives are long-term objectives and consist of enhancements related to (1) linking the approach with the implementation phase and (2) managing the implemented and deployed system, and finally (3) enabling security compliance-By-Design.

1. *Implementation phase.* Our approach focused on the architecture design phases. We need to inspect a way to move to the implementation phase without compromising security enforced at prior phases.

2. *Governance risk and compliance.* In this PhD thesis, we have focused on designing secure software architectures through security design and analyzes from the negative and positive perspectives. However, once a software system is implemented and deployed, risks need to be managed. These risks should be linked to an organization's strategy. Governance, Risk and Compliance (GRC) refers to an organization's strategy for managing the broad issues of corporate governance, enterprise risk management (ERM) and corporate compliance with regards to regulations. For instance, Identity and Access Management (IAM) is a kind of governance about managing the life cycle of identities inside an organization (from recruitment to departure) and their impact on the information system.

3. *Model-Based Security Compliance-By-Design.* Currently, there is a need for a systematic way to validate and trace the application and compliance of cybersecurity controls mandated by standards and regulations throughout the lifecycle of large and complex systems. The traceability of this compliance is a common problem in the industry and is expected to become increasingly challenging to address as these systems continue to evolve. In this topic, we will focus specifically on managing the traceability of compliance requirements to the architecture and design in the

context of cybersecurity evaluation. For example, with the trend of Space Cyber Security gaining more traction [11], more of these security controls and standards, which for the most part, is a combination of NIST 800-53 [39] and CNSS Instruction No. 1253 [99] and CNSS Policy 12 [98], will be developed and enforced. Furthermore, because these security controls are regulated by an external party, there is a challenge in tracing compliance of these controls across the developed system throughout its lifecycle. To achieve our general objective and realize the expected project outcomes, we will focus on achieving the following specific objectives:

(a) *Design and specify a security compliance-by-design architecture framework and methodology.* Design and specify a security compliance-by-design architecture framework and methodology for an industrial application: A software architecture defines the organization of the software in terms of configurations of components, connectors capturing the interactions among components, and constraints on the components, connectors and configurations. We target the development of such a configuration of the components as a reference architecture of the target application domain such that it satisfies the security compliance imposed by external third-party regulators (i.e., the security compliance requirements) for the target application domain.

(b) *Develop traceability support for security compliance requirements and the reference architecture components.* Develop traceability support for security compliance requirements and the developed reference architecture components: To support security assurance and compliance efforts, it is important to be able to link the components of the reference architecture developed to the security compliance requirements identified. This linkage is known as traceability and it helps to provide the rationale and justification for where specific security compliance requirements are taken into account in components of the developed architecture. This contextual information enables more effective checking that all of the security compliance requirements been accounted for in the system design.

Bibliography

[1] Alloy Analyzer. `http://alloytools.org/`. [Accessed: October-2019].

[2] Systems and software engineering – Vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, pages 1–418, 2010.

[3] M. Abi-Antoun and J. M. Barnes. Analyzing security architectures. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 3–12, 2010.

[4] M. Abi-Antoun and J. M. Barnes. STRIDE-based security model in Acme. Technical Report CMU-ISR-10-106, Carnegie Mellon University, January 2010.

[5] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge: Cambridge UP, 2010.

[6] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997.

[7] M. Almorsy, J. Grundy, and A. S. Ibrahim. Automated software architecture security risk analysis using formalized signatures. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 662–671, 2013.

[8] C. Atkinson, J. Bayer, and D. 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.

[9] P. Avgeriou, J. Grundy, J. G. Hall, P. Lago, and I. Mistrík, editors. *Relating Software Requirements and Architectures.* Springer, 2011.

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

[11] B. Bailey, R. Speelman, P. Doshi, N. Cohen, and W. Wheeler. Defending spacecraft in the cyber domain. *Aerospace Corp. TR OTR202000016, El Segundo, CA*, 2019.

[12] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual : Version 6.1. Research Report RT-0203, INRIA, May 1997. Projet COQ.

[13] B. J. Berger, K. Sohr, and R. Koschke. Extracting and analyzing the implemented security architecture of business applications. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 285–294, 2013.

[14] J. Bergstra and J. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[15] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science An EATCS Series. Springer Berlin/Heidelberg, 2004.

[16] L. Bettini. *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition.* Packt Publishing, 2nd edition, 2016.

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

[18] 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.

[19] T. Bures and F. Plasil. Communication style driven connector configurations. In C. V. Ramamoorthy, R. Lee, and K. W. Lee, editors, *Software Engineering Research and Applications*, pages 102–116. Springer Berlin Heidelberg, 2004.

[20] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8, 1990.

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

[22] O. Consortium. Joram: Java open reliable asynchronous messaging. `https://joram.ow2.io/`, 2018. [Accessed: April-2019].

[23] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design.* Addison-Wesley Publishing Company, 5th edition, 2011.

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

[25] 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.

[26] D. D. Garlan. Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events. In *Third International School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures*, pages 1–24. Springer Berlin Heidelberg, 2003.

[27] 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.

[28] D. F. D'Souza and A. C. Wills. *Objects, components, and frameworks with UML : the catalysis approach.* Addison-Wesley Professional, 1998.

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

[30] EU. Regulatin (eu) 2016/679 (general data protection regulation. `https://gdpr-info.eu`, 2018. [Accessed: February-2019].

[31] 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].

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

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

[34] A. Fuchs, S. Gürgens, and C. Rudolph. Formal Notions of Trust and Confidentiality - Enabling Reasoning about System Security. *Journal of Information Processing*, 19:274–291, 2011.

[35] D. Garlan, R. T. Monroe, and D. Wile. Acme: An architecture description interchange language. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '97, pages 7–22. IBM Press, 1997.

[36] I. Graham, B. Henderson-Sellers, and H. Younessi. *The OPEN Process Specification.* ACM Press/Addison-Wesley Publishing Co., 1997.

[37] H. Grandy, D. Haneberg, W. Reif, and K. Stenzel. Developing Provable Secure M-Commerce Applications. In *Emerging Trends in Information and Communication Security*, pages 115–129. Springer, Berlin, Heidelberg, 2006.

[38] 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.

[39] J. T. F. I. W. Group. Security and privacy controls for information systems and organizations. special publication (nist sp) 800-53 revision 5 (draft), national institute of standards and technology, 2020.

[40] C. Haley, R. Laney, J. Moffett, and B. Nuseibeh. Security requirements engineering: A framework for representation and analysis. volume 34, pages 133–153. IEEE, 2008.

[41] J. G. Hall, M. Jackson, R. C. Laney, B. Nuseibeh, and L. Rapanotti. Relating software requirements and architectures using problem frames. In *IEEE Joint International Conference on Requirements Engineering*, pages 137–144. IEEE, 2002.

[42] B. Hamid. SEMCO Project (System and software Engineering with Multi-COncerns support). http://www.semcomdt.org, 2009.

[43] B. Hamid. Interplay of Security&Dependability and Resource Using Model-Driven and Pattern-Based Development. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 254–262. IEEE, 2015.

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

[45] B. Hamid, S. Gürgens, and A. Fuchs. Security patterns modeling and formalization for pattern-based development of secure software systems. *Innovations in Systems and Software Engineering, Springer*, 12(2):109–140, 2016.

[46] B. Hamid and J. Perez. Supporting Pattern-Based Dependability Engineering via Model-Driven Development: Approach, tool-support and empirical validation. *Journal of Systems and Software, Elsevier*, 122:239–273, 2016.

[47] B. Hamid, Q. Rouland, and J. Jaskolka. Distributed maintenance of a spanning tree of k-connected graphs. In *2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 209–217. IEEE, 2019.

[48] B. Hamid and D. Weber. Engineering secure systems: Models, patterns and empirical validation. *Computers & Security*, 77:315–348, 2018.

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

[50] C. L. Heitmeyer. Applying practical formal methods to the specification and analysis of security properties. In V. I. Gorodetski, V. A. Skormin, and L. J. Popyack, editors, *Information Assurance in Computer Networks*, volume 2052 of *Lecture Notes in Computer Science*, pages 84–89. Springer Berlin/Heidelberg, 2001.

[51] T. Heyman, R. Scandariato, and W. Joosen. Reusable formal models for secure software architectures. In *Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, pages 41–50, 2012.

[52] T. Heyman, K. Yskout, R. Scandariato, H. Schmidt, and Y. Yu. The security twin peaks. In *Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSoS)*, volume LNCS 6542 of *Lecture Notes in Computer Science*, pages 167–180. Springer, 2011.

[53] C. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, 1978.

[54] C. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent Kleene algebra and its foundations. *Journal of Logic and Algebraic Programming*, 80(6):266–296, 2011.

[55] S. Hussain, H. Erwin, and P. Dunne. Threat modeling using formal methods: A new approach to develop secure web applications. In *Proceedigns of the 7th International Conference on Emerging Technologies*, pages 1–5, September 2011.

[56] 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.

[57] D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[58] D. Jackson. Alloy: A language and tool for exploring software designs. *Commun. ACM*, 62(9):66–76, Aug. 2019.

[59] D. Jackson. Alloy: A language and tool for exploring software designs. *Commun. ACM*, 62(9):66–76, 2019.

[60] M. Jan, C. Jouvray, F. Kordon, A. Kung, J. Lalande, F. Loiret, J. F. Navas, L. Pautet, J. Pulou, A. Radermacher, and L. Seinturier. Flex-eware: a flexible model driven solution for designing and implementing embedded distributed systems. *Softw., Pract. Exper.*, 42(12):1467–1494, 2012.

[61] J. Jaskolka, R. Khedri, and Q. Zhang. Endowing concurrent Kleene algebra with communication actions. In P. Höfner, P. Jipsen, W. Kahl, and M. Müller, editors, *Proceedings of the 14th International Conference on Relational and Algebraic Methods in Computer Science*, volume 8428 of *Lecture Notes in Computer Science*, pages 19–36. Springer International Publishing Switzerland, 2014.

[62] J. 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.

[63] J. Jürjens. UMLsec: Extending UML for Secure Systems Development. In J.-M. Jézéquel, H. H. smann, and S. 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.

[64] J. Jürjens. UMLsec: Extending UML for Secure Systems Development. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, UML '02, pages 412–425, London, UK, 2002. Springer-Verlag.

[65] 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.

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

[67] J. C. Knight. Safety critical systems: challenges and directions. In *Proceedings of the 24th international conference on software engineering*, pages 547–550, 2002.

[68] P. Kruchten. Architectural blueprints - the "4+ 1" view model of software architecture. *IEEE Software*, 12(6):42–50, 1995.

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

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

[71] C. E. Landwehr. Formal Models for Computer Security. *ACM Computing Surveys*, 13:247–278, 1981.

[72] Y. Ledru, A. Idani, J. Milhau, N. Qamar, R. Laleau, J. Richier, and M. Labiadh. Validation of is security policies featuring authorisation constraints. *Int. J. Inf. Syst. Model. Des.*, 6(1):24–46, 2015.

[73] Y. Lee, J. Lee, and Z. Lee. Integrating Software Lifecycle Process Standards with Security Engineering. *Computers & Security*, 21(4):345–355, 2002.

[74] Y. Lee, Z. Lee, and C. K. Lee. A study of integrating the security engineering process into the software lifecycle process standard (IEEE/EIA 12207). *AMCIS 2000 Proceedings*, page 182, 2000.

[75] T. Lodderstedt, D. A. Basin, and J. 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.

[76] A. Mana and G. Pujol. Towards formal specification of abstract security properties. In *Proceedings of the Third International Conference on Availability, Reliability and Security*, pages 80–87, March 2008.

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

[78] G. McGraw. The security lifecycle-the 7 touchpoints of secure software-just as you can't test quality into software, you can't bolt security features onto code and expect it to become hack-proof security. In *Software Development*, volume 13, pages 42–43, 2005.

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

[80] D. Mellado, C. Blanco, L. E. Sánchez, and E. Fernández-Medina. A systematic review of security requirements engineering. *Computer Standards & Interfaces*, 32(4):153–165, June 2010.

[81] Microsoft. The STRIDE threat model. Microsoft Corporation, 2009.

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

[83] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[84] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes part I. *Information and Computation*, 100(1):1–40, September 1992.

[85] MITRE Corporation. Common attack pattern enumeration and classification (capec). `http://capec.mitre.org/`, 2018. [Accessed: January-2019].

[86] MITRE Corporation. Common weakness enumeration (cwe). `https://cwe.mitre.org/`, 2018. [Accessed: January-2019].

[87] 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.

[88] P. Mohagheghi and V. 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.

[89] R. T. Monroe. Capturing software architecture design expertise with armani. Technical Report CMU-CS-98-163, Carnegie Mellon University, October 1998.

[90] M. S. Nawaz and M. Sun. Reo2PVS: Formal specification and verification of component connectors. In *Proceedings of the 30th International Conference on Software Engineering and Knowledge Engineering*, SEKE 2018, pages 390–395, 2018.

[91] OMG. CORBA Specification, Version 3.1. Part 3: CORBA Component Model. `http://www.omg.org/spec/CCM`, 2008. [Accessed: November-2009].

[92] OMG. OMG Systems Modeling Language (OMG SysML), Version 1.1. http://www.omg.org/spec/SysML/1.1/, 2008. [Accessed: January-2013].

[93] OMG. Object Constraint Language (OCL), Version 2.2. http://www.omg.org/spec/OCL/2.2, 2010. [Accessed: January-2013].

[94] OMG. UML profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE), Version 1.1. http://www.omg.org/spec/MARTE/1.1/, 2011. [Accessed: January-2013].

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

[96] OMG. Precise Semantics of UML Composite Structures, Version 1.2. https://www.omg.org/spec/PSCS/1.2, 2019. [Accessed: September-2019].

[97] OMG. Unified Component Model for Distributed, Real-Time And Embedded Systems, Version 1.1. https://www.omg.org/spec/UCM/1.1, 2019. [Accessed: April-2019].

[98] C. on National Security Systems. National information assurance policy for space systems used to support national security missions. cnss policy 12, committee on national security systems, 2012.

[99] C. on National Security Systems. Security categorization and control selection for national security systems. cnss instruction no. 1253, committee on national security systems, 2014.

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

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

[102] Oracle. Javaspaces service specification. https://river.apache.org/release-doc/current/specs/html/js-spec.html, 2005. [Accessed: April-2019].

[103] Oracle. Java message service. https://javaee.github.io/jms-spec/, 2015. [Accessed: April-2019].

[104] Oracle. Java remote method invocation specification. https://docs.oracle.com/javase/9/docs/specs/rmi/index.html, 2017. [Accessed: April-2019].

[105] OWASP. OWASP CLASP V.A.2. Technical report, Nov. 2007.

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

[107] L. Paulson. Proving Properties of Security Protocols by Induction. Technical Report 409, Computer Laboratory, University of Cambridg, 1996.

[108] R. A. R and D. Dill. A Theory of Timed Automata. *heoretical Computer Science*, 12(6):183–235, 1994.

[109] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady. Security in embedded systems: Design challenges. *ACM Trans. Embed. Comput. Syst.*, 3(3):461–491, 2004.

[110] A. Richter, C. Herber, T. Wild, and A. Herkersdorf. Denial-of-service attacks on PCI passthrough devices: Demonstrating the impact on network- and storage-I/O performance. *Journal of Systems Architecture*, 61(10):592–599, 2015.

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

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

[113] Q. Rouland, B. Hamid, J.-P. Bodeveix, and M. Filali. A formal methods approach to security requirements specification and verification. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 236–241. IEEE, 2019.

[114] Q. Rouland, B. Hamid, and J. Jaskolka. Formalizing reusable communication models for distributed systems architecture. In *International Conference on Model and Data Engineering (MEDI)*, pages 198–216. Springer, 2018.

[115] Q. Rouland, B. Hamid, and J. Jaskolka. Formal specification and verification of reusable communication models for distributed systems architecture. *Future Generation Computer Systems*, 108:178–197, 2020.

[116] Q. Rouland, B. Hamid, and J. Jaskolka. Reusable formal models for threat specification. In *ICSR 2020: Reuse in Emerging Software Engineering Practices*, pages 52–68. Springer, 2020.

[117] Q. Rouland, B. Hamid, and J. Jaskolka. Specification, detection, and treatment of stride threats for software components: Modeling, formal methods, and tool support. *Journal of Systems Architecture*, 2021.

[118] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering*, pages 328–338. IEEE Computer Society Press, 1987.

[119] N. Rozanski and E. Woods. *Software Systems Architecture.* Addison Wesley, 2 edition, 2011.

[120] SAE. Architecture Analysis & Design Language (AADL). `http://www.sae.org/technical/standards/AS5506A`, 2009. [Accessed: January-2011].

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

[122] M. Schumacher. *Security Engineering with Patterns - Origins, Theoretical Models, and New Applications*, volume 2754 of *Lecture Notes in Computer Science.* Springer, ISBN 978-3-540-45180-8, 2003.

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

[124] D. Sgandurra, E. Karafili, and E. Lupu. Formalizing threat models for virtualized systems. In S. Ranise and V. Swarup, editors, *Proceedings of Data and Applications Security and Privacy XXX*, volume 9766 of *Lecture Notes in Computer Science*, pages 251–267. Springer International Publishing, 2016.

[125] M. Shin, H. Gomaa, and D. Pathirage. A software product line approach for feature modeling and design of secure connectors. In *Proceedings of the 13th International Conference on Software Technologies*, ICSOFT 2018, pages 506–517, 2018.

[126] G. Sindre and A. 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.

[127] I. Šljivo, G. J. Uriagereka, S. Puri, and B. Gallina. Guiding assurance of architectural design patterns for critical applications. *Journal of Systems Architecture*, 110:101765, 2020.

[128] R. M. Smullyan. *First-order logic.* Courier Corporation, 1995.

[129] J. Spivey. *The Z notation.* Prentice-Hall, 1989.

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

[131] C. Szyperski. *Component Software: Beyond Object-oriented Programming.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[132] B. Tan, M. Biglari-Abhari, and Z. Salcic. Towards decentralized system-level security for MPSoC-based embedded applications. *Journal of Systems Architecture*, 80:41–55, 2017.

[133] M. Torngren, D. 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, 2005.

[134] H. Van Ditmarsch, W. van Der Hoek, and B. Kooi. *Dynamic epistemic logic*, volume 337. Springer Science & Business Media, 2007.

[135] J. Viega. Building Security Requirements with CLASP. In *Proceedings of the 2005 Workshop on Software Engineering for Secure Systems—Building Trustworthy Applications*, SESS ’05, pages 1–7. ACM, 2005.

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

[137] G. H. Von Wright. Deontic logic. *Mind*, 60(237):1–15, 1951.

[138] F. Wagner. *Modeling software with finite state machines: a practical approach.* Auerbach Publications, 2006.

[139] M. Wazid, A. K. Das, R. Hussain, G. Succi, and J. J. Rodrigues. Authentication in cloud-driven IoT-based big data environment: Survey and outlook. *Journal of Systems Architecture*, 97:185–196, 2019.

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

[141] R. Zurawski. Embedded Systems in Industrial Applications - Challenges and Trends. In *International Symposium on Industrial Embedded Systems (SIES).* IEEE, 2007.

Appendices

Appendix A

Architecture : Additional communication paradigms RPC & DSM

A.1 Scenario view

A.1.1 Communication behavior semantics

A.1.1.1 Remote procedure call

In the typical remote procedure call (RPC) communication style [23], a channel is used for sending invocation (request) messages from a client to a server and for receiving acknowledgement (reply) messages from a server to a client. The communication channel is modeled as a queue of fixed length for both request and reply messages from a client and a server respectively. Note that RPC is a special case of the general message passing model.

Figure A.1 shows the states of a client for sending invocation messages and receiving reply messages. It is shown that if the state for send is sending and the buffer is not full, it changes its state from 0 (invocation sent) to 1 for waiting for a reply. On the other hand, if the state is sending and the buffer is full, it remains at state 0. It is also shown that if the state is 1 (waiting to receive a reply) and the reply is in the buffer, it changes its state from 1 to 2 for receiving a reply. Otherwise, if the reply is not yet in the buffer, it remains at state 1. On the other hand, if the state is receiving and the reply is not in the buffer, it changes its state from 2 to 0.

Figure A.2 shows the states of a connector for pushing and pulling invocation and

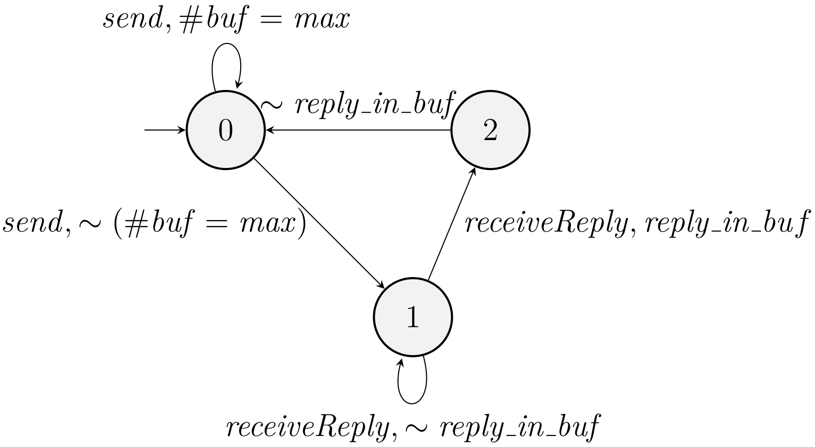


Figure A.1: States of a client for invocation/receiving reply messages

reply messages. It is shown that if the state is 0 (waiting to receive from the caller) and a push of an invocation occurred, it changes its state from 0 to 1. The connector stays in the state 1 (retrieving an invocation to the receiver) until a pull of an invocation which changes its state from 1 to 2 for waiting for a reply. Figure A.2 also shows also that the connector remains in this new state until a push of a reply occurs then it changes its state from 2 to 3 indicating that it is receiving a reply. Finally, it changes its state from 3 to 0 if a pull of a reply occurred.

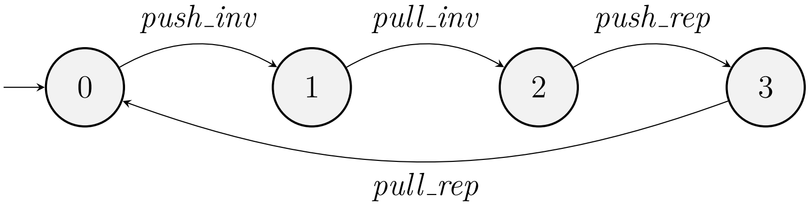


Figure A.2: States of a RPC connector

Figure A.3 shows the states of a server for receiving invocation and sending reply messages. It is shown that if the state is waiting to receive an invocation and an event *receiveInvocation* occurs in the case that an invocation is present in the buffer it changes it state from 0 to 1. Otherwise, if the invocation is absent it remains in the same state. It also shows that after the execution of an event reply, if the buffer is not full it changes its state from 1 to 2. Otherwise, if the buffer is full it stays in state 1. Finally, from state 2 it returns to state 0 when a reply is in the buffer.

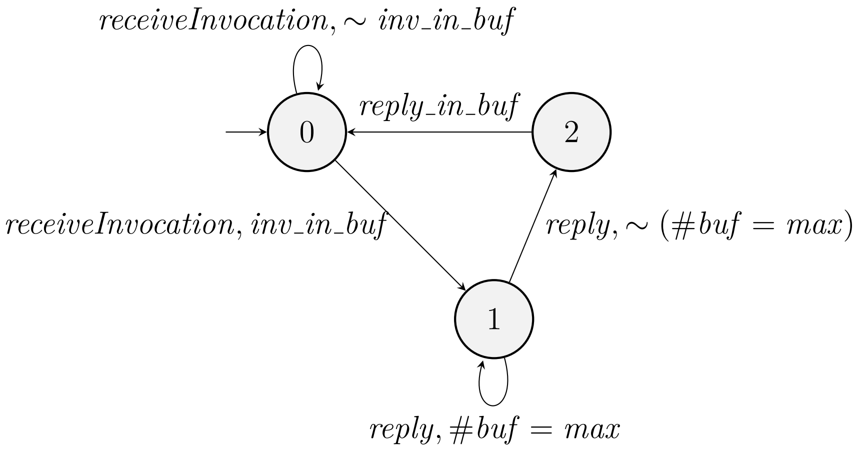

receiveInvocation, ∼ inv_in_buf
reply_in_buf
0
2
receiveInvocation, inv_in_buf
reply, ∼ (#buf = max)
1
reply, #buf = max

Figure A.3: States of a server for receiving invocation/sending reply message

A.1.1.2 Distributed shared memory

In the distributed shared memory (DSM) communication style, a Central Memory Manager keeps the state of the different shared variables and uses an RPC communication model invocation to access (write/read operation) these variables through invocation (request) from a client. The Central Memory Manager reply is either the variable value in the case of a read operation or an acknowledgement in the case of a write operation. So, we can see DSM as an abstraction of the RPC model, hiding the complexity to access shared variables. Note that this means there no need to define a new connector for the DSM communication style, but only to define specific behavior and operations for the clients and the Central Memory Manager (server) on top of RPC.

Figure A.4 shows the states of a client for writing and reading a shared variable. It shows that if the client is waiting to read or write a variable value, a state change can happen either when a read invocation occurred and the buffer is not full or a write invocation occurred and the buffer is not full. In the case of a read invocation, its state changes from 0 (invocation sent) to 1, indicating that it is waiting for a reply that contains the variable value. Otherwise, in the case of a write invocation, its state changes from 0 (invocation sent) to 2, indicating that it is waiting for a reply that contains an acknowledgement that the change of value for the variable is effective in the Central Memory Manager. On the other hand, if the state is sending and the buffer is full, its state remains at 0. It is also shown that if the state is 1 (waiting to receive a reply that contains the variable value) and the reply is in the buffer, it changes its state from 1 to 3 for receiving this reply. Otherwise, if the reply is not yet in the buffer, it remains at state 1. In a similar way, if

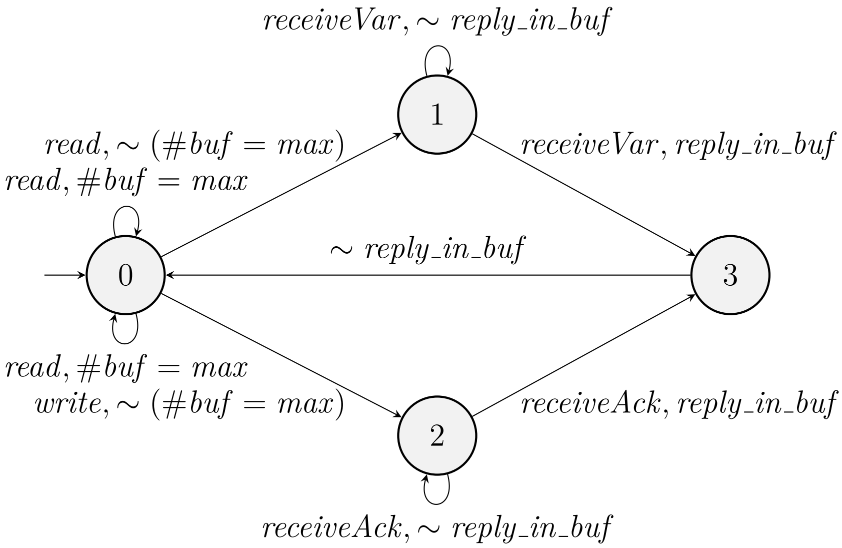


Figure A.4: States of a client for writing and reading a shared variable

the state is 2 (waiting to receive a reply that contains the acknowledgement of the success of the write operation) and the reply is in the buffer, it changes its state from 2 to 3 for receiving this reply. If the reply is not yet in the buffer, it remains at state 2. Finally, if the state is receiving and the reply is not in the buffer, it changes its state from 3 to 0.

Figure A.5 shows the states of the Central Memory Manager (server) for receiving write/read calls and sending the according reply messages. It is shown that if the Central Memory Manager is waiting to receive a read or write invocation, and if an event *receiveRead* occurs when an invocation is present in the buffer, it changes it state from 0 to 1. Otherwise, if an event *receiveWrite* occurs when an invocation is present in the buffer, it changes its state from 0 to 2. On the other hand, if either a read or write invocation is absent, it remains in the same state 0. It is also shown that if the Central Memory Manager is handling a write invocation and it updates the memory according to corresponding write invocation, it changes its state from 2 to 3. We can observe that the transition from 1 to 4 occurs when the Central Memory Manager is sending the reply to a read operation and the buffer is not full. In a similar way, the transition from state 3 to 4 occurs when the Central Memory Manager is sending the reply to a write operation and the buffer is not full. In either case, if its state is 1 or 3, and the buffer is not full, its state remains unchanged. Finally, from state 3 the Central Memory Manager returns to state 0 when a reply is not in the buffer.

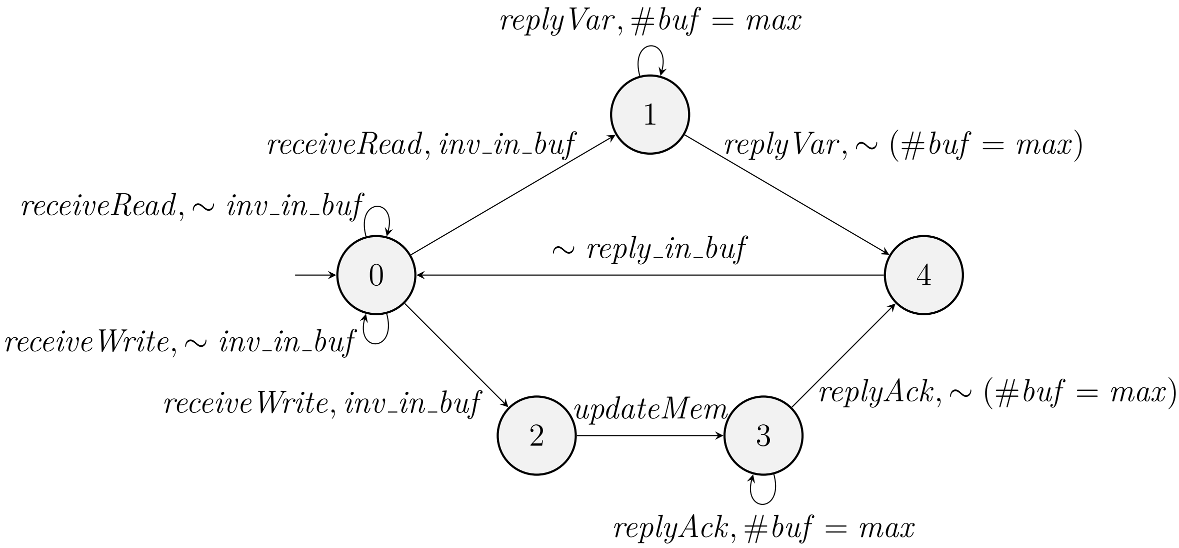


Figure A.5: States of the Central Memory Manager (server) receiving reading/writing calls and returning the corresponding reply messages

A.1.2 Communication paradigms properties specification

A.1.2.1 Remote procedure call

Once the caller $c1$ sends an invocation to callee $c2$, the caller eventually receives an acknowledgement from that callee.

$$\mathbb{H}_{c1}(\mathit{call}(c2, m, \mathit{args\_in}, \mathit{args\_out})) \ \rightsquigarrow \ \mathbb{H}_{c1}(\mathit{executeReply}(c2, m, \mathit{args\_in}, \mathit{args\_out})) \tag{A.1}$$

Once the caller $c1$ receives results corresponding to an invocation of a method m at a certain server $c2$ and the caller $c1$ starts the next invocation of the same method at the same server, the callee $c1$ is eventually executing that invocation.

$$\begin{aligned}(&\mathbb{H}_{c2}(\mathit{reply}(c1, m, \mathit{args\_in}1, \mathit{args\_out}2)) \ < \ \mathbb{H}_{c1}(\mathit{call}(c2, m, \mathit{args\_in}2, \mathit{args\_out}2))) \\ &\rightsquigarrow \ \mathbb{H}_{c1}(\mathit{executeReply}(c2, m, \mathit{args\_in}, \mathit{args\_out}))\end{aligned} \tag{A.2}$$

A.1.2.2 Distributed shared memory

Once a caller c reads from shared variable var, eventually it gets a value $value$ for var.

$$\forall\ c \in \mathcal{C}, d \in \mathcal{D}, typ \in \mathcal{T}\ \cdot\ \mathbb{H}_c(read(var, typ))\ \rightsquigarrow\ \mathbb{H}_c(executeRead(var, value)) \tag{A.3}$$

Once a caller c writes a value *value* in shared variable *var*, eventually it gets an acknowledgement that the operation is successful.

$$\forall\ c \in \mathcal{C}, d \in \mathcal{D}, typ \in \mathcal{T}\ \cdot\ \mathbb{H}_c(write(var, typ))\ \rightsquigarrow\ \mathbb{H}_c(executeWrite(var, value)) \tag{A.4}$$

Once a caller $c1$ writes a value *value* in a shared variable *var* and no others client $c3$ writes any new value $value'$ in *var*, eventually a client $c2$ getting *value* by reading *var*.

$$\begin{aligned}&(\mathbb{H}_{c_1}(executeWrite(var, value))\ \wedge\\ &\quad \neg(\mathbb{H}_{c_3}(executeWrite(var, value))\ <\ \mathbb{H}_{c_2}(executeRead(var, value)))\\ &\qquad \Rightarrow \mathbb{H}_{c_2}(executeRead(var, value))\end{aligned} \tag{A.5}$$

Example of communication properties specification. In the example of the college library web application described in Section 2.8.1, we can identify the corresponding communication semantics that will need to be used in order to fulfill the desired requirements:

- *Req_1.* The remote procedure call communication style allows us to express *Req_1*. As identified in Section 4.5, two components (*Browser* and *Website*) and a connector are in play in this requirement. To satisfy the requirement, the *Browser* must respect the semantics of an RPC client as described in Figure A.1 and the *Website* must respect the semantics of an RPC server as described in Figure A.3. Finally, the connector must respect the semantics of an RPC connector as described in Figure A.2.

- *Req_3.* The distributed shared memory communication style allows us to express *Req_3*. As identified in Section 4.5, two components (*Terminal* and *Database*) and a connector have roles in satisfying this requirement. The *Terminal* must respect the semantics of a DSM client as described in Figure A.4 and the *Website* must respect the semantics of a DSM server (Central Memory Manager) as described in Figure A.5. Finally, the connector must respect the semantics of an RPC connector as described in Figure A.2.

A.2 Meta-Model

A.2.1 Formalizing the software architecture metamodel

(a) “For two components $c1$ and $c2$ to interact using invocation, each must uses a port that realizes the correct *Interface* offering the same *Method m*”

```
pred  invocation_type_check[c1,c2:Component, m:Method] {
  all i:Invocation |
    i.invocation_of = m and c1 = i.client and c2 = i.server =>
    some p1:c1.uses, p2:c2.uses {
      p1.realizes.methods = m
      p2.realizes.methods = m
      p1.realizes.kind = PROVIDED
      p2.realizes.kind = REQUIRED
    }
}
```

The Alloy Analyzer shows that properties (a) hold.

A.2.2 Formalizing and verifying connectors and their properties

A.2.2.1 Remote procedure call

As with the message passing connectors, the RPC connector is also defined as a buffer of invocation (request) messages or acknowledgement (reply) messages. Listing A.1 shows an excerpt of the formalization of the RPC connector. As described in Section A.1.1.1, invocation (request) messages are sent from a client to a server and acknowledgement (reply) messages are received by a client from a server. Associated with these invocations are *push* and *pull* operations which support the high-level communication primitives. For example, when the client component makes a remote procedure call, an invocation is buffered in the connector (line 17). When the server component executes the procedure call, the Invocation is removed from the connector (line 21). A similar definition is provided when the server component acknowledges the execution of a procedure call with a reply that is received by the client component (lines 25 and 29). A fact called *traces* is defined to ensure the state transitions of the RPC connector form only valid executable traces (lines 32 to 36).

```
1  sig ConnectorRPC extends Channel {
2    buffer : Invocation lone -> Tick
3  }
```

```
pred RPC_Init[t: Tick] {
  all c:ConnectorRPC | c.buffer.t = none
}
pred RPC_push [t, t': Tick, c:ConnectorRPC, i:Invocation] {
  c.buffer.t = none
  c.buffer.t' = i
}
pred RPC_pull [t, t': Tick, c: ConnectorRPC, i:Invocation] {
  c.buffer.t = i
  c.buffer.t' = none
}
pred RPC_pushInvocation  [t, t': Tick, c:ConnectorRPC, i:Invocation] {
  #c.buffer.t.arguments_reply.t = 0
  RPC_push[t,t',c,i]
}
pred RPC_pullInvocation  [t, t': Tick, c:ConnectorRPC, i:Invocation] {
  #c.buffer.t.arguments_reply.t = 0
  RPC_pull[t,t',c,i]
}
pred RPC_pushReply  [t, t': Tick, c:ConnectorRPC, r:Invocation] {
  c.buffer.t.arguments_reply.t' != 0
  RPC_push[t,t',c,r]
}
pred RPC_pullReply  [t, t': Tick, c:ConnectorRPC, r:Invocation] {
  #c.buffer.t.arguments_reply.t != 0
  RPC_pull[t,t',c,r]
}
fact traces {
  RPC_Init[TO/first]
  all t:Tick - TO/last | let t' = TO/next[t] |
  some c:ConnectorRPC, i:Invocation, r:Invocation
  | RPC_pushInvocation [t, t', c, i] iff not RPC_pullInvocation [t, t', c, i]
  iff not RPC_pushReply [t, t', c, r] iff not RPC_pullReply [t, t', c, r]
}
```

Listing A.1: Remote procedure call connector

Communication in the remote procedure call communication style is performed using the *call*(), *executeCall*(), *reply*() and *executeReply*() primitives. As depicted in Listing A.2, the *call*() primitive executed at the caller component (line 5) requires the name of the callee component providing the invoked method, the method being invoked, and the associated arguments as parameters. The *executeCall*() primitive (line 17) requires the name of the anticipated caller component, the corresponding invoked method, and its input and output arguments. The *reply*() primitive (line 30) requires the name of the anticipated caller component, the corresponding invoked method, and its result parameters. The *executeReply*() primitive (line 43) requires the name of the anticipated callee component, the corresponding invoked method, and its result parameters.

The semantics of RPC in distributed systems are the same as those of a local procedure call in non-distributed systems: The caller component calls and passes input arguments to the remote procedure and it blocks at the *call*(*callee*, *method*, *input*, *result*) while the remote procedure executes (*executeCall*(*caller*, *method*, *input*, *result*)). When the remote procedure completes, the callee component can return result parameters to the calling component (*reply*(*caller*, *method*, *result*)) and the caller becomes unblocked and continues its execution (*executeReply*(*callee*, *method*, *result*)). As a prerequisite, we added the *check_type_interaction_interface* predicate (lines 2 to 3) to ensure that operations are present at the sending and receiving components before an invocation is executed. Without interface type checking, the presence of the invoked operation is only verified at execution time.

```
 1  pred check_type_interaction_interface[i:Invocation]{
 2    one if:Interface, p:i.client.uses | if in p.realizes and if.kind =
        REQUIRED and  i.invocation_of in if.methods
 3    one if:Interface, p:i.server.uses | if in p.realizes and if.kind =
        PROVIDED and i.invocation_of in if.methods
 4  }
 5  pred Component.H_call[callee:Component, meth: Method, in: set Argument,
      out:set Argument, t:Tick]{
 6    some i : Invocation {
 7      i.client = this
 8      i.server = callee
 9      i.invocation_of = meth
10      i.arguments_call = in
11      #i.arguments_reply.t = 0
12      check_type_interaction_interface[i]
13      one t':t.next | let c = { c:ConnectorRPC | c.portO in i.client.uses
14      and c.portI in i.receiver.uses} | RPC_pushInvocation[t,t',c,i]
15    }
16  }
17  pred Component.H_executeCall[caller:Component, meth: Method, in: set
      Argument, out:set Argument, t:Tick] {
18    some i : Invocation {
19      i.client = caller
```

```
    i.server = this
    i.invocation_of = meth
    i.arguments_call = in
    #i.arguments_reply.t = 0
    check_type_interaction_interface[i]
    one t':t.next | let c = { c:ConnectorRPC | c.portO in i.client.uses
    and c.portI in i.server.uses}
    | RPC_pullInvocation[t,t',c,i] and c.buffer.t'.arguments_reply.t' =
       args_out
  }
}
pred Component.H_reply[caller:Component, meth: Method, out:setArgument,
   t:Tick] {
  some r : Invocation {
    r.client = caller
    r.server = this
    i.invocation_of = meth
    i.arguments_call = in
    i.arguments_reply = out
    check_type_interaction_interface[i]
    one t':t.next | let c = { c:ConnectorRPC | c.portO in r.client.uses
    and c.portI in r.server.uses}
    | RPC_pushReply[t,t',c,r]
  }
}
pred Component.H_executeReply[callee:Component, meth: Method, in: set
   Argument,out:set Argument, t:Tick] {
  some r : Invocation {
    r.client = this
    r.server = callee
    i.invocation_of = meth
    i.arguments_call = in
    i.arguments_reply = out
    check_type_interaction_interface[i]
    one t':t.next | let c = { c:ConnectorRPC | c.portO in r.client.uses
    and c.portI in r.server.uses}
    | RPC_pullReply[t,t',c,r]
  }
}
```

Listing A.2: Remote procedure call communication

Among the set of possible and specified characteristics of the behaviors of the remote procedure call communication style, a subset of them are encoded in terms of properties as predicates and assertions and the results of their verification are stated below.

(a) “Once the caller $c1$ sends an invocation to callee $c2$, the caller eventually receives an acknowledgement from that callee.”

```
pred send_is_eventually_replied[c1,c2:Component, m:Method, args_in, args_out:Argument] {
  one t:Tick | one t':t.nexts |
    c1.H_call[c2,m,args_in,args_out,t] => c1.H_executeReply[c2,m,args_in,args_out,t']
}
```

(b) “Once the caller $c1$ receives results corresponding to an invocation of a method m at a certain server $c2$ and the caller $c1$ starts the next invocation of the same method at the same server, the callee $c1$ is eventually executing that invocation.”

```
pred reply_and_call_is_eventually_received[c1,c2:Component, m:Method, args_in,args_out:Argument] {
  one t:Tick | one t':t.nexts | one t'':t'.next |
    c2.H_reply[c1,m,args_in1,args_out1,t] and c1.H_call[c2,m,args_in2,args_out2,t'] =>
    c2.H_executeCall[c1,m,args_in2,args_out2,t'']
}
```

The Alloy Analyzer shows that both properties (a) and (b) hold for an RPC connector.

A.2.2.2 Distributed shared memory

The DSM connector is defined as an extension of the RPC connector described in Section A.2.2.1. Conforming to the behavioral semantics specified in Section A.1.1.2, this extension further specifies the invocation methods to be specifically read or write operations. Listing A.3 shows the formalization of the DSM connector as an extension of the RPC connector.

```
sig ConnectorSM extends ConnectorRPC{} {
  all t:Tick, i:Invocation | i in buffer.t => i.invocation_of in Read + Write
}
```

Listing A.3: Distributed shared memory connector

Communication in the distributed shared memory style is performed using the *read*(), *executeRead*(), *write*() and *executeWrite*() primitives. As depicted in Listing A.4, the *write*() primitive (line 55) requires the identification of the variable that the caller want to access and the value it wants to assign to this variable. The *executeWrite*() primitive (line 58) requires the name of the anticipated new value of the variable and the corresponding identification of this variable. The *read*() primitive (line 61) executed at the caller component requires the name of the identification of the variable that the caller want to access. The *executeRead*() primitive (line 64) requires the value of the anticipated variable being read and the corresponding identification of this variable.

As the proposed distributed shared memory style is based on RPC, the semantics is related to the same mechanism: The caller component calls the write or read operation and passes input arguments and execute a remote procedure call to Central Memory Manager. It blocks at the *read*(*variable*) / *write*(*variable*, *value*) while the remote procedure executes on the Central Memory Manager (*receiveRead*(*caller*, *variable*, *value*) / *receiveWrite*(*caller*, *variable*, *value*)). When the remote procedure completes, the callee (Central Memory Manager) can return result parameters to the calling component (*replyVar*(*caller*, *variable*, *value*) / *replyAck*(*caller*, *variable*, *value*)) and the caller becomes unblocked and continues its execution (*executeRead*(*variable*, *value*) / *executeWrite*(*variable*, *value*)).

```
1  abstract one sig CentralMemoryManager extends Component {
2    mem: Res -> Tick
3  }{
4    all t:Tick | no disj r,r':Res | r in mem.t  and r' in mem.t and some r
       .variable & r'.variable
5  }
6  sig Res {
7    variable: Var,
8    value:Value
9  }
10 sig Read extends Method {}
11 sig Write extends Method {}
12 sig Var extends Argument {}
13 sig Value extends Argument {}
14 sig ACK extends Argument {}
15 fact traces {
16   CentralMemoryManager.init[TO/first]
17   all t:Tick - TO/last | let t' = TO/next[t] |
18     some c:Component, var:Var, value:Value
19     | CentralMemoryManager.receiveWrite [t,t',c,var,value]
```

```
    iff not CentralMemoryManager.receiveRead [t, t',c,var,value]
    iff not CentralMemoryManager.updateMem [t, t',c,var,value]
    iff not CentralMemoryManager.replyVar [t, t',c,var,value]
    iff not CentralMemoryManager.replyAck [t, t',c,var,value]
    iff not CentralMemoryManager.keepMemState[t,t']
}
pred CentralMemoryManager.init[t:Tick] {
  #this.mem.t = 0
}
pred CentralMemoryManager.receiveWrite[t,t':Tick,  c:Component, var:Var, value:Value ] {
  this.executeCall[c, Write, var + value, ACK, t]
  this.keepMemState[t,t']
  this.updateMem[t',t'.next,c,var,value]
}
pred CentralMemoryManager.updateMem[t,t':Tick,  c:Component, var:Var, val:Value ] {
  one r:Res | r.variable = var and r.value = val and this.mem.t' =  this.mem.t + r
  this.replyAck[t',t'.next,c,var,val]
}
pred CentralMemoryManager.replyAck[t,t':Tick, c:Component, var:Var, val:Value ] {
  this.reply[c, Write, var + val, ACK, t]
  this.keepMemState[t,t']
}
pred CentralMemoryManager.receiveRead[t,t':Tick, c:Component, var:Var, val:Value] {
  this.executeCall[c, Read, var, val, t]
  one r:Res | r in this.mem.t and r.variable = var and r.value = val
  this.replyVar[t',t'.next,c,var,val]
  this.keepMemState[t,t']
}
pred CentralMemoryManager.replyVar[t,t':Tick,  c:Component, var:Var, val:Value ] {
  this.reply[c, Write, var, val, t]
  this.keepMemState[t,t']
}
pred CentralMemoryManager.keepMemState[t,t':Tick] {
  this.mem.t = this.mem.t'
}
pred Component.H_write[var:Var, val:Value, t:Tick] {
  this.call[CentralMemoryManager, Write, var + val, ACK, t]
```

```
}
pred Component.H_executeWrite[var:Var, val:Value, t:Tick] {
  this.executeReply[CentralMemoryManager, Write, var + val, ACK, t]
}
pred Component.H_read[var:Var, t:Tick] {
  this.call[CentralMemoryManager, Read, var, ACK, t]
}
pred Component.H_executeRead[var:Var, val:Value, t:Tick] {
  this.executeReply[CentralMemoryManager, Read, var, val, t]
}
```

Listing A.4: Distributed shared memory

Among the set of possible and specified characteristics of the behaviors of the distributed shared memory communication style, a subset of them are encoded in terms of properties as predicates and assertions and the results of their verification are stated below.

(a) “Once a caller *c* reads from shared variable *var*, eventually it gets a value *value* for *var*.”

```
pred starvation_freeness_read[c:Component, var:Var, value:Value] {
  some t:Tick, t’:t.nexts |
    c.H_read[var, t] => c.H_executeRead[var, value, t’]
}
```

(b) “Once a caller *c* writes a value *value* in shared variable *var*, eventually it gets an acknowledgement that the operation is successful.”

```
pred starvation_freeness_write [c:Component, var:Var, value:Value] {
  all t:Tick, t’:t.nexts |
    c.H_write[var, value, t]  => c.H_executeWrite[var, value, t’]
}
```

(c) “Once a caller $c1$ writes a value *value* in a shared variable *var* and no others client $c3$ writes any new value $value'$ in *var*, eventually a client $c2$ getting *value* by reading *var*.”

```
pred eventual_consistency[c1,c2:Component, var:Var, value:Value {
  some t:Tick, t’:t.nexts |
    all c3:Component, t’’:t.nexts, value’:Value |
      c1.executeWrite[var, value, t]
      and not c3.executeWrite[var,value’,t’’]
      => c2.executeRead[var, value, t’]
}
```

The Alloy Analyzer shows that properties (a), (b) and (c) hold for a DSM communication style.

A.2.3 Building the concrete architecture for the illustrative example

Model the software architecture The software architecture model is defined as an instance of the proposed metamodel with respect to the functional requirements as identified in Section 4.5. Listing A.5 depicts the Alloy specification of the architecture of the college library web application example described in Figure 2.8. For instance, the *Browser* can be seen as an instantiation of the *Component* type. We proceed by defining the component types (lines 1 to 4), ports (lines 6 to 11), and interfaces (lines 13 to 18) as simple extensions to the concepts of our software architecture metamodel.

```
 1  one sig Browser extends Component {}{ uses = PortInterfaceBrowser }
 2  one sig Website extends Component {}{ uses = PortInterfaceWebsite +
       PortDataWebsite }
 3  one sig Database extends CentralMemoryManager {}{ uses =
       PortDataDatabase + PortSMDatabase }
 4  one sig Terminal extends Component{}{ uses = PortSMTerminal }
 5
 6  one sig PortInterfaceBrowser extends Port {}{ realizes =
       InterfaceBrowser }
 7  one sig PortInterfaceWebsite extends Port {}{ realizes =
       InterfaceWebsite }
 8  one sig PortDataWebsite extends Port {}{ realizes = DataWebsite }
 9  one sig PortDataDatabase extends Port {}{ realizes = DataDatabase }
10  one sig PortSMDatabase extends Port {}{ realizes =
       InterfaceSMDataDatabase }
11  one sig PortSMTerminal extends Port {}{ realizes =
       InterfaceSMAdminstrator }
12
13  one sig InterfaceBrowser extends Interface {}{k ind = REQUIRED and
       getBook in methods }
14  one sig InterfaceWebsite extends Interface {}{kind = PROVIDED and
       getBook in methods }
15  one sig getBook extends Method {}{ idBook in parameters }
16  one sig idBook extends Parameter {}{ direction = IN }
17  one sig InterfaceSMAdminstrator extends Interface {}{ kind = REQUIRED
       and Write in methods }
18  one sig InterfaceSMDataDatabase extends Interface {}{ kind = PROVIDED
       and Write + Read in methods }
```

```

one sig DataWebsite extends Data {}{ kind = DATA_IN }
one sig DataDatabase extends Data {}{ kind = DATA_OUT }
one sig DatabaseContent extends DataType {}
```

Listing A.5: Building a concrete software architecture of the college library web application example in Alloy

Incorporate connectors At this step, connectors are integrated by reusing the developed and verified connector libraries. In the same way as the metamodel, connectors are instantiated in the concrete architecture by simply defining connector types as extensions of one of the defined connectors. Listing A.6 depicts the Alloy specification of connectors for the college library web application example described in Section 2.8. The corresponding connector communication style is chosen as identified in Section 4.6.3.

```
one sig BrowerWebsiteConnector extends ConnectorRPC {}{
  portO = PortInterfaceBrowser
  portI = PortInterfaceWebsite
}
one sig DatabaseWebsiteConnector extends ConnectorMPS {}{
  portO = PortDataDatabase
  portI = PortDataWebsite
}
one sig TerminalDatabaseConnectorSM extends ConnectorSM {}{
  portO = PortSMTerminal
  portI = PortSMDatabase
}
```

Listing A.6: Instantiate Connectors of a web application example in Alloy

Verify functional requirements

- *Req_1.* we reuse some of the previously specified and verified structural and communication behavior properties, this time for remote procedure calls, to specify and verify the application specific functional requirement *Req_2* on the concrete architecture. Listing A.7 shows the Alloy specification of this functional requirement as a predicate combining the verified properties *invocation_type_check* (line 3) and *send_is_eventually_replied* (line 4) . These properties are applied to the specific concepts and behaviors identified for this requirement as described in Section 4.5 and 4.6.3. The Alloy Analyzer shows that *Req_2* holds.

```
1  pred Req_2{
2    some w:Website, b:Browser, m:getBook, args_in,args_out: set
         Argument {
3      invocation_type_check[w,b,m]
4      send_is_eventually_replied[w,b,m,args_in,args_out]
5    }
6  }
```

Listing A.7: Using previously verified RPC properties to specify a functional requirement of a web application example in Alloy

- *Req_2.* Similar to *Req_1*, we use some of the previously specified and verified structural and communication behavior properties for message passing to specify and verify the application specific functional requirement *Req_2* on the concrete architecture. Listing A.8 depicts the Alloy specification of this functional requirement as a predicate combining the verified properties *msgpassing_type_check* (line 3) and *send_is_eventually_received* (line 4). These properties are applied to the specific concepts and behaviors identified for this requirement as discussed in Section 4.5 and 4.6.3. The Alloy Analyzer shows that *Req_1* holds.

```
1  pred Req_1 {
2    some w:Website, d:Database, m:Message, typ:DataType {
3      msgpassing_type_check[d,w,typ]
4      send_is_eventually_received[d,w,m,typ]
5    }
6  }
```

Listing A.8: Using previously verified MPS properties to specify functional requirement of a web application example in Alloy

- *Req_3.* Once again, we reuse some of the previously specified and verified structural and communication behavior properties, now for distributed shared memory, to specify and verify the application specific functional requirement *Req_3* on the concrete architecture. Listing A.7 gives the Alloy specification of this functional requirement as a predicate combining the verified properties *invocation_type_check* (line 3) and *starvation_freeness_write* (line 4). As with requirements *Req_1* and *Req_2*, these properties are applied to the specific concepts and behaviors identified for this requirement discussed in Section 4.5 and 4.6.3. The Alloy Analyzer shows that *Req_3* holds.

```
pred Req_3 {
  some a:Administrator, d:Database, var:Var, val:Value {
    invocation_type_check[a,d,Write]
    starvation_freeness_write[a,var,val]
  }
}
```

Listing A.9: Using previously verified DSM properties to specify a functional requirement of a web application example in Alloy

Appendix B

Coq confidentiality

B.1 Introduction to Coq tatics

In this section, we introduce the tactics used in the proof presented Section B.2.

Intros / intro Introduces variables appearing with **forall** *as well as the premises (left-hand side) of implications.*
If the goal contains universally quantifiable variables (i.e., **forall : intros**), we can use the **intros** tactic to incorporate those variables into the context. All hypotheses on the left side of an implication can alternatively be introduced as assumptions using intros. If **intros** is used alone, Coq will introduce all of the variables and hypotheses it can and will name them automatically. By supplying the names in order, we can offer your own names. There is a sister tactic **intro** that only introduces one thing.

For example, if we try to prove the following *modus_tollens* theorem:

$$(\mathrm{P} \land \mathrm{Q}) \rightarrow \mathrm{P}$$

We can start by introducing the variables, as well as the hypotheses, using **intro**. Then, we obtain the following hypothesis:

- *P_implies_Q* : $\mathrm{P} \rightarrow \mathrm{Q}$
- *not_Q*: $\neg\,\mathrm{Q}$

Listing B.1 show the corresponding Coq for this example.

```
Theorem modus_tollens : ∀ (P Q : Prop),
  (P → Q) → ¬Q → ¬P.
```

```
Proof.
  intros P Q P_implies_Q not_Q.
```

Listing B.1: Coq tatic intro

Unfold Unfolds the definitions of terms.
The unfold tactic replaces a defined term with its definition. You may also use unfold on a hypothesis with the syntax **unfold** <term> **in** <hypothesis>.

For example, if we try to prove the following *simple_decrement* theorem:

$$\text{decrement } 220 = 219$$

With *decrement*(x) defined as :

$$\text{x} - 1$$

Then, using the $unfold$ tactic we obtain by replacing the defined term *decrement* by is definition:

$$220 - 1 = 219$$

Listing B.2 show the corresponding Coq for this example.

```
Definition decrement (x : nat) : nat :=
  x − 1.

Theorem simple_decrement :
  decrement 220 = 219.
Proof.
  unfold decrement.
```

Listing B.2: Coq tatic unfold

Apply Uses implications to transform goals and hypotheses.
If you have some hypothesis that states that A holds, as well as another hypothesis $\text{A} \to \text{B}$, you can use **apply** to transform the first hypothesis into B. The syntax is **apply** <term> **in** <hypothesis> or **apply** <term> **in** <hypothesis> **as** <new−hypothesis>.

For example, if try to prove the following *modus_ponens* theorem:

$$(\text{P} \to \text{Q}) \to \text{P} \to \text{Q}.$$

We can start by introducing the variables, as well as the hypotheses, using **intro**. Then, the following hypothesis:

- *P_implies_Q* : $\text{P} \to \text{Q}$

- *P_holds*: P

Therefore, we notice that P holds, and because we know that P → Q, we can use the **apply** tactic to apply the implication *P_implies_Q* in our hypothesis *P_holds* to transform it. This way we obtain the new hypothesis *Q_holds* :

$$Q$$

Listing B.3 show the corresponding Coq for this example.

```
Theorem modus_ponens : ∀ (P Q : Prop),
  (P → Q) → P → Q.
Proof.
  intros P Q P_implies_Q P_holds.
  apply P_implies_Q in P_holds as Q_holds.
```

Listing B.3: Coq tatic apply

Destruct (And) Replaces a hypothesis P / Q with two hypotheses P and Q
A hypothesis P ∧ Q means that both A and B hold, hence it can be deconstructed into two new hypotheses P and Q. You can also use the **destruct** ... **as** [... | ...] syntax to give these new hypotheses a unique name.

For example, given a starting hypothesis *P_and_Q*:

$$P \land Q$$

Using the **destruct** tactic allows us to obtain two new hypotheses:

- *P_holds* : P
- *Q_holds* : Q

Listing B.4 show the corresponding Coq for this example.

```
Theorem and_left : ∀ (P Q : Prop),
  (P ∧ Q) → P.
Proof.
  intros P Q P_and_Q.
  destruct P_and_Q as [P_holds Q_holds].
```

Listing B.4: Coq tatic destruct (and)

Specialize Instantiated by concrete terms the premises of this hypothesis
This tactic works on local hypothesis. The premises of this hypothesis (either universal quantifications or non-dependent implications) are instantiated by concrete terms.

For example, if we have the following hypothesis H :

$$\mathrm{A} \to \mathrm{B}$$

And hypothesis a :

$$\mathrm{A}$$

We can use the **specialize** tactic to specialize H for a. We obtain this way the new hypothesis:

$$\mathrm{B}$$

Listing B.5 show the corresponding Coq for this example.

```
Theorem specialize {A B: Type} (H: A → B) (a: A): B.
Proof.
  specialize (H a).
```

Listing B.5: Coq specialize simpl

Generalize Generalizes the conclusion with respect to some term.
For example, if we have the following hypothesis :

0 <= x + y + y

Using the **generalize** tactic we can generalize it as :

∀ n : nat, 0 <= n.

Listing B.6 show the corresponding Coq for this example.

```
Theorem T x y:
  0 <= x + y + y.
Proof.
  generalize (x + y + y).
```

Listing B.6: Coq generalize simpl

Subst Transform an identifier into an equivalent term
The **subst** tactic substitute an identifier by something else which is equal.

For example, if we know that :

a = b

And we want to show :

$$b = a$$

We can use **subst** to transform the *a* in the goal into *b*, so our goal becomes b = b.

Listing B.7 show the corresponding Coq for this example.

```
Inductive bool: Set :=
  | true
  | false.

Lemma equality_commutes:
  ∀ (a: bool) (b: bool), a = b → b = a.
Proof.
  intros.
  subst.
```

Listing B.7: Coq tatic subst

Rewrite Replaces a term with an equivalent term if the equivalence of the terms has already been proven.

Given some known equality a = b, the rewrite tactic lets you replace a with b or vice versa in a hypothesis. The syntax is **rewrite** <equality> **in** <hypothesis> to replace a with b or **rewrite** ← <equality> **in** <hypothesis> to replace b with a.

For example, if we have the hypothesis $H0$:

$$f\ y = f\ z$$

And our goal is :

$$f\ x = f\ z$$

We can change our goal from fy into fx using **rewrite** backwards. This way we get :

$$f\ x = f\ y$$

Listing B.8 show the corresponding Coq for this example.

```
Inductive bool: Set :=
  | true
  | false.

Lemma equality_of_functions_transits:
  ∀ (f: bool→ bool) x y z, (f x) = (f y) → (f y) = (f z) → (f x) = (f z).
Proof.
```

```
  intros.
  rewrite ← H0.
```

Listing B.8: Coq tatic rewrite

Simpl Simplifies the goal or hypotheses in the context.
The **simpl** tactic reduces complex terms to simpler forms. Using the syntax **simpl in** < hypothesis>, **simpl** can also be used on a specific hypothesis in the context.

For example, if we have the following hypothesis :

$$220 - 1 = 219$$

Using the **simpl** tactic we obtain the simplified hypothesis :

$$219 = 219$$

Listing B.9 show the corresponding Coq for this example.

```
Theorem simple_decrement :
  220 − 1 = 219.
Proof.
  simpl.
```

Listing B.9: Coq tatic simpl

Clear Erases a hypothesis from the local context.
clear <hypothesis> erases the hypothesis in the local context of the current goal. As a consequence, the hypothesis is no more displayed and no more usable in the proof development.

For example, during a proof if we don't need a hypothesis named *P_holds*, then we can delete it using the **clear** tactic. Listing B.10 show the corresponding Coq for this example.

```
Theorem [...] : [...],
  [...]
Proof.
  [...]
  clear P_holds.
```

Listing B.10: Coq tatic clear

Auto Solves a variety of easy goals.
auto performs a recursive proof search. It never fails even if it cannot do anything.

For example, if we try to prove the following *modus_tollens* theorem:

$$(P \to Q) \to \neg Q \to \neg P.$$

Using, the **auto** tactic show that the theorem is valid.

Listing B.11 show the corresponding Coq for this example.

```
Theorem modus_tollens: ∀ (P Q : Prop),
  (P → Q) → ¬Q → ¬P.
Proof.
  auto.
Qed.
```

Listing B.11: Coq tatic auto

B.2 Proof

```
Definition PayloadConfidentiality B c1 c2 :=
  ∀ c3 m d, not (In c3 [c1; c2]) → (B ⊢Eventually Any (E c3 (get_pld m d))) →
    not (B ⊢(H c1 (inject m) && has_rcv m c2) << (E c3 (get_pld m d))).
```

Listing B.12: Coq PayloadConfidentiality property definition

Security policy. *restrictiveGetPld* property constrains the *inject* and *get_pld* operations using *AllowedGetPld*. It is as an abstract security mechanism to satisfy the confidentiality property by identifying components who are able to get the payload.

```
Record AllowedGetPld : Type := {
  msg: Message ;
  comp : Component;
}.

Definition restrictiveGetPld (p:AllowedGetPld) m B :=
  (∀ c, (B ⊢Eventually Any (E c (inject m)) → (msg p = m ∧ Rcv m = comp p)))
    ∧
  (∀ c d, (B ⊢Eventually Any (E c (get_pld m d))) → (msg p = m ∧ c = comp p)).

Definition connectorRestritiveGetPld p B :=
  ∀ c m, (B ⊢Eventually Any (E c (intercept m))) → restrictiveGetPld p m B.
```

Listing B.13: Coq restrictiveGetPld policy definition

We can now proceed with the proof.

Proof goal. We want to prove that if the connector *con* between $c1$ and $c2$ enforces the *allowedGetPld* policy (hypothesis *connectorRestritiveGetPld*), then the confidentiality objective ($PayloadConfidentiality$) between $c1$ and $c2$ holds, i.e., no $c3$ component is able to get the payload of a message m send by $c1$ to $c2$ through connector *con*.

Proof overview. In order to show this, we use proof by contradiction. We assume that *connectorRestritiveGetPld* (the policy is deployed) and B ⊢ (H c1 (inject m) && has_rcv m c2)<< E c3 (get_pld m d) (the negation of the final implication of our proof goal $PayloadConfidentiality$) and we derive a contradiction. The proof consists of the following steps:

Step 1 We define the initial hypotheses for all B behavior; $c1$, $c2$, $c3$ components; p *allowedGetPld* policy; m a message and d a payload.

Unfolding the definition and specializing some of them, we obtain the following starting hypothesizes:

1. *connector_restritive_get_pld* (specialize for c3 and m):

```
B ⊢ Eventually Any (E c3 (intercept m)) →
   ( ∀ c : Agent, B ⊢ Eventually Any (E c (inject m))
         → msg p = m ∧ Rcv m = comp p)
   ∧
   ( ∀ (c : Agent) (d : Data), B ⊢ Eventually Any (E c (get_pld m d))
         → msg p = m ∧ c = comp p)
```

2. *c3_not_in_c1_c2* (from $MessageConfidenciality$) :

$$\neg \text{In c3 [c1; c2]}$$

3. *c3_eventually_enable_to_get_d* (from $MessageConfidenciality$) :

$$\text{B} \vdash \text{Eventually Any (E c3 (get\_pld m d))}$$

4. *c1_injected_m_to_c2* (negation of the final implication of $MessageConfidenciality$

(proof by contradiction)) :

$$B \vdash (H\ c1\ (inject\ \ m)\ \&\&\ has\_rcv\ m\ c2) << E\ c3\ (get\_pld\ m\ d)$$

Listing B.14 show the corresponding part of the proof using Coq.

```
Theorem confidentialityHold:
  ∀ B c1 c2 p, connectorRestritiveGetPld p B → PayloadConfidentiality B c1 c2.
Proof.
  intros B c1 c2 p connector_restritive_get_pld .
  unfold PayloadConfidentiality, connectorRestritiveGetPld, restrictiveGetPld  in ∗.
  intros c3 m d c3_not_in_c1_c2 c3_eventually_enable_to_get_d  c1_injected_m_to_c2.
  specialize (connector_restritive_get_pld  c3 m).
```

Listing B.14: Coq confidentiality proof step 1

Step 2 In this step, we show that, as we know, the behavior B entails that eventually $c3$ is able to get the payload d of message m, then it means that a behavior B entails that eventually $c3$ intercepted this message before.

Starting from the hypothesis *c3_eventually_enable_to_get_d* :

$$B \vdash Eventually\ Any\ (E\ c3\ (get\_pld\ m\ d))$$

1. We apply *axiomInterceptHasPldImpEGetPld* :

   $$\forall\ B\ c\ m\ d,\ B \vdash (H\ c\ (intercept\ \ m)\ \&\&\ (has\_pld\ m\ d)) << E\ c\ (get\_pld\ m\ d)$$

   For the component $c3$, message m and payload d.
   This way we obtain *axiomInterceptHasPldImpEGetPld_c3_m_d* :

   $$B \vdash (H\ c3\ (intercept\ \ m)\ \&\&\ (has\_pld\ m\ d)) << E\ c3\ (get\_pld\ m\ d)$$

2. Then, we apply the lemma *prec_eventually* :

   $$\begin{aligned} &\forall\ B\ p\ q\ A,\ (B \vdash p << q)\\ &\quad \rightarrow (B \vdash Eventually\ A\ q)\\ &\qquad \rightarrow (B \vdash Eventually\ A\ p) \end{aligned}$$

For *axiomInterceptHasPldImpEGetPld_c3_m_d* :

$$
\begin{gathered}
(\text{B} \vdash (\text{H c3 (intercept m)} \&\& \text{ has\_pld m d}) << \text{E c3 (get\_pld m d)}) \\
\rightarrow (\text{B} \vdash \text{Eventually Any H c3 (intercept m)} \&\& \text{ has\_pld m d})) \\
\rightarrow (\text{B} \vdash \text{Eventually Any E c3 (get\_pld m d)})
\end{gathered}
$$

Therefore, we get the new hypothesis *c3_eventually_intercept_m_and_hasPld_m_d* :

$$\text{B} \vdash \text{Eventually Any (H c3 (intercept m)} \&\& \text{ has\_pld m d)}$$

3. By applying the lemma *satEventuallyAnd_elim* :

$$\text{B} \vdash \ \forall \text{ B p q, } (\text{B} \vdash \text{p } \&\& \text{ q}) \rightarrow ((\text{B} \vdash \text{p}) \wedge (\text{B} \vdash \text{q}))$$

for *c3_eventually_intercept_m_and_hasPld_m_d*.
We come by *c3_eventually_intercept_m*:

$$\text{B} \vdash \text{Eventually Any (H c3 (intercept m))} \wedge \text{B} \vdash \text{Eventually Any (has\_pld m d)}$$

4. Finally, by breaking apart the conjunction we get *c3_eventually_intercept_m* :

$$\text{B} \vdash \text{Eventually Any (H c3 (intercept m))}$$

Therefore, we show that behavior B satisfies that eventually $c3$ intercepted this message. Listing B.15 show the corresponding part of the proof using Coq.

```
generalize (axiomInterceptHasPldImpEGetPld B c3 m d). intro
    axiomInterceptHasPldImpEGetPld_c3_m_d.
generalize (prec_eventually _ _ _ _ axiomInterceptHasPldImpEGetPld_c3_m_d
    c3_eventually_enable_to_get_d). clear axiomInterceptHasPldImpEGetPld_c3_m_d;
intro c3_eventually_intercept_m_and_hasPld_m_d.
apply satEventuallyAnd_elim in c3_eventually_intercept_m_and_hasPld_m_d.
destruct c3_eventually_intercept_m_and_hasPld_m_d as [c3_eventually_intercept_m _].
```

Listing B.15: Coq confidentiality proof step 2

Step 3. We show that as the behavior B satisfies eventually $c3$ intercepted the message m, then it means that the behavior B satisfies eventually $c3$ was also able to intercept this message.

Starting from the newly obtain hypothesis *c3_eventually_intercept_m* :

$$B \vdash \text{Eventually Any (H c3 (intercept m))}$$

1. First, we generalize Axiom *axiomHImpE* :

   $$\forall \text{ B c1 p, B} \vdash \text{(H c1 p)} \Rightarrow \text{(E c1 p)}$$

   For component *c*3 and the action *intercept* on the message *m*.
   Consequently, we obtain *axiomHImpE_c3_intercept_m* :

   $$\begin{aligned}&\forall \text{ B : nat} \rightarrow \text{Component} * \text{Action,}\\ &\text{B} \vdash \text{Eventually Any (H c3 (intercept m))}\\ &\quad \Rightarrow \text{Eventually Any (E c3 (intercept m))}\end{aligned}$$

2. Next, we apply the lemma *eventually_imply* :

   $$\begin{aligned}&\forall \text{ p q A, (} \forall \text{ B, B} \vdash \text{p} \Rightarrow \text{q)}\\ &\quad \rightarrow \forall \text{ B, B} \vdash \text{Eventually A p} \Rightarrow \text{Eventually A q}\end{aligned}$$

   For *axiom1_c3_intercept_m*, we obtain *ev_c3_intercept_m_impl_enable_c3_intercept_m*:

   $$\begin{aligned}&\text{B} \vdash \text{Eventually Any (H c3 (intercept m))}\\ &\quad \Rightarrow \text{Eventually Any (E c3 (intercept m))}\end{aligned}$$

3. Then, we use the lemma *satImpl_elim* :

   $$\forall \text{ B p q, (B} \vdash \text{p} \Rightarrow \text{q)} \rightarrow \text{((B} \vdash \text{p)} \rightarrow \text{(B} \vdash \text{q))}$$

   Generalized for *ev_c3_intercept_m_impl_enable_c3_intercept_m* and *axiomHImpE_c3_intercept_m*.

Thus, we obtain *ev_c3_intercept_m_impl_enable_c3_intercept_m* :

$$\begin{aligned}&(\text{B} \vdash \text{Eventually Any (H c3 (intercept m))}\\ &\quad \Rightarrow \text{Eventually Any (E c3 (intercept m)))}\\ &\qquad \rightarrow (\text{B} \vdash \text{Eventually Any (H c3 (intercept m)))}\\ &\qquad\quad \rightarrow (\text{B} \vdash \text{Eventually Any (E c3 (intercept m)))}\end{aligned}$$

4. Therefore, we get the new hypothesis *c3_eventually_e_intercept_m* :

$$\text{B} \vdash \text{Eventually Any (E c3 (intercept m))}$$

Hence, we prove that the behavior B satisfies that eventually $c3$ was also able to intercept this message. Listing B.16 show the corresponding part of the proof using Coq.

```
generalize(fun B → axiomHImpE B c3 (intercept m)); intro axiomHImpE_c3_intercept_m.
generalize (eventually_imply _ _ Any axiomHImpE_c3_intercept_m B); clear axiomHImpE_c3_intercept_m;
intro ev_c3_intercept_m_impl_enable_c3_intercept_m.
generalize (satImpl_elim B _ _ ev_c3_intercept_m_impl_enable_c3_intercept_m c3_eventually_intercept_m);
clear ev_c3_intercept_m_impl_enable_c3_intercept_m ev_c3_intercept_m_impl_enable_c3_intercept_m;
intro c3_eventually_e_intercept_m.
```

Listing B.16: Coq confidentiality proof step 3

Step 4. We apply the hypothesis from the *Step 1 connector_restritive_get_pld* in the new hypothesis *c3_eventually_e_intercept_m* obtained in the previous step in order to simplify it. From this we then deduce *restritive_get_pld*.

1. We apply *connector_restritive_get_pld* :

$$
\begin{aligned}
&\mathrm{B} \vdash \text{Eventually Any (E c3 (intercept m))} \rightarrow \\
&\quad (\ \forall\ \text{c : Agent,} \\
&\qquad \mathrm{B} \vdash \text{Eventually Any (E c (inject m))} \\
&\qquad\quad \rightarrow \text{msg p} = \text{m} \wedge \text{Rcv m} = \text{comp p}) \\
&\quad \wedge \\
&\quad (\ \forall\ \text{(c : Agent) (d : Data),} \\
&\qquad \mathrm{B} \vdash \text{Eventually Any (E c (get\_pld m d))} \\
&\qquad\quad \rightarrow \text{msg p} = \text{m} \wedge \text{c} = \text{comp p})
\end{aligned}
$$

In *c3_eventually_e_intercept_m* :

$$\mathrm{B} \vdash \text{Eventually Any (E c3 (intercept m)}$$

Therefore, we get *restritive_get_pld* :

$$
\begin{aligned}
&(\ \forall\ \text{c : Agent,} \\
&\quad \mathrm{B} \vdash \text{Eventually Any (E c (inject m))} \\
&\qquad \rightarrow \text{msg p} = \text{m} \wedge \text{Rcv m} = \text{comp p}) \\
&\wedge \\
&(\ \forall\ \text{(c : Agent) (d : Data),} \\
&\quad \mathrm{B} \vdash \text{Eventually Any (E c (get\_pld m d))} \\
&\qquad \rightarrow \text{msg p} = \text{m} \wedge \text{c} = \text{comp p})
\end{aligned}
$$

Listing B.17 show the corresponding part of the proof using Coq.

```
apply connector_restritive_get_pld  in c3_eventually_e_intercept_m  as restritive_get_pld  ;  clear
    connector_restritive_get_pld   c3_eventually_e_intercept_m .
```

Listing B.17: Coq confidentiality proof step 4

Step 5. In this step, we simplify again the hypothesis *restritive_get_pld* and show that $c3$ and m must be the same, respectively, as the component, denoted by *comp p*, and the message, denoted by *msg p*, of the policy p. The obtained terms are then substituted in the rest of the hypotheses.

Starting from *restritive_get_pld* :

$$
\begin{aligned}
&(\ \forall\, \mathrm{c} :\ \mathrm{Agent},\\
&\mathrm{B} \vdash \mathrm{Eventually\ Any\ (E\ c\ (inject\ \ m))} \rightarrow \mathrm{msg\ p} = \mathrm{m} \wedge \mathrm{Rcv\ m} = \mathrm{comp\ p})\\
&\wedge\\
&(\forall\, (\mathrm{c} :\ \mathrm{Agent})\, (\mathrm{d} :\ \mathrm{Data}),\\
&\mathrm{B} \vdash \mathrm{Eventually\ Any\ (E\ c\ (get\_pld\ m\ d))} \rightarrow \mathrm{msg\ p} = \mathrm{m} \wedge \mathrm{c} = \mathrm{comp\ p})
\end{aligned}
$$

1. First, we separate *c3_eventually_e_intercept_m* into two sub hypotheses:

   - *eventually_e_inj_m_impl_valid_policy*

     $$
     \begin{aligned}
     &(\ \forall\, \mathrm{c} :\ \mathrm{Agent},\\
     &\mathrm{B} \vdash \mathrm{Eventually\ Any\ (E\ c\ (inject\ \ m))} \rightarrow \mathrm{msg\ p} = \mathrm{m} \wedge \mathrm{Rcv\ m} = \mathrm{comp\ p})
     \end{aligned}
     $$

   - *eventually_e_get_m_d_impl_valid_policy*

     $$
     \begin{aligned}
     &(\forall\, (\mathrm{c} :\ \mathrm{Agent})\, (\mathrm{d} :\ \mathrm{Data}),\\
     &\mathrm{B} \vdash \mathrm{Eventually\ Any\ (E\ c\ (get\_pld\ m\ d))} \rightarrow \mathrm{msg\ p} = \mathrm{m} \wedge \mathrm{c} = \mathrm{comp\ p})
     \end{aligned}
     $$

2. After, we apply the *eventually_e_inj_m_impl_valid_policy* for $c1$:

   $$\mathrm{B} \vdash \mathrm{Eventually\ Any\ (E\ c1\ (inject\ \ m))} \rightarrow \mathrm{msg\ p} = \mathrm{m} \wedge \mathrm{Rcv\ m} = \mathrm{comp\ p}$$

3. Then, we apply the *eventually_e_inj_m_impl_valid_policy* for $c3$ and d :

   $$\mathrm{B} \vdash \mathrm{Eventually\ Any\ (E\ c3\ (get\_pld\ m\ d))} \rightarrow \mathrm{msg\ p} = \mathrm{m} \wedge \mathrm{c3} = \mathrm{comp\ p}$$

4. As we know, *c3_eventually_enable_to_get_d* from *Step 1* :

   $$\mathrm{B} \vdash \mathrm{Eventually\ Any\ (E\ c3\ (get\_pld\ m\ d))}$$

   Is true, we deduce from *eventually_e_inj_m_impl_valid_policy*:

   $$\mathrm{msg\ p} = \mathrm{m} \wedge \mathrm{c3} = \mathrm{comp\ p}$$

By breaking apart the conjunction the following two terms :

- msg p = m
- c3 = comp p

5. Finally, we substitute these terms in our hypothesis. This way we obtain the following updated hypothesis:

- *c3_not_in_c1_c2* (from *Step 1*) :

$$\neg \, \text{In (comp p)[c1; c2]}$$

- *c3_eventually_intercept_m* (from *Step 2*) :

$$\text{B} \vdash \text{Eventually Any (H (comp p)(intercept (msg p)))}$$

- *c1_injected_m_to_c2* (from *Step 1*):

$$\text{B} \vdash \text{(H c1 (inject (msg p)) \&\& has\_rcv (msg p)c2)} << \\ \text{E (comp p)(get\_pld (msg p) d)}$$

- *eventually_e_inj_m_impl_valid_policy* (from current Step) :

$$\text{B} \vdash \text{Eventually Any (E c1 (inject (msg p)))} \rightarrow \\ \text{msg p} = \text{msg p} \wedge \text{Rcv (msg p)} = \text{comp p}$$

Listing B.18 show the corresponding part of the proof using Coq.

```
destruct c3_eventually_e_intercept_m as [eventually_e_inj_m_impl_valid_policy
    eventually_e_get_m_d_impl_valid_policy ].
specialize (eventually_e_inj_m_impl_valid_policy  c1).
specialize (eventually_e_get_m_d_impl_valid_policy  c3 d).
destruct (eventually_e_get_m_d_impl_valid_policy c3_eventually_enable_to_get_d ) ; clear
    eventually_e_get_m_d_impl_valid_policy .
subst.
```

Listing B.18: Coq confidentiality proof step 5

Step 6. We show that, since the behavior B satisfies that eventually the component denoted by *comp p* of the policy p is able to get the payload d from the message denoted by *msg p* of the policy p, the behavior B satisfies that eventually the component *comp p* is the one that intercepts the message *msg p*.

Starting from hypothesis *c3_eventually_enable_to_get_d*:

$$\text{B} \vdash \text{Eventually Any (E (comp p)(get\_pld (msg p)d))}$$

1. We begin by using Axiom *axiomInterceptHasPldImpEGetPld*:

   $$\forall \text{ B c m d, B} \vdash \text{(H c (intercept m) \&\& (has\_pld m d))<< E c (get\_pld m d)}$$

   Generalized for the component of the policy *comp p*, the message protected by the policy *msg p* and the payload *d*.
   Therefore, we obtain *axiomInterceptHasPldImpEGetPld_comp_p_msg_p_d*:

   $$\begin{array}{l}\text{B} \vdash \text{(H (comp p)(intercept (msg p)) \&\& has\_pld (msg p)d)} \\ \quad \text{<< E (comp p)(get\_pld (msg p)d}\end{array}$$

2. Then, using lemma *prec_eventually*

   $$\forall \text{ B p q A, (B} \vdash \text{p << q)} \rightarrow \text{(B} \vdash \text{Eventually A q)} \rightarrow \text{(B} \vdash \text{Eventually A p)}$$

   Generalized for *axiomInterceptHasPldImpEGetPld_comp_p_msg_p_d* and *c3_eventually_enable_to_get_d*.

   We get *eventually_h_comp_p_intercept_msg_p*:

   $$\text{B} \vdash \text{Eventually Any (H (comp p)(intercept (msg p)) \&\& has\_pld (msg p)d)}$$

3. Finally, we apply Lemma *satEventuallyAnd_elim* :

   $$\text{B} \vdash \forall \text{ B p q, (B} \vdash \text{p \&\& q)} \rightarrow \text{((B} \vdash \text{p)} \wedge \text{(B} \vdash \text{q))}$$

for *eventually_h_comp_p_intercept_msg_p* to get :

$$\begin{gathered}\text{B} \vdash \text{Eventually Any (H (comp p)(intercept (msg p)))} \land \\ \text{B} \vdash \text{Eventually Any (has\_pld (msg p)d)}\end{gathered}$$

And, break apart the conjunction and keep only the left part as *eventually_h_comp_p_intercept_msg_p* :

$$\text{B} \vdash \text{Eventually Any (H (comp p)(intercept (msg p)))}$$

Thus, we prove that the behavior B satisfies eventually that the component of the policy *comp p* is the one that intercepts the message protected by the policy *msg p*. Listing B.19 show the corresponding part of the proof using Coq.

```
generalize(axiomInterceptHasPldImpEGetPld B (comp p) (msg p) d); intro
    axiomInterceptHasPldImpEGetPld_comp_p_msg_p_d.
generalize (prec_eventually _ _ _ _ axiomInterceptHasPldImpEGetPld_comp_p_msg_p_d
    c3_eventually_enable_to_get_d).
clear axiomInterceptHasPldImpEGetPld_comp_p_msg_p_d.
intro eventually_h_comp_p_intercept_msg_p.
apply satEventuallyAnd_elim in eventually_h_comp_p_intercept_msg_p.
destruct eventually_h_comp_p_intercept_msg_p as [eventually_h_comp_p_intercept_msg_p _].
```

Listing B.19: Coq confidentiality poof step 6

Step 7. In this step, we show that since the behavior B satisfies that eventually the component *comp p* is the one that intercepts the message *msg p*, then the behavior B satisfies that eventually the component $c1$ injected the message *msg p*.

Starting from hypothesis *eventually_h_comp_p_intercept_msg_p*:

$$\text{B} \vdash \text{Eventually Any (H (comp p)(intercept (msg p)))}$$

1. First, we use axiom *axiomInjectPrecIntercept* :

$$\forall \text{ B c1 c2 m, B} \vdash \text{H c2 (inject m)} << \text{E c1 (intercept m)}$$

Generalized for the component of the policy *comp p*, component $c1$ and the message protected by the policy *msg p*.

We get *axiomInjectPrecIntercept_comp_p_c1_msg_p_d*:

$$\text{B} \vdash \text{H c1 (inject (msg p))} << \text{E (comp p)(intercept (msg p))}$$

2. Next, we use *axiomHImpE*:

$$\forall \text{ B c1 p, B} \vdash \text{(H c1 p)} \Rightarrow \text{(E c1 p)}$$

Generalized for *comp p*, *c*1 and *msg p* to obtain *axiomHImpE_comp_p_intercept_msg_p*:

$$\forall \text{ B : nat} \rightarrow \text{Agent} * \text{Action,}$$
$$\text{B} \vdash \text{H (comp p)(intercept (msg p))} \Rightarrow \text{E (comp p)(intercept (msg p))}$$

3. After, we apply Lemma *eventually_imply*:

$$\forall \text{ p q A, } (\forall \text{ B, B} \vdash \text{p} \Rightarrow \text{q}) \rightarrow \forall \text{ B, B} \vdash \text{Eventually A p} \Rightarrow \text{Eventually A q}$$

For *comp p* and *msg p*.

Therefore, we have *ev_H_compP_intercept_msgP*:

$$\text{B} \vdash \text{Eventually Any (H (comp p)(intercept (msg p)))}$$
$$\Rightarrow \text{Eventually Any (E (comp p)(intercept (msg p)))}$$

4. Then, we use Lemma *satImpl_elim*:

$$\forall \text{ B p q, (B} \vdash \text{p} \Rightarrow \text{q)} \rightarrow \text{((B} \vdash \text{p)} \rightarrow \text{(B} \vdash \text{q))}$$

Generalized for *ev_H_compP_intercept_msgP* and *eventually_h_comp_p_intercept_msg_p*.

As the result, we get *H_c1_inject_msgP_impl_E_c1_inject_msgP*:

$$\text{B} \vdash \text{Eventually Any (E (comp p)(intercept (msg p)))}$$

5. We use Lemma *prec_eventually*:

   ∀ B p q A, (B ⊢p << q)→ (B ⊢Eventually A q)→ (B ⊢Eventually A p)

   Applied for *axiomInjectPrecIntercept_compp_c1_msg_p_d* and *H_c1_inject_msgP_impl_E_c1_inject_msgP*.
   Thus, we obtain *H_c1_inject_msgP_impl_E_c1_inject_msgP*:

   B ⊢ Eventually Any (E (comp p)(intercept (msg p)))

   Consequently, we deduce *ev_c1_inject_msgP*:

   B ⊢ Eventually Any (H c1 (inject (msg p))

It shows that the behavior *B* satisfies eventually that *c*1 injected *msg p*. Listing B.20 shows the corresponding part of the proof using Coq.

```
generalize (axiomInjectPrecIntercept B (comp p) c1 (msg p)); intro
    axiomInjectPrecIntercept_comp_p_c1_msg_p_d.
generalize(fun B → axiomHImpE B (comp p) (intercept (msg p))); intro
    axiomHImpE_comp_p_intercept_msg_p.
generalize (eventually_imply _ _ Any axiomHImpE_comp_p_intercept_msg_p B);
clear axiomHImpE_comp_p_intercept_msg_p;
intro ev_H_compP_intercept_msgP.
generalize (satImpl_elim B _ _ ev_H_compP_intercept_msgP eventually_h_comp_p_intercept_msg_p);
clear ev_H_compP_intercept_msgP;
intro H_c1_inject_msgP_impl_E_c1_inject_msgP.
generalize (prec_eventually _ _ _ _ axiom2_comp_p_c1_msg_p_d
    H_c1_inject_msgP_impl_E_c1_inject_msgP);
clear axiom2_comp_p_c1_msg_p_d H_c1_inject_msgP_impl_E_c1_inject_msgP;
intro ev_c1_inject_msgP.
```

Listing B.20: Coq confidentiality proof step 7

Step 8. We show that since the behavior *B* satisfies that eventually *c*1 injected the message *msg p*, the behavior *B* also satisfies that eventually *c*1 was able to inject *msg p*.

Starting from *ev_c1_inject_msgP*

B ⊢ Eventually Any (H c1 (inject (msg p)))

1. First, we use Axiom *axiomHImpE* :

   $$\forall \text{ B c1 p, B} \vdash \text{(H c1 p)} \Rightarrow \text{(E c1 p))}$$

   Generalized for the component *c*1 doing an *inject* for a message *msg p*. We obtain *axiomHImpE_c*1*_inject_msgP* :

   $$\forall \text{ B : nat} \rightarrow \text{Agent} * \text{Action, B} \vdash \text{H c1 (inject (msg p))} \Rightarrow \text{E c1 (inject (msg p))}$$

2. Then, using lemma *eventually_imply*

   $$\forall \text{ B p q, (B} \vdash \text{p} \Rightarrow \text{q)} \rightarrow \text{((B} \vdash \text{p)} \rightarrow \text{(B} \vdash \text{q))}$$

   Applied for *axiom*1*_c*1*_inject_msgP*.
   We get *H_c*1*_inject_msgP_impl_E_c*1*_inject_msgP*:

   $$\text{B} \vdash \text{Eventually Any (E c1 (inject (msg p)))}$$

3. Next, from Lemma *satImpl_elim*:

   $$\text{B} \vdash \text{Eventually Any (H c1 (inject (msg p)))}$$
   $$\Rightarrow \text{Eventually Any (E c1 (inject (msg p)))}$$

   Using *H_c*1*_inject_msgP_impl_E_c*1*_inject_msgP* and *ev_c*1*_inject_msgP*, we come by *E_c*1*_inject_msgP*:

   $$\text{B} \vdash \text{Eventually Any (E c1 (inject (msg p)))}$$

Thus, we prove that the behavior *B* satisfies eventually *c*1 was able to inject the message protected by the policy *msg p*. Listing B.21 shows the corresponding part of the proof using Coq.

```
generalize(fun B → axiomHImpE B c1 (inject (msg p))); intro axiomHImpE_c1_inject_msgP.
generalize (eventually_imply _ _ Any axiomHImpE_c1_inject_msgP B);
clear axiomHImpE_c1_inject_msgP;
intro H_c1_inject_msgP_impl_E_c1_inject_msgP.
generalize (satImpl_elim B _ _ H_c1_inject_msgP_impl_E_c1_inject_msgP ev_c1_inject_msgP);
clear H_c1_inject_msgP_impl_E_c1_inject_msgP;
intro E_c1_inject_msgP.
```

Listing B.21: Coq confidentiality proof step 8

Step 9. In this step, we show that through some simplifications from the previous hypotheses, the component *comp p* declared in the policy *p* must be the receiver *Rcv* (*msg p*) of the message *msg p* declared in the policy *p*. Then, we rewrite the corresponding terms in all our hypotheses.

Starting from hypothesis *eventually_e_inj_m_impl_valid_policy* :

$$\text{B} \vdash \text{Eventually Any (E c1 (inject (msg p)))} \rightarrow \text{msg p} = \text{msg p} \land \text{Rcv (msg p)} = \text{comp p}$$

1. First, as we know *E_c1_inject_msgP* is true:

   $$\text{B} \vdash \text{Eventually Any (E c1 (inject (msg p)))}$$

   We obtain *policy* :

   $$\text{msg p} = \text{msg p} \land \text{Rcv (msg p)} = \text{comp p}$$

2. Then, we break apart the conjunction in *policy* and keep only the right part as *policyRight* :

   $$\text{Rcv (msg p)} = \text{comp p}$$

3. Therefore, we can rewrite terms using *policyRight* in our hypothesis. This way we obtain the following updated hypothesis:

   - *c3_not_in_c1_c2* (from *Step 5*):

     $$\neg\, \text{In (Rcv (msg p))[c1; c2]}$$

   - *c3_eventually_enable_to_get_d* (from *Step 5*):

     $$\text{B} \vdash \text{Eventually Any (E (Rcv (msg p))(get\_pld (msg p) d))}$$

- *c1_injected_m_to_c2* (from *Step 5*) :

$$\begin{aligned}&\mathrm{B} \vdash (\text{H c1 (inject (msg p)) \&\& has\_rcv (msg p)c2)}\\ &<< \text{E (Rcv (msg p))(get\_pld (msg p)d)}\end{aligned}$$

- *eventually_h_comp_p_intercept_msg_p* (from *Step 6*) :

$$\mathrm{B} \vdash \text{Eventually Any (H (Rcv (msg p))(intercept (msg p)))}$$

Listing B.22 shows the corresponding part of the proof using Coq.

```
generalize (eventually_e_inj_m_impl_valid_policy  E_c1_inject_msgP).
clear E_c1_inject_msgP;
intro policy.
destruct policy as [_ policyRight].
rewrite ← policyRight in ∗; clear policyRight.
```

Listing B.22: Coq confidentiality proof step 9

Step 10. In this step, we show that *c*2 must also be the receiver *Rcv* (*msg p*) of the message *msg p* of the policy *p*, which in turn contradicts the previous result and conclude the proof.

Starting from hypothesis *c1_injected_m_to_c2* :

$$\mathrm{B} \vdash (\text{H c1 (inject (msg p)) \&\& has\_rcv (msg p)c2)} << \text{E (Rcv (msg p))(get\_pld (msg p)d)}$$

1. First, we generalize Lemma *prec_eventually* :

   $$\forall \text{ B p q A, } (\mathrm{B} \vdash \text{p} << \text{q}) \rightarrow (\mathrm{B} \vdash \text{Eventually A q}) \rightarrow (\mathrm{B} \vdash \text{Eventually A p})$$

   For *c1_injected_m_to_c2* and *c3_eventually_enable_to_get_d*.
   This way we get *ev_H_c1_inject_msgP*:

   $$\mathrm{B} \vdash \text{Eventually Any(H c1 (inject (msg p)) \&\& has\_rcv (msg p)c2))}$$

2. Next, we apply Lemma *satEventuallyAnd_elim* :

   $$\mathrm{B} \vdash \forall \text{ B p q, } (\mathrm{B} \vdash \text{p \&\& q}) \rightarrow ((\mathrm{B} \vdash \text{p}) \wedge (\mathrm{B} \vdash \text{q}))$$

in $ev\_h\_c1\_inject\_msgP\_and\_hasRcv\_msgP\_c2$:

$$\text{B} \vdash \text{Eventually Any (H c1 (inject (msg p)) \&\& has\_rcv (msg p)c2)}$$

This way we get :

$$\text{B} \vdash \text{Eventually Any (H c1 (inject (msg p)))} \land \text{B} \vdash \text{Eventually Any (has\_rcv (msg p)c2)}$$

3. Then, by breaking apart the conjunction and keeping only the right part we obtain $ev\_hasRcv\_msg2\_c2$:

$$\text{B} \vdash \text{Eventually Any (has\_rcv (msg p)c2)}$$

4. Then, we apply Axiom $HasRcv$:

$$\forall \text{ B A r m, (B} \vdash \text{Eventually A (has\_pld m r))} \rightarrow \text{r} = \text{Pld m}$$

in $ev\_hasRcv\_msg2\_c2$. As a result, we come by $ev\_hasRcv\_msg2\_c2$:

$$\text{c2} = \text{Rcv (msg p)}$$

5. Finally, we substitute these terms in our hypothesis. This way we obtain the following updated hypothesis:

   - $c3\_not\_in\_c1\_c2$ (from *Step 9*) :

$$\neg \text{ In (Rcv (msg p))[c1; Rcv (msg p)]}$$

Hypothesis $c3\_not\_in\_c1\_c2$ shows a contradiction, hence, as Coq IDE shows, the proof is complete. **Q.E.D.**

Listing B.23 shows the corresponding part of the proof using Coq.

```
generalize (prec_eventually _ _ _ _ c1_injected_m_to_c2 c3_eventually_enable_to_get_d );
clear c1_injected_m_to_c2 c3_eventually_enable_to_get_d ;
intro ev_h_c1_inject_msgP_and_hasRcv_msgP_c2.
apply satEventuallyAnd_elim in ev_h_c1_inject_msgP_and_hasRcv_msgP_c2.
destruct ev_h_c1_inject_msgP_and_hasRcv_msgP_c2 as [_ ev_hasRcv_msg2_c2].
apply HasRcv in ev_hasRcv_msg2_c2.
subst.
apply c3_not_in_c1_c2.
```

```
  simpl.
  auto.
Qed.
```

Listing B.23: Coq confidentiality proof step 10

Glossary

Alloy

Alloy is an open source language and analyzer for software modeling. It has been used in a wide range of applications, from finding holes in security mechanisms to designing telephone switching networks. (taken from `http://alloytools.org/`)

API

Application Programming Interface

CBD

Component-Based Development

CBSE

Component Based Software Engineering

CIAA

CIAA is security objectives classification references. It include classical CIA triad (*Confidentiality*, *Integrity*, *Availability*) and its extension to include the fourth pillar of *Authenticity* named CIAA quartet [30].) Therefore, CIAA classifies security objectives into four categories: *Confidentiality*, *Integrity*, *Authenticity*, and *Availability.*

CLASP

see: Comprehensive, Lightweight Application Security Process

Comprehensive, Lightweight Application Security Process

The Comprehensive, Lightweight Application Security Process (CLASP) is a process, developed by the OWASP Foundation, providing a well-organized and structured approach for moving security concerns into the early stages of the software development life cycle, whenever possible.

Coq

Coq is a formal proof management system. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs. (taken from `https://coq.inria.fr/`)

DSL

Domain Specific Language

DSM

Domain Specific Modeling

DSM

Distributed Shared Memory

DSML

Domain Specific Modeling Language

Eclipse Modeling Framework Technology

The Eclipse Modeling Framework Technology (EMFT) project exists to incubate new technologies that extend or complement EMF. *see also:* Eclipse Modeling Framework

Eclipse Modeling Framework

The Eclipse Modeling Framework is an Eclipse-based modeling framework and code generation facility for building tools and other applications based on a structured data model.

Ecore

Ecore is a reference implementation of OMG's EMOF. *see also:* Eclipse Modeling Framework

EMF

see: Eclipse Modeling Framework

EMFT

see: Eclipse Modeling Framework Technology

EMOF

Essential MOF

IDE

see: Integrated Development Environment

IEEE

Institute of Electrical and Electronics Engineers

Integrated Development Environment

An Integrated Development Environment is software that consolidates the basic tools needed for software testing and writing.

ISO

International Organization for Standardization

MBE

Model-Based Engineering

MDD

Model-Driven Development

MDE

Model-Driven Engineering

Meta-Object Facility

The Meta-Object Facility (MOF) is an Object Management Group standard for model-driven engineering, defining a meta-metamodel.

Microsoft STRIDE

Microsoft STRIDE is security model which categorizes different types of threats and simplifies the overall security conversations. It classifies threats into six categories: *Spoofing*, *Tampering*, *Repudiation*, *Information Disclosure*, *Denial of Service*, and *Elevation of Privilege*.

Microsoft Security Development Life cycle

Microsoft Security Development Life cycle (SDL) is a security-oriented development lifecycle proposed by Microsoft.

MOF

see: Meta-Object Facility

MPS

Message Passing System

NIST

National Institute of Standards and Technology

Object Constraint Language

Object Constraint Language (OCL), a declarative language for describing rules applying to OMG-style metamodels and providing constraint and object queries.

Object Management Group

Object Management Group (OMG), an international, open membership, not-for-profit technology standards consortium.

OCL

see: Object Constraint Language

OMG

see: Object Management Group

Open Web Application Security Project

The Open Web Application Security Project is a nonprofit foundation that works to improve the security of software. Through community-led open-source software projects, hundreds of local chapters worldwide, tens of thousands of members, and leading educational and training conferences, the OWASP Foundation is the source for developers and technologists to secure the web. (taken from `https://owasp.org/`

OWASP

see: Open Web Application Security Project

PBSE

Pattern-Based System and Software Engineering

RPC

Remote Procedure Call

S&D

Security and Dependability

SDL

see: Microsoft Security Development Life cycle

Touchpoints

McGraw's Touchpoints is a security approach based touchpoints between security concepts and concepts generally used in process models.

UML

see: Unified Modeling Language

Unified Modeling Language

Unified Modeling Language (UML) is a general-purpose modeling language in the field of software engineering and is defined by the Object Management Group.

Xtend

Xtend a statically typed programming language that produces understandable Java code. Xtend is based on the Java programming language in terms of syntax and semantics, however it improves on several aspects. This aspects made Xtend well-suited to the task of code generation. It is fully integrated into Xtext.

Xtext

Xtext is a framework for development of programming languages and domain specific languages. It covers all aspects of a complete language infrastructure, from parsers, over linker, compiler or interpreter to fully-blown top-notch Eclipse IDE integration. It comes with great defaults for all these aspects which at the same time can be easily tailored to your individual needs. (taken from `https://eclipse.org/Xtext/`)