

# **Enabling the Automation of Safety and Security Co-Analysis through Lightweight Semantics**

Dissertation  
an der Fakultät für Mathematik, Informatik und Statistik  
der Ludwig-Maximilians-Universität München

eingereicht von  
Yuri Gil Dantas

München, den 21.09.2023

Erstgutachter: PD. Dr. Ulrich Schöpp  
Zweitgutachter: Professor Dr. Ludovic Apvrille  
Drittgutachter: Professor Dr. Peter Csaba Ölveczky  
Tag der mündlichen Prüfung: 17.05.2024

**Eidesstattliche Versicherung**

Hiermit erkläre ich an Eidesstatt, dass die Dissertation von mir selbstständig, ohne unerlaubte Beihilfe angefertigt ist.

---

Gil Dantas, Yuri  
Name, Vorname

---

München, 08.07.2024  
Ort, Datum

# Contents

|   |             |
|---|-------------|
| <b>Abstract</b>   | <b>viii</b> |
| <b>Acknowledgements</b>   | <b>xiii</b> |
| <b>Publications</b>   | <b>xv</b>   |
| <b>1 Introduction and Background</b>  | <b>1</b>    |
| 1.1 Safety Activities . . . . .   | 3           |
| 1.2 Security Activities . . . . .   | 5           |
| 1.3 Safety and Security Interactions . . . . .  | 7           |
| <b>2 Methodology and Highlights of Research Outcomes</b>  | <b>11</b>   |
| 2.1 Lightweight Semantics for Architecture Artifacts . . . . .  | 15          |
| 2.2 Lightweight Semantics for Safety and Security Artifacts . . . . .                                 | 17          |
| 2.3 Lightweight Semantics for Safety and Security Architecture Patterns . . . . .                     | 21          |
| 2.4 Safety and Security Reasoning . . . . .   | 25          |
| 2.4.1 Safety Reasoning . . . . .  | 25          |
| 2.4.2 Security Reasoning . . . . .  | 27          |
| 2.4.3 Reasoning on Safety and Security Consequences . . . . .   | 29          |
| 2.5 Connecting Lightweight Semantics with Formal Verification . . . . .                               | 31          |
| 2.6 Discussion . . . . .  | 33          |
| <b>3 Supplementary Related Work</b>   | <b>36</b>   |
| <b>4 Knowledge Representation and Reasoning for Safety System Architectures</b>                       | <b>42</b>   |
| <b>5 Integrating Safety Architecture Patterns into MBSE: A Plugin for Automated Pattern Synthesis</b> | <b>58</b>   |
| <b>6 An Intruder Model for Automotive Service-Oriented Architectures</b>                              | <b>94</b>   |
| <b>7 Knowledge Representation and Reasoning for Security System Architectures</b>                     | <b>107</b>  |

---

|    |  |     |
|----|--|-----|
| 8  | Integrating Security Knowledge in MBSE: A Plugin for Automated Synthesis   | 120 |
| 9  | Knowledge Representation and Reasoning for Safety and Security Co-Analysis | 141 |
| 10 | A Framework for Assessing the Safety and Security of CACC Systems          | 171 |
| 11 | Conclusion   | 180 |

# List of Figures

|      |  |    |
|------|--|----|
| 2.1  | Breakdown of the thesis contributions presented as scientific articles. . . .            | 12 |
| 2.2  | Example of metadata from logical architecture and allocations. . . . .                   | 16 |
| 2.3  | Example of metadata from safety artifacts (HARA + STPA). . . . .                         | 18 |
| 2.4  | Example of metadata from safety artifacts (HARA + FTA). . . . .                          | 19 |
| 2.5  | Example of metadata from security artifacts (TARA). . . . .                              | 20 |
| 2.6  | Metadata (instantiation) from the Heterogeneous Duplex pattern. . . . .                  | 22 |
| 2.7  | Metadata (attributes and requirements) from the Heterogeneous Duplex<br>pattern. . . . . | 22 |
| 2.8  | Metadata (instantiation) from the M. Authentication pattern. . . . .                     | 23 |
| 2.9  | Metadata (attributes and requirements) from the M. Authentication pattern.               | 24 |
| 2.10 | Examples of safety and security consequences caused by architecture patterns.            | 30 |
| 2.11 | Illustration of both initial state and search pattern of our framework. . . .            | 32 |

# List of Tables

|     |  |    |
|-----|--|----|
| 2.1 | Mapping from safety artifacts (HARA + STPA) to security artifacts. . . . | 27 |
|-----|--|----|

# Abstract

This thesis proposes methods to enable the automation of safety and security co-analysis activities. Safety and security co-analysis consists of activities involving safety and security, and their interactions on the left side of the V-model. The application of safety and security co-analysis is especially relevant to safety-critical systems, which are systems that may cause harm to people or the environment. Examples of safety-critical systems are automated driving systems and aircraft flight control systems. Integrating safety and security co-analysis poses several challenges. Safety and security have different goals, i.e., safety aims to protect the environment from the system, while security aims to protect the system from the environment. It can be challenge to ensure that safety and security do not conflict with each other. Addressing these potential conflicts late in the development process may result in delays and additional costs for industries. Safety and security standards (e.g., ISO 26262 and ISO/SAE 21434 for automotive) recommend that safety and security activities are initially performed on the left side of the V-model, e.g., during the design of the system architecture. To this end, Hazard Analysis and Risk Assessment (HARA) and, e.g., Fault Tree Analysis (FTA), are performed to identify safety artifacts, such as hazards, faults, and failures. Security-wise, Threat Analysis and Risk Assessment (TARA) activities are performed to identify security artifacts, such as identification of assets, threat scenarios, and attack paths. Several of the activities performed within safety and security analysis are manually performed by experts. Once safety and security artifacts are identified, experts select and instantiate safety and security architecture patterns to address the identified safety and security artifacts, respectively. Examples of safety and security architecture patterns are Triple Modular Redundancy and Firewall, respectively. Safety and security architecture patterns are also manually selected and instantiated by experts. This manual work is required because such architecture patterns are documented in textual and graphical formats with limited support for automation. The limited support for automation is due to the absence of explicit semantics associated with key artifacts computed by safety and security activities. Due to the complexity of safety-critical systems, it becomes challenging for experts to manually handle safety and security analysis processes.

This thesis proposes lightweight semantics for key artifacts computed by safety and security activities throughout the design of safety-critical systems. By lightweight semantics, we mean a defined vocabulary and corresponding semantics for which the meaning is uniformly understood in the corresponding domain. For example, in the automotive domain, the terms ECU and asset are well-known by system architects and security engineers,



respectively. By incorporating lightweight semantics for key artifacts, machines may understand their meaning, interpret, and make informed decisions based on the meaning of such artifacts. This thesis anticipates the development of a language enabling the precise specification of key artifacts and safety and security architecture patterns.

This thesis proposes a Domain-Specific Language (DSL) for introducing lightweight semantics to system architecture artifacts, safety and security artifacts, and safety and security architecture patterns. The proposed DSL has been implemented in a logic programming solver. Within the solver, we propose reasoning rules to enable the automation of safety and security co-analysis activities, including the recommendation of safety and security architecture patterns. Our DSL and automated reasoning rules have been implemented following a Model-Based Systems Engineering (MBSE) approach, which is often used in the automotive and avionics industry. The use of this approach enabled us to integrate the implemented DSL and reasoning rules into MBSE plugins, which function in the background as a part of the Model-Based Systems Engineering plugin. Through this integration, we propose safety and security MBSE plugins to reduce the workload and effort required by safety and security engineers during the early stages of system development.

This thesis also investigates the benefits of the introduced lightweight semantics when applied to activities conducted on the right side of the V-model. Specifically, our focus lies on the verification activity, which determines whether the developed system fulfills the requirements specified on the left side of the V-model. Formal verification is a well-known method used to automate checks on whether the system meets such requirements. We develop a formal framework to assess how the safety of Cooperative Adaptive Cruise Control (CACC) systems is affected by intruders attacking the communication channels of the system. The developed framework enables the specification and evaluation of security mechanisms. This thesis investigates whether the security requirements derived from security architecture patterns (left side of the V-model) may serve as input for the verification activity (right side of the V-model). We anticipate that such inputs (i.e., security requirements) required for formal verification are derived from the artifacts computed through the application of lightweight semantics. That is, by leveraging the defined vocabulary and meanings provided by lightweight semantics, we aim to enable the computation of requirements that may subsequently be utilized as inputs for formal verification tools.

We draw the conclusion that this thesis proposes solutions to address an important real-world problem. We believe that the introduction of lightweight semantics opens up a range of opportunities to enable the automation of safety and security activities, as the ones achieved by this thesis. The developed tools have been validated through industrial cases studies from the automotive domain. We have also shown that the developed tools may be used in other domains, such as the unmanned air vehicle domain.

# Zusammenfassung

Diese Dissertation schlägt Methoden vor, um die Automatisierung von Co-Analyse-Aktivitäten für funktionale Sicherheit und Cybersicherheit zu ermöglichen. Die Co-Analyse von funktionaler Sicherheit und Cybersicherheit umfasst Aktivitäten auf der linken Seite des V-Modells, die sich mit funktionaler Sicherheit, Cybersicherheit sowie deren Wechselwirkungen beschäftigen. Eine Co-Analyse von funktionaler Sicherheit und Cybersicherheit ist besonders für sicherheitskritische Systeme relevant, die Schaden für Menschen oder die Umwelt verursachen können. Beispiele für sicherheitskritische Systeme sind automatisierte Fahrassistenten und Steuerungssysteme von Flugzeugen. Die Integration der Co-Analyse von funktionaler Sicherheit und Cybersicherheit stellt mehrere Herausforderungen dar. Funktionale Sicherheit und Cybersicherheit haben unterschiedliche Ziele, d.h., funktionale Sicherheit zielt darauf ab, die Umwelt vor dem System zu schützen, während Cybersicherheit darauf abzielt, das System vor der Umwelt zu schützen. Es kann eine Herausforderung sein, sicherzustellen, dass funktionale Sicherheit und Cybersicherheit nicht miteinander in Konflikt geraten. Die Behebung dieser potenziellen Konflikte spät im Entwicklungsprozess kann zu Verzögerungen und zusätzlichen Kosten für die Industrie führen. Normen für funktionale Sicherheit und Cybersicherheit (z.B. ISO 26262 und ISO/SAE 21434 für die Automobilindustrie) empfehlen, dass funktionale Sicherheits- und Cybersicherheits-Aktivitäten zunächst auf der linken Seite des V-Modells durchgeführt werden, z.B. während des Designs der Systemarchitektur. Zu diesem Zweck werden Gefährdungsanalysen und Risikobewertungen (Hazard Analysis and Risk Assessment – HARA) sowie z.B. Fehlerbaumanalysen (Fault Tree Analysis – FTA) durchgeführt, um funktionale Sicherheitsartefakte wie Gefährdungen, Fehler und Ausfälle zu identifizieren. Sicherheitsseitig werden Bedrohungsanalysen und Risikobewertungen (Threat Analysis and Risk Assessment – TARA) durchgeführt, um Cybersicherheitsartefakte wie die Identifizierung von Vermögenswerten, Bedrohungsszenarien und Angriffswege zu identifizieren. Viele der innerhalb der funktionalen Sicherheits- und Cybersicherheitsanalysen durchgeführten Aktivitäten werden manuell von Experten durchgeführt. Sobald funktionale Sicherheits- und Cybersicherheitsartefakte identifiziert sind, wählen und instanzieren Experten funktionale Sicherheits- und Cybersicherheitsarchitektur-Muster, um die identifizierten funktionalen Sicherheits- und Cybersicherheitsartefakte jeweils zu adressieren. Beispiele für funktionale Sicherheits- und Cybersicherheitsarchitektur-Muster sind Dreifachredundanz und Firewall. Diese Architekturmuster werden ebenfalls manuell von Experten ausgewählt und instanziiert. Diese manuelle Arbeit ist erforderlich, da solche Architektur-Muster in textuellen und grafischen Formaten dokumentiert sind, mit begrenzter

Unterstützung für die Automatisierung. Die begrenzte Unterstützung für die Automatisierung ist auf das Fehlen expliziter Semantik zurückzuführen, die mit den wichtigsten Artefakten verbunden ist, die durch funktionale Sicherheits- und Cybersicherheitsaktivitäten berechnet werden. Aufgrund der Komplexität sicherheitskritischer Systeme wird es für Experten schwierig, die funktionalen Sicherheits- und Cybersicherheitsanalyseprozesse manuell zu bewältigen.

Diese Dissertation schlägt leichtgewichtige Semantiken für Schlüsselartefakte vor, die durch funktionale Sicherheits- und Cybersicherheitsaktivitäten während des Designs sicherheitskritischer Systeme berechnet werden. Unter leichtgewichtigen Semantiken verstehen wir ein definiertes Vokabular mit einer zugeordneten Semantik, deren Bedeutung im jeweiligen Bereich einheitlich verstanden wird. Beispielsweise sind in der Automobilbranche Begriffe wie ECU und schützenswertes Gut bei Systemarchitekten und Cybersicherheitsingenieuren bekannt. Durch die Einbindung leichtgewichtiger Semantiken für Schlüsselartefakte können Maschinen deren Bedeutung verstehen, interpretieren und informierte Entscheidungen auf Basis der Bedeutung dieser Artefakte treffen. Diese Dissertation antizipiert die Entwicklung einer Sprache, die die präzise Spezifikation von Schlüsselartefakten und funktionalen Sicherheits- und Cybersicherheitsarchitektur-Mustern ermöglicht.

Diese Dissertation schlägt eine domänenspezifische Sprache (Domain-Specific Language – DSL) vor, um leichtgewichtige Semantiken in Systemarchitektur-Artefakte, funktionale Sicherheits- und Cybersicherheitsartefakte sowie funktionale Sicherheits- und Sicherheitsarchitektur-Muster einzuführen. Die vorgeschlagene DSL wurde in einem Logikprogrammiersolver implementiert. Mithilfe des Solvers schagen wir Schlussregeln vor, um die Automatisierung von funktionalen Sicherheits- und Cybersicherheits-Co-Analyse-Aktivitäten zu ermöglichen, einschließlich der Empfehlung von funktionalen Sicherheits- und Cybersicherheitsarchitektur-Mustern. Die Implementierung der DSL und der automatisierten Schlussregeln folgt einem Ansatz des modellbasierten Systems Engineering (Model-Based Systems Engineering – MBSE), der häufig in der Automobil- und Luftfahrtindustrie verwendet wird. Durch die Verwendung dieses Ansatzes konnten wir die implementierte DSL und die Schlussregeln in MBSE-Plugins integrieren, die im Hintergrund als Teil des modellbasierten Systems Engineering-Plugins funktionieren. Mit dieser Integration schlagen wir funktionale Sicherheits- und Cybersicherheits-MBSE-Plugins vor, um den Arbeitsaufwand für Ingenieure, die sich mit funktionaler Sicherheit oder Cybersicherheit beschäftigen, in den frühen Phasen der Systementwicklung zu reduzieren.

Diese Dissertation untersucht auch die Vorteile der eingeführten leichtgewichtigen Semantiken, wenn sie auf Aktivitäten auf der rechten Seite des V-Modells angewendet werden. Insbesondere konzentrieren wir uns auf Verifikationsaktivitäten zur Evaluierung, ob das entwickelte System die auf der linken Seite des V-Modells festgelegten Anforderungen erfüllt. Formale Verifikation ist eine bekannte Methode, um automatisierte Überprüfungen durchzuführen, ob das System diese Anforderungen erfüllt. Wir entwickeln ein formales Rahmenwerk, um zu bewerten, wie die funktionale Sicherheit von kooperativen adaptiven Tempomatsystemen (Cooperative Adaptive Cruise Control – CACC) durch Angreifer, die die Kommunikationskanäle des Systems angreifen, beeinträchtigt wird. Das entwickelte Rahmenwerk ermöglicht die Spezifikation und Bewertung von Cybersicherheitsmechanismen.

Diese Dissertation untersucht, ob die aus Cybersicherheitsarchitektur-Mustern abgeleiteten Cybersicherheitsanforderungen (linke Seite des V-Modells) als Eingabe für Verifikationsaktivitäten (rechte Seite des V-Modells) dienen können. Wir erwarten, dass solche Eingaben (d.h. Cybersicherheitsanforderungen), die für die formale Verifikation erforderlich sind, aus den Artefakten abgeleitet werden, die durch die Anwendung leichtgewichtiger Semantiken berechnet werden. Das heißt, durch die Nutzung des definierten Vokabulars und der Bedeutungen, die durch leichtgewichtige Semantiken bereitgestellt werden, wollen wir die Berechnung von Anforderungen ermöglichen, die anschließend als Eingaben für formale Verifikationstools genutzt werden können.

Wir kommen zu dem Schluss, dass diese Dissertation Lösungen für ein in der Praxis wichtiges Problem vorschlägt. Wir glauben, dass die Einführung leichtgewichtiger Semantiken eine Vielzahl von Möglichkeiten eröffnet, um die Automatisierung von funktionalen Sicherheits- und Cybersicherheitsaktivitäten zu ermöglichen, wie sie durch diese Dissertation erreicht wurden. Die entwickelten Werkzeuge wurden durch industrielle Fallstudien aus der Automobilbranche validiert. Wir haben auch gezeigt, dass die entwickelten Werkzeuge in anderen Bereichen, wie dem Bereich unbemannter Luftfahrzeuge, verwendet werden können.

# Acknowledgements

I would like to thank my supervisors, Ulrich Schöpp and Vivek Nigam, for their support and guidance throughout my doctoral journey. I am truly grateful for the supervision meetings we had, where they provided me insightful feedback and guidance that have been instrumental in shaping the direction and of the results presented in this thesis. In particular, Ulrich Schöpp also provided valuable assistance throughout my doctoral journey by offering clinical feedback and engaging in close collaboration. I would like to express my gratitude to Vivek Nigam: Kudos to him! Vivek Nigam played a crucial role in my journey towards becoming a researcher. Our paths crossed back in 2013, and at first, I mistook him for a fellow student due to his youthful appearance. Little did I know that Vivek Nigam was a professor and would become the supervisor for my Bachelor's and Master's theses. Vivek Nigam's ability to guide students towards successful careers is remarkable. He has an amazing talent for guiding students in transforming fragments of potential research ideas into exceptional ones. I still remember his anecdotes and insightful stories from 2013. Vivek Nigam used to say "...writing is the only skill that improves as you age. While sight may fade and speed may diminish, writing only gets better with time, as long as you practice." Since that day, I carried those words with me.

I am grateful to the examiners, Ludovic Apvrille and Peter Csaba Ölveczky, for their time and effort to review my doctoral thesis.

I would like to thank fortiss GmbH for providing me with the opportunity to work for this amazing research institute. The resources, infrastructure, and support they provided have been crucial in conducting my research and achieving the results presented in this thesis. I would like to thank my colleagues from fortiss GmbH: Simon Barner, Carmen Cârlan, Tiziano Munaro, Chuangjie Xu, as well as my supervisor Ulrich Schöpp, the scientific managing director Harald Rüss, and the scientific director Alexander Pretschner. I did appreciate the research discussions we had, particularly those focused on model-based systems engineering, as they greatly influenced the results presented in this thesis.

I would like to thank my colleagues from TU Darmstadt: Florian Dewald, Lorenzo Gheri, Richard Grewe, Tobias Hamann, Görkem Kiliç, Matthias Perner, Johannes Schickel, David Schneider, Markus Tasch, Alexandra Weber, and, of course, Heiko Mantel. I want to acknowledge that every one of you has made significant contributions to this achievement.

I would like to thank my friends Cauê Azeredo, Raoni Azeredo, Camilo Cabral, Cassio Cabral, Fernanda Martins Cabral, Arnaldo Gualberto, Hugo Neves, Hugo Noronha, Pere Massanés Padró, Daniel Pedroza, Dorothy Rogers, Glauco de Sousa, José Ivan Vilarouca,

Stefan Weber, and Yasaman Yousefi.

A big shout goes to my family, including my parents, Maria de Fatima Gil Dantas and João Carlos Dantas, my brothers, Sheldon Gil Dantas and Eric Gil Dantas (including, my sister-in-law Karolina Röder), and my uncle, Wellington Gomes Dantas. Additionally, I want to extend my deepest gratitude to my wife, Isadora Aguiar, who always believed in me. Together, you are my home, and I am forever grateful for your love and support.

To everyone who have contributed in any other way, please accept my thanks. This thesis would not have been possible without your support and encouragement.

Thank you all.

# Publications

We have published contributions contained in this thesis at peer-reviewed conferences and journals as follows.

1. Yuri Gil Dantas, Antoaneta Kondeva, and Vivek Nigam: Less Manual Work for Safety Engineers: Towards an Automated Safety Reasoning with Safety Patterns. ICLP Technical Communications 2020: 244-257
2. Yuri Gil Dantas, Tiziano Munaro, Carmen Cârlan, Vivek Nigam, Simon Barner, Shiqing Fan, Alexander Pretschner, Ulrich Schöpp, and Sergey Tverdyshev: A Model-based System Engineering Plugin for Safety Architecture Pattern Synthesis. MOD-ELSWARD 2022: 36-47
3. Yuri Gil Dantas, Tiziano Munaro, Carmen Cârlan, Vivek Nigam, Simon Barner, Shiqing Fan, Alexander Pretschner, Ulrich Schöpp, and Sergey Tverdyshev: A Toolchain for Synthesizing and Validating Safety Architectures. SN Computer Science 2023: Volume 4, Article number: 335.
4. Yuri Gil Dantas, Simon Barner, Pei Ke, Vivek Nigam. and Ulrich Schöpp: Automating Vehicle SOA Threat Analysis Using a Model-Based Methodology. ICISSP 2023: 180-191
5. Yuri Gil Dantas and Ulrich Schöpp: SeCloud: Computer-Aided Support for Selecting Security Measures for Cloud Architectures. ICISSP 2023: 264-275
6. Yuri Gil Dantas, Vivek Nigam, and Ulrich Schöpp: A Model-Based Systems Engineering Plugin for Cloud Security Architecture Design. SN Computer Science 2024: Volume 5, Article number: 553.
7. Yuri Gil Dantas and Vivek Nigam: Automating Safety and Security Co-design through Semantically Rich Architecture Patterns. ACM Transactions on Cyber-Physical Systems 2023: Volume 7, Issue 1, Pages: 1-28
8. Yuri Gil Dantas, Vivek Nigam, and Carolyn L. Talcott: A Formal Security Assessment Framework for Cooperative Adaptive Cruise Control. VNC 2020: 1-8

The contributions listed below have been published but have not been included into this thesis. Article #1 has been published in a workshop, while Article #2 has been published as a whitepaper.

1. Tarik Terzimehic, Simon Barner, Yuri Gil Dantas, Ulrich Schöpp, Vivek Nigam, and Pei Ke: Safety-Aware Deployment Synthesis and Trade-Off Analysis of Apollo Autonomous Driving Platform. ICSCA-C 2023: 309-316
2. Yuri Gil Dantas, Vivek Nigam, and Harald Ruess. Security Engineering for ISO 21434. CoRR, abs/2012.15080, 2020.



# Chapter 1

## Introduction and Background

*Safety-critical systems* are systems that may cause harm to people or the environment [1]. Examples of safety-critical systems are automated driving systems and aircraft flight control systems. These systems consist of software and hardware units implementing safety functions, such as perception functions in automated driving systems. The malfunction behavior of such functions may cause harm, such as serious injuries to people (e.g., driver and passengers) or the environment (e.g., road users). Safety is the top priority for the development of safety-critical systems, as it involves preserving human lives. In simple terms, *safety* is about protecting the involved people and the environment from the system [2], such as addressing internal malfunction behavior of the system that may lead to crashes. Hazard Analysis and Risk Assessment (HARA) is one instance of a safety activity aimed at protecting a system, which involves computing safety artifacts, such as hazards.

Businesses developing safety-critical systems are undergoing a transformation due to the rapid development of highly advanced technologies and complex software solutions. This transformation is, in particular, prominent within the automotive industry, which is focusing on the development of automated and connected vehicles. These vehicles interact or communicate with the environment to improve the overall transportation system performance and driver comfort via cooperation [3]. The introduction of communication capabilities into vehicles brings in security risks whereby intruders may exploit vulnerabilities to gain unauthorized access and control over the vehicle to reduce the safety of passengers. Hence, it is imperative to also prioritize security in the development of (connected) safety-critical systems. In simple terms, *security* is about protecting the system from the environment [2], such as addressing intentional attacks carried out by external intruders. Even though external intruders are the main actors against connected safety-critical systems, it is important to acknowledge that internal intruders also play a role in the security landscape, as discussed in this thesis. Consequently, the scope of security shall be extended to also protect systems against internal intruders. Threat Analysis and Risk Assessment (TARA) consists of several activities aimed at protecting a system from a security perspective, which involves computing security artifacts, such as assets, attack paths and threat scenarios.

The V-model [4] is a representation of a system's development lifecycle. The left side of the V-model represents the early stages of the system development process, starting from

requirements analysis and design. As the development progresses, the right side of the V-model represents the verification and validation activities. Within this context, there is a concept known as safety and security co-analysis. *Safety and security co-analysis* [5, 6] consists of safety and security activities and their interactions performed on the left side of the V-model, i.e., during the system design. Although safety and security activities may be performed independently, their effective collaboration necessitates synchronization between safety and security engineers. These collaborative efforts facilitate the identification of potential synergies, including opportunities for information exchange, as well as conflicts, including safety implications arising from the implementation of security mechanisms.

As emphasized throughout this thesis, many of the safety and security activities, including co-analysis activities, are currently performed manually by safety or security experts. This manual work is required due to the absence of explicit semantics associated with safety and security artifacts computed by safety and security activities, respectively. Our goal is to propose lightweight semantics for key artifacts computed by or relevant to safety and security activities performed throughout the design of safety-critical systems.

By *lightweight semantics*, we mean a defined vocabulary, along with its associated semantics, for which the meaning is uniformly understood in the corresponding domain. The vocabulary shall consist of terms related to safety-critical systems, particularly used during the system design. For example, in the automotive domain, the terms ECU (Electronic Control Unit), hazard, and asset are well-known terms by system engineers, safety engineers, and security engineers, respectively. These terms shall be specified in a manner that enables machines to read and interpret them. In this thesis, the proposed lightweight semantics are embodied in the form of specified logic programming facts and rules to enable automated reasoning. By incorporating lightweight semantics for key artifacts into a logic programming language, machines may understand their meaning, interpret, and make informed decisions based on the meaning of such artifacts. We anticipate the need to enhance the reader's understanding of our lightweight semantics by providing a simple example. Consider the following facts that may be specified in a logic programming solver, such as DLV [7].

```
% Representation of ECUs and components
ecu(adas).
component(perception).
```

```
% Representation of allocations
allocation(perception,adas).
```

In the example above, we specify an ECU, a component, and the allocation of the component, i.e., a component implemented within a specific ECU. The vocabulary consists of the terms `ecu`, `component`, and `allocation`, along with its associated semantics, i.e., `adas` is an `ecu`, `perception` is a `component`, and `perception` is allocated to `ecu`. With this specification, a logic programming solver is able to understand the specified facts and automatically derive all components allocated to a specific ECU. This automation is made possible through the specification of reasoning rules, such as the following rule, which lists all ECU components based on the specified facts.

```
% Rule to make informed decisions - listing components allocated to ECU
list_ecu_components(ECU,COMPONENT) :-
    ecu(ECU),
    component(COMPONENT),
    allocation(COMPONENT,ECU).
```

**Hypothesis:** *The introduction of lightweight semantics to key artifacts advances the state-of-the-art by enabling the automation of several safety and security activities.*

The remainder of this chapter provides a concise overview of safety and security activities undertaken by engineers during the design of safety-critical systems, including the artifacts computed by such activities. Our main focus is on artifacts for which our goal is to provide lightweight semantics in order to enable the automation of their corresponding activities. To streamline our discussion, we concentrate on (automated and connected) vehicle systems. Within this context, we emphasize industrial and research needs, as well as the research questions addressed within the scope of this thesis. While the remainder of this chapter provides a high-level description of safety and security activities, Chapter 2 describes the methodology used by this thesis to provide lightweight semantics to artifacts computed by such activities. In addition, Chapter 2 outlines the methodology used to enable automation based on the proposed lightweight semantics.

## 1.1 Safety Activities

The development of safety-critical systems requires rigorous analysis at all stages of development to ensure the correct behavior of safety functions and minimize the risk of accidents. Moreover, the development of safety-critical systems requires compliance with safety standards and regulations [8], such as the safety standard – ISO 26262 [9] – for electrical and electronic (E/E) vehicle systems.

ISO 26262 [10] requires conducting safety analysis on E/E vehicle systems during the initial phases of development. Based on the system architecture (a.k.a item definition), often designed by system architects, safety engineers perform Hazard Analysis and Risk Assessment (HARA) to identify hazards and define safety goals to address the identified hazards. A *hazard* [9] denotes a potential source of harm, while a *safety goal* [9] denotes a top-level safety requirement to address the identified hazard.

### **Example of a hazard:**

‘The ego vehicle<sup>1</sup> violates the safety distance to other road users or objects on the road’.

### **Example of a safety goal:**

‘Prevent unintended safety distance violation to other road users or objects on the road’.

---

<sup>1</sup>The ego vehicle denotes the vehicle that implements safety functions.

*Automotive Safety Integrity Level* (ASIL) is a classification that denotes the criticality of a hazard. Each hazard is assigned to an ASIL class computed based on the severity (potential harm), exposure (likelihood of its occurrence), and controllability (e.g., driver ability to control the hazard) of the hazard. The ASIL class ranges from A (lowest) to D (highest). A higher ASIL level denotes a more critical hazard that requires more stringent safety mechanisms. Notice that there exists an additional ASIL class known as QM (Quality Management) that does not pose any safety risks and does not require any safety mechanisms. The hazard’s description, along with its ASIL class, is denoted as a *hazardous event*. This thesis often uses both hazard and hazardous event interchangeably.

Safety engineers conduct additional safety analysis to identify potential causes for hazards. To this end, safety engineers may use top-down methods, such as Fault Tree Analysis (FTA) [11] or System Theoretic Process Analysis (STPA) [11]. FTA identifies faults, failures, and minimal cut sets as potential causes for hazards. A *fault* [12] denotes the hypothesized cause of an error (i.e., deviation of the expected behavior of a function). A *failure* [12] is the result of a fault denoting an event that when occurs results in an error. For example, a fault (e.g., a bug) in a software unit implementing a perception function may lead to erroneous outputs (failure). A *minimal cut set* is a set of basic events that when occurring at the same time or in sequence may lead to a top event. Those basic events may be failures that are typically represented as leaf nodes in a fault tree, while the top event may be a hazard. STPA identifies unsafe control actions, and loss scenarios as potential causes for hazards. An *unsafe control action* [11] is a control that, in a particular context and worst-case environment, will lead to a hazard. A *loss scenario* [11] denotes the casual factors that can lead to the unsafe control actions and consequently to hazards. For example, the function perception computing erroneous outputs (loss scenario) may lead to a high acceleration (possibly an unsafe control action) of the ego vehicle, meaning that a collision to a front object is imminent (hazard).

This thesis uses the term *safety artifacts* (or *safety elements*) to denote the artifacts identified by HARA (e.g., hazards), FTA (e.g., faults), or STPA (e.g., loss scenarios).

Safety engineers define a functional safety concept after the identification of safety artifacts. A *functional safety concept* [10] specifies functional safety requirements to achieve the identified safety goals. These safety requirements are specified in terms of safety mechanisms allocated to elements of the system architecture. This thesis refers to safety mechanisms at the architecture level as safety architecture patterns.

A *safety architecture pattern* [13] is an abstract representation for how to solve a general safety problem which occurs over and over in many applications. Specifically, a safety architecture pattern is an architecture solution for addressing recurring safety artifacts identified during the design of the system architecture. We consider two types of safety architecture patterns, namely fault detection pattern and fault tolerant pattern. A *fault detection pattern* deactivates the system in the presence of a failure (triggered by a fault) by either transitioning the system to a safe state or shutting down the system. Monitor-Actuator [13, 14] is an example of a fault detection pattern. A *fault tolerant pattern* ensures that the system will continue to operate in the presence of a failure (triggered by a fault) by providing a redundant component to take over the operation. Heterogeneous Duplex

pattern [13, 14] is an example of a fault tolerant pattern.

In the field of safety-critical systems, catalogs of safety architecture patterns can be found in the literature, such as in [13] and [14]. These patterns are documented in both textual and graphical formats, utilizing pattern templates that describe, e.g., the structure, intent, and specific problem addressed by each pattern. In the current industry landscape, the process of selecting appropriate safety architecture patterns from such catalogs and instantiating them in the system architecture relies heavily on the expertise of safety engineers, i.e., leading the manual work by safety engineers. Due to the complexity of automotive systems, this manual process of pattern selection and instantiation poses significant challenges. According to [15], automotive systems have about 30.000 parts more than 100 million lines of code. This number is expected to increase with the development of automated driving systems. Furthermore, given the fast-paced nature of the automotive industry and the need to deliver several projects within tight deadlines, there is an increased risk of errors introduced by safety engineers. Making erroneous or sub-optimal design choices during the early stages of system design can result in substantial development delays and increased costs.

This thesis aims to introduce lightweight semantics to key safety artifacts and safety architecture patterns. To this end, the thesis conducts an investigation into the existing descriptions of safety artifacts and safety architecture patterns, both described in textual and graphical formats. The aim is to identify the key safety artifacts and which parts of safety architecture patterns that are crucial for enabling the automated recommendation of safety architecture patterns. Through the utilization of lightweight semantics, the thesis anticipates the development of a language enabling the precise specification of safety artifacts and safety architecture patterns. This language shall facilitate the automated recommendation of safety architecture patterns.

This leads to our first research question:

**RQ1:** Can the introduction of lightweight semantics to safety artifacts and safety architecture patterns enable the automated recommendation of safety architecture patterns?

## 1.2 Security Activities

Communication serves as the primary driving force behind automated driving systems [16]. By *communication* (a.k.a. interconnectivity) [17], we mean any direct connection between systems to exchange data or share information resources. Automated driving systems rely on various forms of communication, such as vehicle-to-vehicle (V2V) communication, which allows vehicles to exchange information with one another, enabling activities like forming vehicle platoons [18]. Additionally, vehicle-to-infrastructure (V2I) communication enables vehicles to exchange information with the infrastructure, enabling computations for determining the vehicle's next trajectory based on data obtained from, e.g., traffic lights [19]. Establishing network connections, including wireless connections, enables the exchange of information for V2V and V2I communications. Furthermore, modern vehicles incorporate sensors (e.g., GPS and cameras) to gather data about the vehicle itself and the surrounding environment. This sensor data serves as input information for computing the

trajectory of vehicles driving in an autonomous mode.

There have been several attacks targeting the communication channels of automated driving systems (as documented in the survey by [20]). What is even more concerning is the potential safety implications that such attacks may have on people or road-users. By introducing communication channels, attackers may exploit vulnerabilities to gain unauthorized access to a vehicle, thereby disabling or tampering with safety functions. For example, hackers have shown how to remotely control a Jeep vehicle by manipulating brakes, engine and steering [21]. This attack is a reminder that security vulnerabilities in actual vehicles may be exploited by attackers to compromise passenger safety.

The adoption of new technologies, such as Service-Oriented Architectures (SOA), increases the attack surfaces (beyond external communications) that intruders may exploit. That is, communications within an organization's internal networks is also security relevant, as there exists the potential for intruders to carry out attacks from within targeting critical functions, including safety-related ones. To emphasize this point, we refer to the findings of researchers [22], who have shown that insider intruders may exploit security vulnerabilities in the service discovery mechanism of vehicle SOA to compromise passenger safety.

To tackle security for automotive systems (in particular E/E vehicle systems), ISO/SAE 21434 [23] has been published in 2021. ISO/SAE 21434 requires conducting security analysis on E/E vehicle systems during the early stages of development. To this end, Threat Analysis and Risk Assessment (TARA) are performed to identify security artifacts in the system architecture. TARA consists of seven activities, namely asset identification (including damage scenario identification), impact rating, threat scenario identification, attack path analysis, attack feasibility rating, risk value determination, and risk treatment decision. We provide below the definitions of these activities as outlined in ISO/SAE 21434 [23]. An *asset* is an object (e.g., a function) that has value. An asset has one or more security properties (e.g., integrity or availability). The compromise of the asset's security property may damage the vehicle. This damage is represented as a *damage scenario*, which denotes the adverse consequences involving a vehicle or vehicle function and affecting the road user. The *impact rating* determines the impact of a damage scenario on four categories, namely safety, financial, operational, and privacy. A *threat scenario* denotes a potential cause (e.g., tampering) to compromise the security properties of assets to realize the damage scenario. An *attack path* denotes a set of deliberate actions/steps to realize a threat scenario. The *attack feasibility rating* describes the ease of successfully carrying out the actions of an attack path. The *risk value determination* denotes the risk of the threat scenario. The risk value can be determined based on the impact rating and attack feasibility rating. The *risk treatment decision* denotes a decision on how the risk shall be addressed, such as reducing the risk which requires the utilization of security mechanisms to reduce the identified risk.

This thesis uses the term *security artifacts (or security elements)* to denote the artifacts identified by TARA, such as assets, threat scenarios, and attack paths.

Security goals may be defined to protect assets against threat scenarios, thus reducing the risk. A *security goal* [23] denotes a top-level security requirement associated with one or more threat scenarios. To achieve the defined security goals, security requirements are specified based on suitable security mechanisms. This thesis refers to security mechanisms at the

architecture level as security architecture patterns. A *security architecture pattern* [24, 25] is an architecture solution to address recurring security artifacts identified during the design of the system architecture. Examples of security architecture patterns are Firewall, Message Authentication, and Mutual Authentication patterns.

In the current industry landscape, security engineers perform TARA activities manually. Some activities can be relatively straightforward, such as computing threat scenarios using methodologies like STRIDE [26], and determining risk values through established methods like risk matrices [23]. However, it is important to note that those activities are interdependent. That is, computing threat scenarios relies on the security properties of assets, and calculating risk values depends on factors like impact rating and attack feasibility rating. The other activities, such as attack path analysis, may be time-consuming, requiring a comprehensive analysis of the system. The attack path analysis is also system-specific, dependent on the Original Equipment Manufacturer (OEM) requirements, system vehicle, and the attacker capabilities. Consequently, security engineers may not always be able to reuse results from previous analysis.

There are catalogs available in the literature that describe security architecture patterns, such as [27, 28, 29]. Similar to safety architecture patterns, these catalogs provide documentation in both textual and graphical formats, with limited automation support. Security engineers face the same challenges faced by safety engineers when selecting and instantiating security architecture patterns into the system architecture. With several security architecture patterns [24, 25] available, thoroughly evaluating the implications of each choice is often impractical, especially considering time constraints.

This thesis aims to introduce lightweight semantics to key security artifacts and security architecture patterns. To this end, the thesis conducts an investigation into the existing descriptions of security artifacts and security architecture patterns, both described in textual and graphical formats. The aim is to identify the key security artifacts and which parts of security architecture patterns that are crucial for enabling the automation of TARA activities. Through the utilization of lightweight semantics, the thesis anticipates the development of a language enabling the precise specification of security artifacts and security architecture patterns. This language shall facilitate the automation of TARA activities, including the recommendation of security architecture patterns.

This leads to the second research question:

**RQ2:** Can the introduction of lightweight semantics to security artifacts and security architecture patterns enable the automated computation of TARA activities?

## 1.3 Safety and Security Interactions

Establishing a strong connection between safety and security is of utmost importance in ensuring the overall safety of safety-critical systems. Standards and guidelines for avionics [30] and automotive [31, 23] industries have taken steps towards integrating safety and security. For example, ISO/SAE 21434 recommends that the organization responsible for the security analysis shall identify disciplines interacting with security, and establish

communications channels to those disciplines to coordinate the exchange of information [23]. Safety is a very important discipline for security because the developed safety-critical system is not safe if it is not secure [32]. Therefore, there shall be potential interaction points between safety and security analysis during the design of safety-critical systems. These interactions include when information gathered by safety engineers shall be made available to security engineers and vice versa [33].

Safety and security interactions may lead to trade-offs. These trade-offs may result in either synergies or conflicts between safety and security analysis. Synergies may arise because safety analysis are typically conducted prior to security analysis, allowing for a possible mapping from safety artifacts to security artifacts. This is an approach known as *safety-informed security* (sometimes referred to as security for safety [34]), which refers to protecting safety functions in safety-critical systems from a security perspective. According to Paul [35], safety-informed security is an approach that consists of conducting security analysis independently from safety analysis, while considering relevant information from the safety analysis. In this approach, the security analysis focuses on the safety artifacts identified in the safety analysis, examining how these artifacts may be compromised from a security perspective and proposing security mechanisms to address potential threats. As a result, the scope of the security analysis is bounded by the safety artifacts identified in a safety analysis. A significant benefit of this approach is the full traceability between safety and security analysis, where one can check whether all causes of hazards (e.g., faults and failures in FTA) have traces to the security analysis. Another clear benefit is that the security analysis may become more efficient. For example, security engineers may quickly identify all safety-relevant assets (such as, components of safety patterns) that shall be analyzed from a security perspective. This thesis investigates the benefits of using the safety-informed security approach as part of the safety and security co-analysis process.

Conflicts may arise from the use of safety and security architecture patterns [5]. The deployment of safety architecture patterns may have consequences on security. The components of a safety architecture pattern may be a target for intruders, and therefore they may become assets that need to be analyzed from a security perspective. The other way around may also be possible. The deployment of a security architecture pattern have safety consequences. For example, the deployment of a firewall at a communication channel with safety-relevant information may unintentionally block safety-critical flows. As a result, the deployed security architecture patterns shall be analyzed from a safety perspective.

As mentioned earlier, this thesis aims to introduce lightweight semantics to safety and security artifacts. Our hypothesis is that the introduction of lightweight semantics may also be used to identify synergies between safety and security artifacts, specifically in automating the mapping from safety artifacts to security artifacts. Moreover, it may also be used to enable the automated identification of conflicts that may arise due to the instantiation of safety and security architecture patterns.

This leads to the third research question:

|  |
|--|
| <b>RQ3:</b> Can the introduction of lightweight semantics enable the automated identification of synergies and conflicts that may arise from safety and security analysis? |
|--|



Up to this point, we have discussed safety and security activities performed on the left side of the V-model. The left side of the V-model specifies several artifacts, including safety and security requirements, software requirements, and hardware requirements. The specified safety and security requirements shall be implemented in the system to achieve the corresponding safety and security goals, respectively. The verification activity is a crucial activity on the right side of the V-model. This activity determines whether the developed system fulfills the requirements specified on the left side of the V-model [23]. We substantiate the above statements by mentioning Clauses 9 and 10 from ISO/SAE 21434. Clause 9 includes the specification of security goals and requirements, and Clause 10 includes the implementation and verification of security requirements. In this thesis, security requirements are derived from security architecture patterns.

Although the main focus of this thesis is on safety and security activities on the left side of the V-model, we also consider the verification activity to investigate (i) how the safety of systems may be affected by security attacks, i.e., how safety goals may be violated by attacks carried out by intruders, and (ii) how security architecture patterns may help to both mitigate attacks and ensure the safe behavior of the system.

Consider the promising technology for connected and automotive vehicles, namely Cooperative Adaptive Cruise Control (CACC). CACC is an extension of Adaptive Cruise Control (ACC) to enable vehicles (e.g., a fleet of trucks) to drive in a cooperative manner, a.k.a. platoon [36]. CACC establishes a V2V communication topology to enable vehicle platoon to exchange information with each other. This exchange of information may result in changes in the vehicle's state, such as the vehicle's speed and position. A reasonable safety goal for CACC systems is to "prevent unintended safety distance violation between vehicles in the platoon". This safety goal may be violated by malicious data injected by intruders through communication channels with the intent of maliciously changing the vehicle's state. These attacks may be mitigated by security mechanisms implemented based on security architecture pattern requirements identified during the design of the system.

Formal verification is a well-known method used to automate checks on whether the system fulfills safety and security requirements specified on the left side of the V-model. This thesis investigates whether the security requirements derived from security architecture patterns (left side of the V-model) may be used as input for the verification activity (right side of the V-model) to verify the safe behavior of automotive systems. We expect that such inputs (i.e., security requirements) are derived from the artifacts computed with the help of the proposed lightweight semantics. That is, by leveraging the defined vocabulary and meanings provided by lightweight semantics, we aim to enable the automated computation of requirements that can subsequently be utilized as inputs for formal verification tools, such as TTool [37] or Soft-Agents framework [38].

This leads to the fourth research question:

**RQ4:** Can the introduction of lightweight semantics assist traditional formal verification in verifying safety-critical systems?

In summary, this thesis aims to provide contributions to both the research and industry worlds. From a research perspective, this thesis aims to advance state-of-the-art of safety and

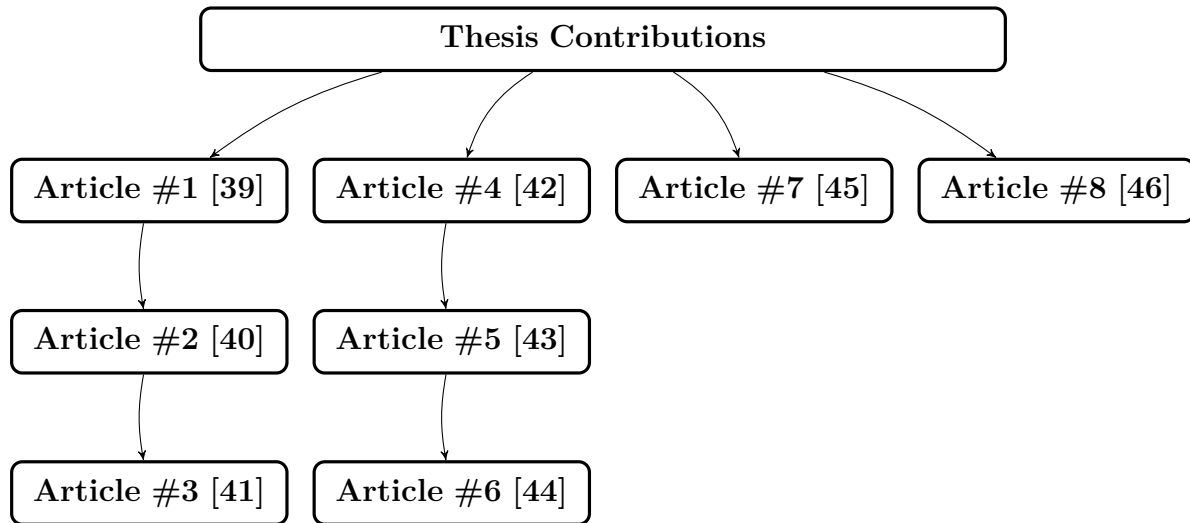
security co-analysis by providing (i) lightweight semantics to artifacts computed throughout the design of safety-critical systems, and (ii) methods to enable the automation of safety and security activities, including co-analysis activities. From an industry perspective, the automation of safety and security activities is a significant contribution in itself. Additionally, this thesis aims to ensure that the artifacts developed within this research align closely with safety and security standards strictly followed by the industry.

## Chapter 2

# Methodology and Highlights of Research Outcomes

This chapter outlines the methodology employed to answer the research questions introduced in Chapter 1. Our goal is to introduce lightweight semantics to key artifacts computed by activities performed on the left side of the V-model. By incorporating lightweight semantics into such artifacts, the intended outcome is to enable the automation of safety and security co-analysis activities. To this end, our methodology focuses on using a model-based systems engineering approach and knowledge representation and reasoning. Additionally, we investigate how the introduced lightweight semantics can enhance and support formal verification processes. Next, we provide a brief description on model-based systems engineering, knowledge representation and reasoning, and formal verification. Figure 2.1 illustrates the breakdown of the results obtained in the thesis. This illustration provides an overview of the articles used in the thesis, along with their respective chapters, highlights the primary methodology utilized by each article, and which research question was tackled by each article. Sections 2.1, 2.2, 2.3, 2.4, and 2.5 provide a concise overview of how this thesis utilized the respective methodology to address the research questions. The results achieved by this thesis were evaluated through realistic case studies from both the automotive domain and the unmanned air vehicle domain.

**Model-based Systems Engineering (MBSE).** According to the INCOSE Systems Engineering Vision [47], *model-based systems engineering (MBSE)* is the formalized application of modeling to support system requirements, design, analysis, verification, and validation of activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases. The intended outcome of MBSE is to replace the traditional document-based approach, where engineers write documents to record information about the developed system. A MBSE approach captures system information, such as system requirements and architecture views (e.g., logical and hardware architectures), using a collection of interconnected models (a.k.a. viewpoints). When changes are made to one model, automated updates are triggered in all related models, ensuring traceability and consistency of information across the system. This interconnectedness of models helps



| Article | Chapter | Methodology | Research Question(s) |
|---------|---------|-------------|----------------------|
| #1 [39] | 4       | KRR         | RQ1                  |
| #2 [40] | 5       | MBSE        | RQ1                  |
| #3 [41] | 5       | MBSE        | RQ1                  |
| #4 [42] | 6       | KRR         | RQ2 and RQ3          |
| #5 [43] | 7       | KRR         | RQ2                  |
| #6 [44] | 8       | MBSE        | RQ2                  |
| #7 [45] | 9       | KRR         | RQ1, RQ2, and RQ3    |
| #8 [46] | 10      | FV          | RQ4                  |

Figure 2.1: Breakdown of the thesis contributions presented as scientific articles.

engineers manage the complexities associated with safety-critical systems throughout the entire development cycle. ISO 26262 [10] and ISO/SAE 21434 [23] require the availability of an item definition prior conducting safety and security analysis, respectively. The *item definition* represents the system architecture in preliminary version or more advanced version depending on the status of the system development. The *system architecture* [9] represents the structure of the system that allows identification of building blocks, their boundaries and interfaces, and includes the allocation of functions or components to hardware and software units. In this thesis, the system architecture (a.k.a. item definition) is expected to be modeled using a model-based systems engineering approach. The MBSE approach is valuable due its comprehensive representation of system information within the models, facilitating the extraction of relevant data for reasoning about the safety and security aspects of the system architecture. Section 2.1 describes the specification of lightweight

semantics obtained from relevant architecture artifacts designed using a MBSE approach.

**Knowledge Representation and Reasoning (KRR).** KRR [48] is the area of Artificial Intelligence (AI) concerned with how knowledge can be symbolically represented and manipulated in an automated fashion by reasoning programs. *Knowledge* denotes what is known in the world by a stakeholder, such as a person or a system. *Representation* denotes how the stakeholder represents that knowledge through symbols. For example, a system architect responsible for designing the functional architecture possesses knowledge about system functions and their interconnections. Based on that knowledge, the system architect represents the functions and their interconnections through models in a model-based systems engineering tool, such as AutoFOCUS3 [49]. These MBSE tools use metadata items to symbolically represent architecture artifacts, such as system functions and communication channels. *Reasoning* denotes the manipulation of represented knowledge to produce representations of new ones [48]. For example, consider a system architecture with functions A, B, and C, along with a communication channel from function A to function B and another communication channel from function B to function C. Based on this representation, one can reason that there exists an indirect flow of information from function A to function C.

Answer Set Programming (ASP) [50] is a declarative language for KRR. ASP solvers like DLV [7] and clingo [51] enable the knowledge representation through facts and knowledge reasoning through rules. Facts may be represented by predicates holding a number of arguments. We illustrate below how the above functional architecture can be represented in DLV. This representation contains two predicates, namely `function` and `channel`. The predicate `function` holds one argument representing the function name, and `channel` holds two arguments representing a channel from one function to another. The representation of those facts expresses that the following three functions and two channels are true.

```
% Representation of system functions
function(a).
function(b).
function(c).

% Representation of communication channels
channel(a,b).
channel(b,c).
```

A DLV rule may contain the form  $a_0 \leftarrow a_1, \dots, a_n$ , where the fact  $a_0$  is the head of the rule, and the facts  $a_1, \dots, a_n$  are the body of the rule. A DLV rule can be used to derive new facts based on the existing ones. The head of a rule is true if all facts represented in the body of the rule are true. We illustrate below one DLV rule to derive `indirect_information_flow` from the existing facts. For the sake of simplicity, the rule only derives indirect information flows from two subsequent communication channels.

```
% Reasoning for indirect information flow based on represented functions and channels
indirect_information_flow(X,Z) :-
    function(X), function(Y), function(Z),
    channel(X,Y), channel(Y,Z).
```

KRR methods have played a significant role in automating various technologies, including the semantic web [52, 53]. The semantic web employs metadata representation to facilitate automated interpretation, enabling personalized search and data integration. This thesis proposes lightweight semantics to define the meaning and interpretation of artifacts computed by safety and security co-analysis activities. KRR solvers, such as DLV, provide languages to specify semantics within a given domain. By specifying semantics in KRR solvers, one can establish a formal and structured representation of knowledge, enabling machines to reason, query, and make inferences based on the defined semantics. This thesis proposes (see a brief overview in Sections 2.1 through 2.4) the use of KRR to represent key artifacts computed by or relevant to safety and security co-analysis activities and reason about the safety and security of the system architecture.

**Formal Verification (FV).** *Formal methods* [54] are techniques to mathematically specify the desired behavior and properties of complex systems, such as safety-critical systems. The formal specification of the chosen system and its properties are expressed in a formal language with a well-defined syntax and semantics to enable precise and unambiguous specifications. *Formal verification* (FV) is a specific application of formal methods focusing on automated analysis to search through the execution paths of the specified system to identify violations of the system properties.

There are several approaches for formal verification, including symbolic execution, abstract interpretation, and model checking [55]. This thesis focuses on both symbolic execution and model checking. *Symbolic execution* analyzes the behavior of a specified system through symbolic expressions (instead of actual values) representing the system behavior. Symbolic execution represents system inputs and outputs as symbolic expressions to generate a set of execution paths describing the system behavior based on the system inputs. *Model checking* specifies a finite-state model of the system and explore all possible states of the model to verify violations of a given system property.

Maude [56] is a rewriting logic specification language that may be used for formal verification. Maude supports symbolic execution through its model checking capabilities. To enable symbolic execution, the user of Maude specifies the system behavior following Maude specifications, such as rewrite rules:  $\tau \longrightarrow \tau'$  if `cond`, which may be interpreted as a transition from state  $\tau$  to state  $\tau'$  if all conditions in  $\tau$  are true. Maude provides a `search` command that may be used to verify system properties. The `search` command takes as input both an initial state and a search pattern, and uses breath-first search for searching for states (specified by rewrite rules) that match with the search pattern [57]. For example, the user may define an initial ‘safe’ state for the system and use the `search` command for searching for states that violate a safety property. In this thesis, Maude is used for conducting safety and security verification of an automotive system, specifically the Cooperative Adaptive Cruise Control (CACC) system. Further elaboration on this topic can be found in Section 2.5, where we also investigate the connection between the introduced lightweight semantics and formal verification.

**Roadmap.** As motivated in Chapter 1, it is crucial to introduce semantics into key artifacts to enable the automation of safety and security co-analysis activities. The challenge lies in identifying the key artifacts that necessitate the introduction of lightweight semantics. Our proposition centers on the adoption of lightweight semantics to key artifacts computed throughout the design of safety-critical systems. That is, this thesis proposes the introduction of lightweight semantics to system architecture artifacts, safety and security artifacts, and safety and security architecture patterns. By introducing lightweight semantics into these artifacts, we have successfully enabled the automation of several safety and security co-analysis activities (see articles 1–7 in Figure 2.1). Our proposal focuses on the adoption of lightweight semantics that leverage metadata associated with such artifacts. The *metadata* describes the knowledge about a specific artifact. To enhance the reader’s understanding of the proposed lightweight semantics, we now provide examples illustrating (i) the metadata of system architecture artifacts (Section 2.1), (ii) safety and security artifacts (Section 2.2), and (iii) safety and security architecture patterns (Section 2.3). We provide by example how such metadata may be represented as facts in a KRR solver. Additionally, Section 2.4 provides a high-level explanation on how reasoning rules may be specified to enable the automation of safety and security activities and their interactions based on the represented metadata. Section 2.5 describes how we connect lightweight semantics with formal verification. Section 2.6 discusses gaps and corresponding future directions based on the gaps and results presented in the thesis.

## 2.1 Lightweight Semantics for Architecture Artifacts

System architectures designed following a MBSE approach consists of several viewpoints, including requirements, logical architecture, software architecture, and hardware architecture. Viewpoints are designed using a specific language (e.g., SysML) encompassing the syntax and semantics of the viewpoint. These viewpoints are commonly stored as metadata within an MBSE tool. For example, MBSE tools that utilize SysML often store metadata in XML format. The metadata associated with a viewpoint provides comprehensive information about each artifact (a.k.a. architecture element) within the viewpoint.

We illustrate a subset of the metadata items from a logical architecture. The upper part of Figure 2.2 illustrates the metadata from a logical architecture. The logical architecture contains seven artifacts, i.e., four components and three communication channels. The metadata items of a component contain the unique identification (ID for short), name, input port and output port, and component type (i.e., public, gateway or private). A *public component* denotes a component outside of the system boundary that may be accessed by external users. A *gateway component* denotes a component at the border of the system boundary. A *private component* denotes a component within the system boundary. The metadata items of a communication channel contain the ID, name, input port and output port. The bottom part of Figure 2.2 illustrates the allocations of the four components designed in the logical architecture. An *allocation* denotes whether a component is implemented as software or hardware unit. The metadata items of an allocation contain

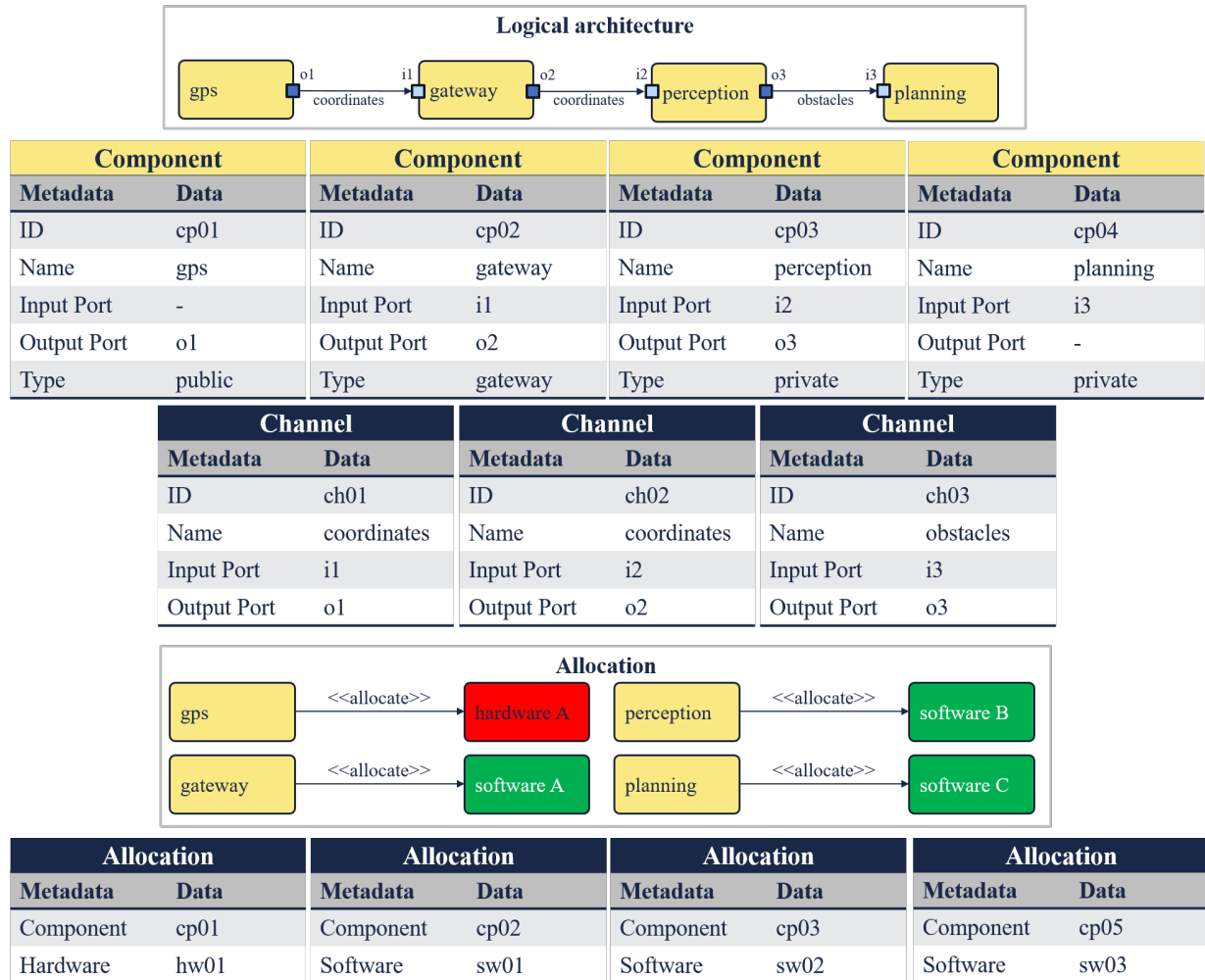


Figure 2.2: Example of metadata from logical architecture and allocations.

the component ID, and the ID of a hardware or a software unit. The assignment of the component type is expected to take place at the hardware architecture, where the system boundary and entry points (i.e., public components) are defined. This information may be cross-referenced with other architecture views (i.e., traceability) using the allocation table. The software architecture and hardware architecture are omitted in Figure 2.2.

Based on the metadata of system architecture artifacts, we introduce lightweight semantics for different viewpoints of a system architecture. We use knowledge representation for construction of a Domain-Specific Language (DSL) for representing system architecture artifacts. This representation shall enable the reasoning of the system architecture from a safety and security perspective, as demonstrated in our article [45] for pattern instantiation and in our article [42] for attack path enumeration. Our DSL has been initially introduced in our article [39] and has since been refined through subsequent articles, including [45]. The representation of the system architecture may be manually encoded in our DSL or



automatically obtained from an MBSE tool, as exemplified in our articles [40, 41]. The metadata of the logical architecture, as illustrated in Figure 2.2, can be represented in our Domain-Specific Language as follows.

```
% Representation of components - 'not_specified' means the data was not specified
component_input(cp01,gps,not_specified).
component_output(cp01,gps,o1).
component_input(cp02,gateway,i1).
component_output(cp02,gateway,o2).
component_input(cp03,perception,i2).
component_output(cp03,perception,o3).
component_input(cp04,planning,i3).
component_output(cp04,planning,not_specified).

% Representation of component type
public(cp01).
gateway(cp02).
private(cp03).
private(cp04).

% Representation of communication channels
channel(ch01,coordinates,o1,i1).
channel(ch02,obstacles,o2,i2).
channel(ch03,obstacles,o3,i3).

% Representation of allocations
allocation(cp01,hw01).
allocation(cp02,sw01).
allocation(cp03,sw02).
allocation(cp04,sw03).
```

The above representation provides lightweight semantics for a logical architecture and allocation table. The vocabulary consists of the terms `component_input`, `component_output`, `public`, `private`, `channel` and `allocation`, along with its associated semantics, such as `component_output(cp03,perception,o3)` denoting the component `cp03` (named as `perception`) providing output data through output port `o3`.

## 2.2 Lightweight Semantics for Safety and Security Artifacts

The results of a safety analysis (a.k.a. safety artifacts) using methods like FTA and STPA can be described on spreadsheet programs or special tools, such as Arbore-Analyst [58] for FTA and XSTAMP [59] for STPA. Similarly, the results of a security analysis (a.k.a. security artifacts) using TARA are often described on spreadsheet programs, such as excel. Some OEMs or suppliers may have their own tools to describe safety and security artifacts.

The metadata of safety and security artifacts may be extracted from either a spreadsheet

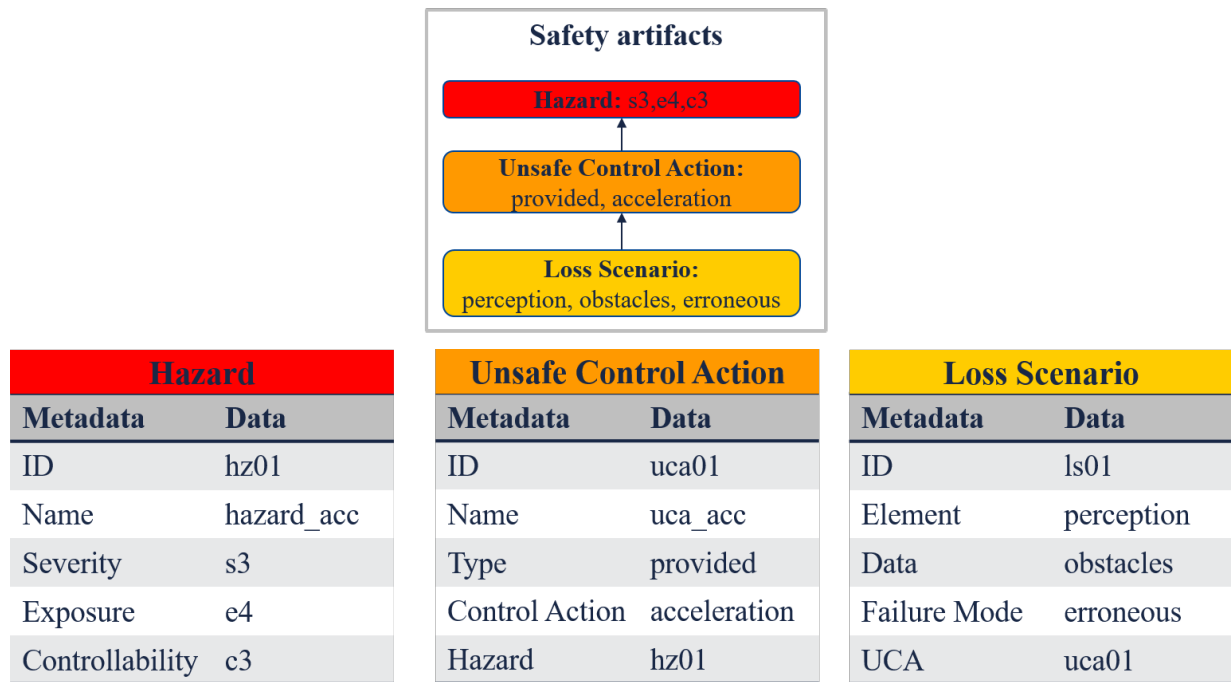


Figure 2.3: Example of metadata from safety artifacts (HARA + STPA).

program or a special tool. The extraction is possible (even manually from a spreadsheet program) because there are well-known terms associated with safety and security artifacts.

Avizienis et al [12] propose a taxonomy on dependability, where several safety-relevant terms are defined. For example, a fault has several attributes, including the fault dimension which defines whether the fault arises from a software or hardware error. Similarly, a failure resulting from a fault contains several attributes, including the failure mode, which defines how the failure manifests in the system. Examples of failure modes are *erroneous* (a.k.a. content failure or malfunction), denoting incorrect computation of output values by a function, and *loss* (a.k.a. halt failure), indicating that a function is no longer operating.

Security-wise, an asset has one or more security properties that shall be satisfied. One can use the well-known CIA (Confidentiality, Integrity and Availability) triad model or STRIDE to define the security properties of an asset. STRIDE, in particular, provides a mapping from a security property to a threat, such as the security property availability may be violated by denial of services attacks. ISO/SAE 21434 provides definitions and examples of security artifacts, such as attack paths, which is a sequence of actions to realize a threat scenario. These actions are associated with architecture elements that an attacker may need to compromise to realize a threat scenario. In summary, we gather relevant metadata from safety and security artifacts based on well-known terms used by researchers and practitioners when performing safety and security analysis.

Figure 2.3 illustrates the metadata from safety artifacts computed by HARA and STPA. The upper part of Figure 2.3 illustrates a hazard, an unsafe control action, and a loss scenario. The bottom part of Figure 2.3 describes the metadata of such safety artifacts.

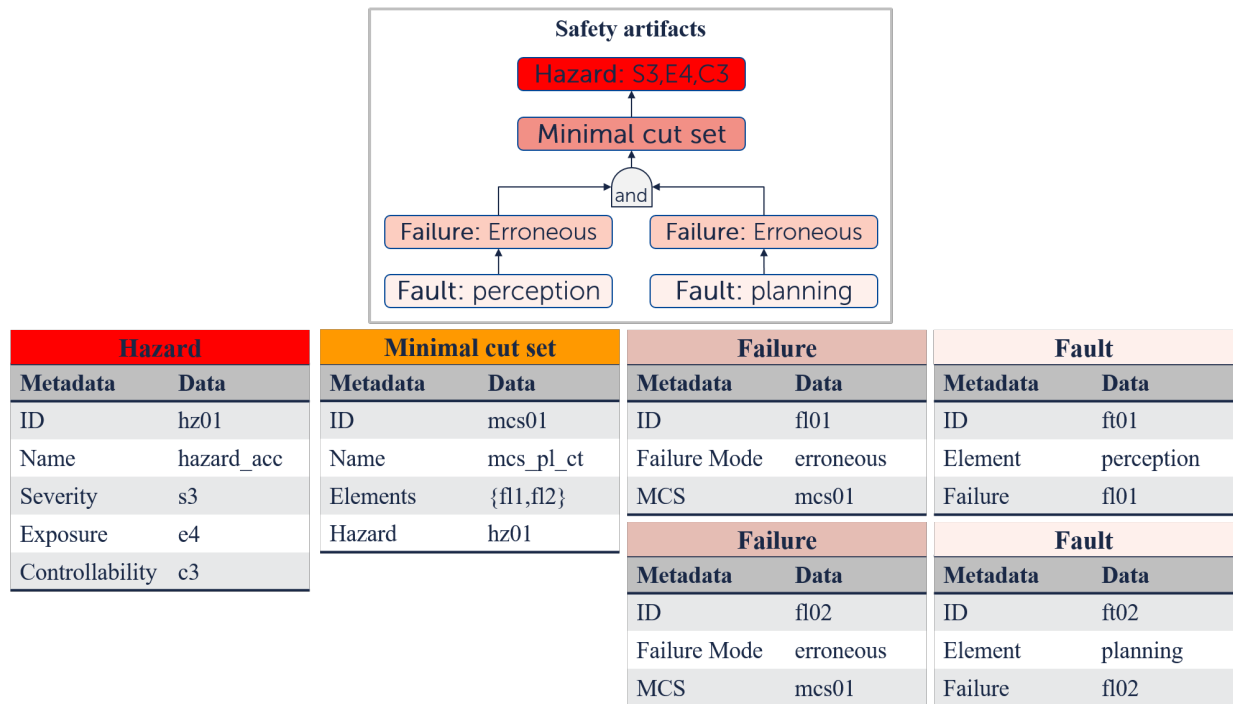


Figure 2.4: Example of metadata from safety artifacts (HARA + FTA).

The metadata items of a hazard contain an ID, name, severity, exposure, and controllability. The metadata items of an unsafe control action contain an ID, name, type, control action, and the hazard caused by the unsafe control action. Finally, the metadata items of a loss scenario contain an ID, element (e.g., a component), data (computed by the element), failure mode, and the unsafe control action caused by the loss scenario.

Figure 2.4 illustrates the metadata from safety artifacts computed by HARA and a basic FTA. The upper part of Figure 2.4 illustrates a hazard, a minimal cut set, two failures, and two faults. The faults in perception and planning trigger erroneous failures which are part of the minimal cut set. If both of these failures occur (at the same time or in sequence), the minimal cut set culminates in the manifestation of the hazard. The bottom part of Figure 2.4 describes the metadata of such safety artifacts (similar to the above explanation).

Figure 2.5 illustrates the metadata from security artifacts computed by TARA. The upper part of Figure 2.5 illustrates the security artifacts computed by the first six activities of TARA, i.e., from asset identification to risk determination. The bottom part of Figure 2.5 describes the metadata of such security artifacts. The metadata items of an asset contain an ID, an element (e.g., perception) that computes some data (e.g., obstacles), and the relevant security property (e.g., integrity) of the asset. The damage scenario for this asset includes an ID, and name (a.k.a. description). The impact rating includes an ID, and the safety impact for a damage scenario. Notice, however, that the impact rating activity may also compute information on financial, operational, and privacy impact. The threat scenario includes an ID, and the threat (e.g., tampering) that may violate the security

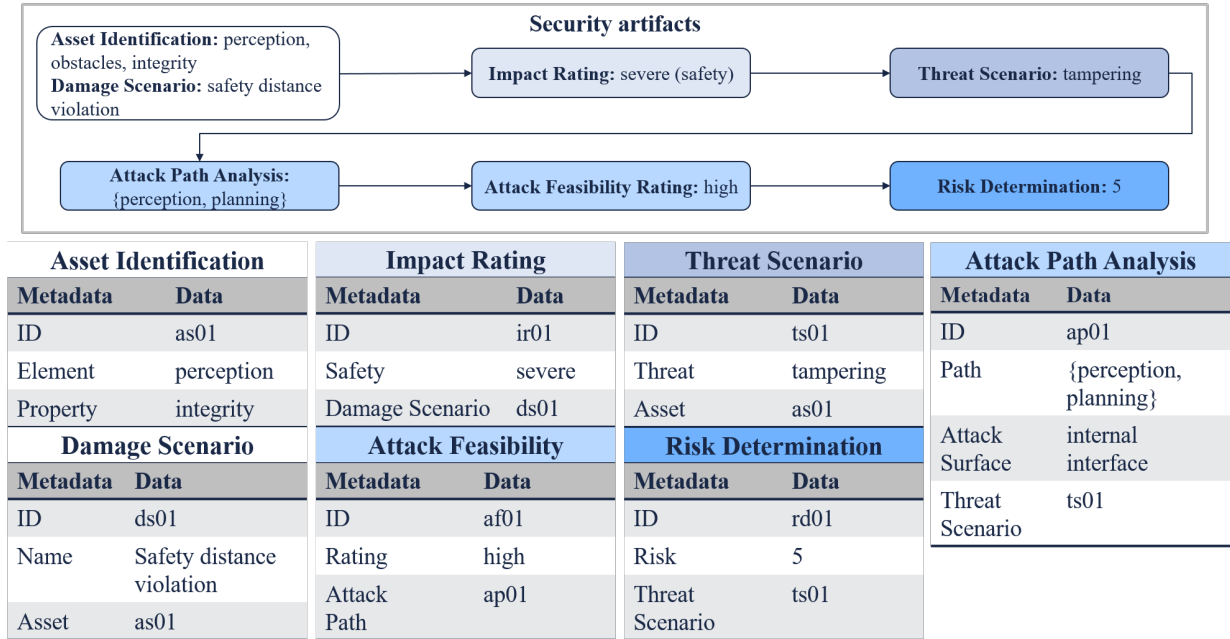


Figure 2.5: Example of metadata from security artifacts (TARA).

property of an asset. The attack path contains an ID, the path itself (e.g., perception  $\rightarrow$  planning), the attack surface, and the threat scenario associated with the attack path, which one can read as a path exploited by an intruder to carry out tampering attacks. The *attack surface* [43] is an abstract term that describes whether an attack path includes public elements or not. It provides a distinction between internal interfaces, which correspond to attack paths originating from within the system boundary without involving public elements, and external interfaces, which correspond to attack paths that do involve public elements. An external interface can originate from outside the system boundary (e.g., the GPS in Figure 2.2)) or originate from within the system boundary but terminates at a public element. The attack feasibility includes an ID, and the rating for an attack path. Finally, the risk determination includes an ID and the risk value for a threat scenario.

We introduce lightweight semantics to represent safety and security artifacts based on their metadata. To this end, we use knowledge representation for construction of a Domain-Specific Language (DSL) for representing safety and security artifacts. This representation enables reasoning capabilities for safety and security artifacts, including the mapping of safety to security artifacts, as demonstrated in our articles [45, 42]. In our article [45], we introduce the DSL for representing safety artifacts computed by HARA and FTA. Our article [42] extends the DSL to specify safety artifacts computed by HARA and STPA. Our DSL for representing security artifacts computed by TARA has been introduced by our article [45]. The representation of safety and security artifacts may be manually encoded in our DSL or automatically obtained a MBSE tool, as demonstrated by our articles [40, 45]. As an example, we represent the metadata of the safety artifacts illustrated in Figure 2.2 in our DSL as follows. For the sake of presentation, we use the name of the element (i.e.,

perception) and data (i.e., obstacles) instead of using their IDs.

```
% Representation of hazards
hazard(hz01,hazard_acc,s3,e4,c3).

% Representation of unsafe control actions
unsafe_control_action(uca01,uca_acc,provided,acceleration).

% Representation of loss scenarios
loss_scenario(ls01,perception,obstacles,erroneous).

% Representation of relationships between safety artifacts
loss_scenario2uca(ls01,uca01).
uca2hazard(uca01,hz01).
```

The above representation provides lightweight semantics for safety artifacts computed by HARA and STPA. The vocabulary consists of the terms `hazard`, `unsafe_control_action`, `loss_scenario`, `loss_scenario2uca` and `uca2hazard`, along with its associated semantics, such as `loss_scenario2uca(ls01,uca01)` denoting that the loss scenario `ls01` leads to unsafe control action `uca01`.

## 2.3 Lightweight Semantics for Safety and Security Architecture Patterns

Safety and security architecture patterns are described in textual and graphical form. Researchers and practitioners often describe architecture patterns using templates, such as the pattern templates used in [13] and [24]. Our goal is to reason about safety and security architecture patterns to enable their recommendations in an automated fashion. To this end, the initial step involves extracting relevant metadata from safety and security architecture patterns to enable their declarative representations.

In contrast to safety and security artifacts, there is a lack of consensus within the scientific and industrial communities regarding the terminology used in relation to safety and security architecture patterns, even for safety architecture patterns which have a longer history compared to security architecture patterns. The question now is “what are the relevant metadata associated with safety and security architecture patterns to enable their automation?” Answering this question is no simple task due to the descriptive nature of architecture patterns, which primarily rely on textual and graphical representations.

To answer this question, we have conducted a literature review on safety and security architecture patterns. Chapter 3 provides a summary of this literature review. Through our literature review, we found that examining the instantiation of pattern templates can provide valuable insights into the relevant metadata associated with architecture patterns. Firstly, the pattern structure is graphically described in the template. This description illustrates the placement of pattern components and channels within the system architecture. Consequently, we can extract relevant metadata from the instantiation of architecture patterns. Secondly, the textual description within the pattern template describes the

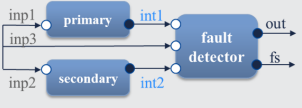
|                   |   | Pattern Instantiation     |         |                   |                 |                   |                |
|-------------------|---|---------------------------|---------|-------------------|-----------------|-------------------|----------------|
| Name              | Heterogeneous Duplex  | Metadata                  |         | Data              |                 |                   |                |
| Structure         |    | ID                        | htd01   |                   |                 |                   |                |
| Intent            | This pattern provides fault tolerance. This pattern can tolerate a single fault in the primary or secondary fault.  | Name Heterogeneous Duplex |         |                   |                 |                   |                |
| Problem addressed | This pattern is suitable for addressing life-critical hazards (ASIL D). This pattern tolerates hardware and software faults. This pattern tolerates faults by addressing failure mode: erroneous or loss. | Pattern Component         |         | Pattern Component |                 | Pattern Component |                |
| Requirements      | Primary and secondary components shall be implemented with different designs. ...   | Metadata                  |         | Data              |                 | Metadata          |                |
|                   |   | ID                        | cp01    | ID                | cp02            | ID                | cp03           |
|                   |   | Name                      | primary | Name              | secondary       | Name              | fault detector |
|                   |   | Input Port                | i1      | Input Port        | i2              | Input Port        | i3, i4, i5     |
|                   |   | Output Port               | o1      | Output Port       | o2              | Output Port       | o3, o4         |
|                   |   | Pattern Channel           |         |                   | Pattern Channel |                   |                |
|                   |   | Metadata                  |         | Data              |                 | Metadata          |                |
|                   |   | ID                        | ch01    | ... (omitted)     |                 | ID                | ch04           |
|                   |   | Name                      | inp1    |                   |                 | Name              | int1           |
|                   |   | Input Port                | n/s     |                   |                 | Input Port        | o1             |
|                   |   | Output Port               | i1      |                   |                 | Output Port       | i3             |

Figure 2.6: Metadata (instantiation) from the Heterogeneous Duplex pattern.

|                   |   | Pattern Attributes   |  |                    |
|-------------------|---|----------------------|--|--------------------|
| Name              | Heterogeneous Duplex  | Metadata             |  | Data               |
| Intent            | This pattern provides fault tolerance. This pattern can tolerate a single fault in the primary or secondary fault.  | Name                 | Heterogeneous Duplex                         |                    |
| Problem addressed | This pattern is suitable for addressing life-critical hazards (ASIL D). This pattern tolerates hardware and software faults. This pattern tolerates faults by addressing failure mode: erroneous or loss. | Intent               | fault tolerance                              |                    |
| Requirements      | Primary and secondary components shall be implemented with different designs. ...   | Problem addressed    | ASIL Class                                   | d                  |
|                   |   |                      | Fault Dimension                              | hardware; software |
|                   |   |                      | Failure Mode                                 | erroneous; loss    |
|                   |   | Pattern Requirements |  |                    |
|                   |   | Metadata             |  | Data               |
|                   |   | ID                   | rq01   |                    |
|                   |   | Name                 | shall be implemented with different designs. |                    |
|                   |   | Pattern              | Heterogeneous Duplex                         |                    |
|                   |   | Component            | primary; secondary                           |                    |

Figure 2.7: Metadata (attributes and requirements) from the Heterogeneous Duplex pattern.

pattern's intent and the specific problem it addresses. This description represents what guarantees the pattern provides, and the safety or security problem addressed by the pattern. As a result, we can extract relevant metadata on why and when to use architecture patterns. Thirdly, the pattern template may include requirement descriptions associated with the pattern, such as functional, software or hardware requirements. These descriptions outline how the pattern shall be implemented to address safety or security problems. We can extract relevant metadata from such requirements descriptions.

Figure 2.6 illustrates the metadata items related to the instantiation of a safety architecture pattern. The left side of Figure 2.6 illustrates the instantiation of a pattern template for the Heterogeneous Duplex pattern [13]. This pattern consists of three components, namely primary, secondary (redundancy), and fault detector. The right side of Figure 2.6 illustrates relevant metadata extracted from the first two rows (i.e., name and structure) of the pattern

| Description       |   | Pattern Instantiation |               |                        |                |                   |          |
|-------------------|---|-----------------------|---------------|------------------------|----------------|-------------------|----------|
| Name              | Message Authentication  | Metadata              |               | Data                   |                |                   |          |
| Structure         |   | ID                    |               | ma01                   |                |                   |          |
| Intent            | This pattern ensures the integrity and authenticity of messages computed by the origin.   | Name                  |               | Message Authentication |                |                   |          |
| Problem addressed | This pattern prevents that messages computed by the origin are tampered or spoofed. The messages computed by the origin are signed with its private key. The receiver verifies the received message with the origin's public key. This pattern is suitable for protecting messages exchanged between internal components. | Pattern Component     |               | Pattern Component      |                | Pattern Component |          |
| Requirements      | Origin shall implement a signature generator to support message signing.<br>...   | Metadata              |               | Data                   |                | Metadata          |          |
|                   |   | ID                    | cp01          | ID                     | cp02           | ID                | cp03     |
|                   |   | Name                  | origin        | Name                   | sig. generator | Name              | receiver |
|                   |   | Input Port            | i1            | Input Port             | i2             | Input Port        | i3, i4   |
|                   |   | Output Port           | o1,o2         | Output Port            | o3             | Output Port       | o4       |
|                   |   | Pattern Channel       |               | ...                    |                | Pattern Channel   |          |
|                   |   | Metadata              |               | Data                   |                | Metadata          |          |
|                   |   | ID                    | cp04          | ...                    |                | ID                | ch05     |
|                   |   | Name                  | sig. verifier | ...                    |                | Name              | int5     |
|                   |   | Input Port            | i5            | ...                    |                | Input Port        | o5       |
|                   |   | Output Port           | o5            | ...                    |                | Output Port       | i4       |

Figure 2.8: Metadata (instantiation) from the M. Authentication pattern.

template. The pattern instantiation contains an ID and a pattern name (assumed to be unique as well). The metadata related to pattern components and channels are extracted from the second row of the pattern template (left side of Figure 2.6). Notice that some metadata related to pattern channels are not completed, e.g., the input port of channel **ch01** is not specified (n/s). This is because only a snippet of the system architecture is illustrated in the pattern template.

Figure 2.7 illustrates the metadata items related to the intent, problem address, and requirements of the Heterogeneous Duplex pattern. This pattern provides fault tolerance by providing a redundant component, namely secondary. This pattern may be deployed to address (i) life-critical hazards [13], such as ASIL D, (ii) software and hardware faults (e.g., by ensuring software and hardware independence), and (iii) failure mode erroneous and loss (e.g., by implementing plausibility checks). The right side of Figure 2.7 illustrates the relevant metadata extracted from the pattern intent and problem addressed by the pattern. The bottom of the pattern template describes by example a single software requirement of the Heterogeneous Duplex pattern. The metadata of this requirement is illustrated on the right side of Figure 2.7. The metadata includes the requirement description and the components that shall implement the requirement (i.e., primary and secondary).

Figure 2.8 illustrates the metadata items related to the instantiation of a security architecture pattern. The left side of Figure 2.8 illustrates the instantiation of a pattern template for the Message Authentication pattern (a.k.a. Verifiable Transmission pattern) [25]. This pattern consists of four components, namely origin, signature generator, receiver, and signature verifier. The metadata extracted from security architecture patterns is similar to the metadata described above for safety architecture patterns.

Figure 2.9 illustrates the metadata items related to the intent, problem address, and requirements of the Message Authentication pattern. This pattern satisfies the integrity and authenticity of messages computed by the origin. The Message Authentication pattern may

| Description       |   | Pattern Attributes          |   |                     |
|-------------------|---|-----------------------------|---|---------------------|
| Name              | Message Authentication  | <b>Metadata</b>             | <b>Data</b>   |                     |
| Intent            | This pattern ensures the integrity and authenticity of messages computed by the origin.   | Name                        | Message Authentication  |                     |
| Problem addressed | This pattern prevents that messages computed by the origin are tampered or spoofed. The messages computed by the origin are signed with its private key. The receiver verifies the received message with the origin's public key. This pattern is suitable for protecting messages exchanged between internal components. | Intent                      | integrity; authenticity   |                     |
| Requirements      | Origin shall implement a signature generator to support message signing.<br>...   | Problem addressed           | Threat  | tampering; spoofing |
|                   |   |                             | Attack Surface  | internal interface  |
|                   |   | <b>Pattern Requirements</b> |   |                     |
|                   |   | <b>Metadata</b>             | <b>Data</b>   |                     |
|                   |   | ID                          | rq01  |                     |
|                   |   | Name                        | shall implement a signature generator to support message signing. |                     |
|                   |   | Pattern                     | Message Authentication  |                     |
|                   |   | Component                   | origin  |                     |

Figure 2.9: Metadata (attributes and requirements) from the M. Authentication pattern.

be deployed to address tampering and spoofing attacks against messages computed by the origin. To this end, the origin has an asymmetric key pair, where the public key certificate has been shared with the receiver through a secure channel. The messages computed by the origin are signed with the origin's private key, whereas the received messages are verified by the receiver using the origin's public key. This pattern is suitable for mitigating tampering or spoofing attacks carried out within the system (internal interface). This is because it might be impractical to expect external clients (i.e., public elements) to create an asymmetric key pair and distribute the public key certificate to all relevant components. The right side of Figure 2.9 illustrates the relevant metadata extracted from the pattern intent and problem addressed by the pattern. The bottom of the pattern template describes by example a single software requirement of the Message Authentication pattern.

We introduce lightweight semantics to represent safety and security architecture artifacts based on their metadata items. We use knowledge representation for construction of a Domain-Specific Language (DSL) for representing safety and architecture patterns. This representation enables the reasoning of safety and security architecture patterns, including pattern recommendation, as demonstrated several of our articles, such as [45]. Our articles [39] and [45] introduce the DSL for the representation of safety and security architecture patterns, respectively. For example, one can represent the metadata extracted from the Heterogeneous Duplex pattern in our DSL as follows. For the sake of presentation, we use the name of components and channels instead of using their IDs.

```
% Representation of safety architecture patterns (instantiation)
safety_pattern(htd01,
  heterogeneous_duplex,
  elem(primary,secondary,fault_detector),
  input(inp1,inp2,inp3),
  internal(int1,int2),
  output(out,fs)).
```

```
% Representation of safety architecture patterns (attributes)
```



```

safety_pattern_attributes(heterogeneous_duplex,
    type(fault_tolerant),
    asil(d),
    fault(hardware,software),
    failure(erroneous,loss)).

% Representation of safety architecture patterns (requirements)
requirement(rq01,
    shall_be_implemented_with_different_designs,
    elem(primary,secondary)).

```

The above representation provides lightweight semantics for safety architecture patterns and requirements. The vocabulary consists of the terms `safety_pattern`, `safety_pattern_attributes`, and `requirement`, along with its associated semantics, such as `requirement(rq01, ...)` denoting requirement `rq01` is relevant to architecture elements `primary` and `secondary` and that such elements shall be implemented with different designs.

## 2.4 Safety and Security Reasoning

### 2.4.1 Safety Reasoning

The first question of this thesis (**RQ1**) asks whether the introduction of lightweight semantics enables the automated recommendation of safety architecture patterns. To answer this question, we use KRR and propose safety reasoning rules to derive safety architecture patterns from the existing knowledge represented in our DSL. The proposed DSL enables the representation of both the system architecture often designed by a system architect, and the safety artifacts obtained from a safety analysis performed by a safety engineer. These representations, and the representation of safety architecture patterns, are the basis to enable automated reasoning for safety architecture patterns.

Our safety reasoning rules infer when a safety architecture pattern can be recommended based on the safety artifacts, and instantiate the recommended pattern based on the system architecture. In our articles, such as [45], the safety reasoning rules for pattern recommendation are expressed as disjunction rules of the following form

$$a_0 \vee \text{not\_}a_0 \leftarrow a_1, \dots, a_n$$

where  $a_0$  denotes when a pattern is recommended and  $\text{not\_}a_0$  denotes when a pattern is not recommended. The recommendation decision is based on the existing knowledge  $a_1, \dots, a_n$ . Specifically, the relevant knowledge required to make this decision consists of the safety artifacts and the attributes of safety architecture patterns. The safety artifacts are hazards (identified by a HARA analysis) and the potential causes for the hazards identified by either a FTA or STPA analysis. The attributes of a safety architecture pattern denote the intent and problem addressed by the pattern. The intent of a safety architecture

pattern denotes whether the pattern provides fault tolerance or fault detection. The user may choose the desired intent of the pattern by specifying an additional requirement. The problem addressed by the pattern has a direct link with the represented safety artifacts.

Our safety reasoning rules infer that a safety architecture pattern is recommended if the following conditions hold:

$$\begin{aligned} \text{asil\_class} &\in \text{safety\_pattern\_attribute} \\ \text{fault\_dimension} &\in \text{safety\_pattern\_attribute} \\ \text{failure\_mode} &\in \text{safety\_pattern\_attribute} \\ \text{pattern\_type} &\in \text{safety\_pattern\_attribute} \end{aligned}$$

The `asil_class` can be calculated based on represented severity, exposure and controllability of a hazard. The `fault_dimension` can be obtained based on faults identified by FTA or derived from loss scenarios, i.e., check whether the element associated with a loss scenario is implemented as software or hardware. The `failure_mode` can be obtained from failures identified by FTA or loss scenarios identified by STPA. The `pattern_type` can be obtained from an additional requirement that specifies the intent of the pattern.

Our articles, such as [41, 40], propose safety reasoning rules to automate the recommendation of safety architecture patterns based on the aforementioned conditions. It is important to note that a safety architecture pattern is recommended if the aforementioned conditions are met. This may result in scalability issues as several solutions may be computed. For example, consider the safety artifacts computed by HARA and FTA depicted in Figure 2.4: Two safety architecture patterns, one for perception and one for planning, are recommended if the aforementioned conditions are met. However, this dual-pattern solution is not required because the minimal cut set only culminates in the manifestation of the hazard if failures from both perception and planning occur. Consequently, having just one safety architecture pattern, either for perception or planning, suffices to address the hazard’s manifestation. To address scalability issues, our article [45] provides solutions to deal with scalability by using DLV constraints. Once a safety architecture pattern is recommended, it may be instantiated based on the system architecture. Communication channels may be derived from the system architecture to instantiate pattern channels (i.e., input, internal, and output channels). For example, from the Heterogeneous Duplex pattern (see Figure 2.6), one can derive the channels for the secondary (a.k.a. redundant) component based on the existing channels associated with the primary component. Our articles [40, 41] discuss in more detail how to instantiate safety architecture patterns after their recommendations. For each recommended and instantiated safety architecture pattern, we specify reasoning rules to compute the pattern requirements. These requirements shall be realized during the system development process to ensure the intended functionality of the pattern. Our article [40] discusses in more detail about the pattern requirements computed by our reasoning rules. In addition, our articles [40, 41] provide specific criteria to assist users in their selection of an architecture solution based on the recommended safety architecture patterns.

| Safety Artifact   |   | Security Artifact   |
|---|---|---|
| Element/Data of loss scenarios<br><b>Example:</b> <i>obstacles computed by perception</i>         | → | Assets<br><i>obstacles computed by perception</i>           |
| Description of hazards<br><b>Example:</b> <i>safety distance violation</i>                        | → | Damage scenarios<br><i>safety distance violation</i>        |
| Controllability and Severity of hazards<br><b>Example:</b> <i>C3, S3</i>                          | → | Impact rating (safety)<br><i>severe</i>                     |
| Failure mode of loss scenarios<br><b>Example:</b> <i>erroneous</i><br><b>Example:</b> <i>loss</i> | → | Threat scenarios<br><i>integrity</i><br><i>availability</i> |

Table 2.1: Mapping from safety artifacts (HARA + STPA) to security artifacts.

In summary, our safety reasoning rules enable (i) the recommendation of safety architecture patterns, (ii) the instantiation of recommended safety architecture patterns, and (iii) the computation of pattern requirements for instantiated safety architecture patterns.

## 2.4.2 Security Reasoning

The second research question of this thesis (**RQ2**) asks whether the introduction of lightweight semantics to security artifacts and security architecture patterns enable the automated computation of TARA activities. To answer this question, we use KRR and propose security reasoning rules to derive security artifacts from the existing knowledge represented in our DSL. The representation of both safety and security artifacts and security architecture patterns in our DSL are the basis to enable automated reasoning for TARA activities. Additionally, the representation of system architecture artifacts plays a crucial role in facilitating the automation of one TARA activity, namely attack path analysis.

Section 2.2 described the representation of security artifacts computed by TARA activities. These security artifacts may be represented in our DSL as input information provided by users. This thesis, however, investigates how much of this information can be automated and how. To this end, we have conducted a literature review on approaches on how to derive security artifacts from safety artifacts. Chapter 3 provides a summary of this literature review. Based on the outcome of this literature review, we have identified specific security artifacts that may be automated based on existing safety artifacts, namely (i) assets, (ii) damage scenarios, (iii) impact rating, and (iv) threat scenarios. Table 2.1 provides a high-level illustration of the mapping from safety artifacts (computed by HARA and STPA) to security artifacts. Our article [42] provides a detailed explanation about this mapping, which can be automated through the specification of reasoning rules. This automated mapping answers the third research question of this thesis (**RQ3**), paving the way for the automated identification of synergies between safety and security analysis. This

is a synergy generated by combining safety and security analysis to enhance the efficiency and safety-awareness of the security analysis process. Consequently, safety artifacts such as hazards are analyzed and addressed also from a security perspective.

Security reasoning rules may be specified to enumerate attack paths based on the represented system architecture and assets. An attack path consists of a sequence of architecture elements where the first element is the entry point of the intruder and the last element (i.e., asset) is the target of the intruder. For example, an attack path where the first element is a public component and the last element is an asset. Security reasoning rules may infer such a sequence of architecture elements based on, e.g., the communication channels between components. Enumerating attack paths solely on the system architecture and assets may be naive and lead to unrealistic attack paths. An intruder model is essential to decide on the sequence of architecture elements of an attack path, e.g., to decide on realistic entry points, and how the attack propagates within the architecture until reaching the target asset. We have conducted a literature review on several attacks against automated driving systems. Our literature focuses on automated driving systems, particularly systems implementing Service-Oriented Architectures (SOA). Chapter 3 provides a summary of this literature review. We formalize an intruder model tailored to SOA based on the outcome of the literature review. The intruder model formalizes the intruder capabilities in performing attacks from outside the system boundary and from within the system boundary. We have specified security reasoning rules to capture the formalized intruder model and automate the enumeration of attack paths. Our intruder model, including the formalization and implementation of security reasoning rules, is presented by our article [42]. The attack paths enumerated by our security reasoning rules cover several attack paths reported in the literature and attack paths not yet reported in the literature.

The subsequent TARA activity, known as attack feasibility rating, involves determining the rating of an attack path. The rating is assigned in ascending order, ranging from very low to high (easier to be exploited). Expertise from a security engineer is required for this activity, as they evaluate the feasibility of the attack path from the perspective of an intruder. To this end, a security engineer evaluates core factors, such as elapsed time, expertise, equipment, and knowledge of the item or component, to determine the attack feasibility rating of an attack path. Those core factors are part of a method called attack potential [23]. The automation of this activity is only possible if the rating of such core factors are represented as input information, as discussed by our whitepaper [60]. The risk value for a threat scenario is determined from the impact rating and attack feasibility rating. Security reasoning principles may be specified to automate the risk determination activity if both impact rating and attack feasibility rating have been represented in our DSL. For example, our whitepaper [60] uses the risk matrix (suggested by the ISO/SAE 21434) to automate the computation of risk values.

The last activity of TARA, namely risk treatment decision, includes the selection of security architecture patterns for addressing threat scenarios, and therefore reducing the risk value. We specify security reasoning rules to infer when a security architecture pattern can be recommended based on the represented security artifacts, and instantiate the recommended pattern based on the system architecture. The relevant security artifacts

to make a decision on whether a pattern is recommended or not are the security property of the asset represented in the damage scenario, the threat represented in the threat scenario, and the attack surface represented in the attack path. The attributes (i.e., intent and problem addressed) of a security architecture pattern are also relevant to decide on whether the pattern is recommended. Concretely, our security reasoning rules infer that a security architecture pattern is recommended if the following conditions hold:

$$\begin{aligned} \text{security\_property} &\in \text{security\_pattern\_attribute} \\ \text{threat} &\in \text{security\_pattern\_attribute} \\ \text{attack\_surface} &\in \text{security\_pattern\_attribute} \end{aligned}$$

The aforementioned conditions imply that a security architecture pattern shall be recommended to mitigate a particular threat (e.g., tampering) which occurs at the attack surface (e.g., an internal interface) targeting the security property (e.g., integrity) of an asset. Our articles, such as [45, 43], propose security reasoning rules to enable the automation of security architecture patterns based on such conditions. It is important to note that a security architecture pattern is recommended whenever the aforementioned conditions are met, which may result in scalability issues. Our article [43] provides solutions to deal with such scalability issues and usability issues by using clingo constraints. As for safety architecture patterns, we also specify reasoning rules for computing security requirements once a security architecture pattern has been recommended and instantiated into the system architecture. Those security requirements shall be realized during system development to ensure that the pattern works as intended. Our articles [43, 44] provide specific criteria to assist users in their selection of an architecture solution based on the requirements associated with the recommended security architecture patterns.

In summary, our security reasoning rules enable the automation of the following TARA activities: (i) asset identification (including damage scenarios) based on faults or loss scenarios and hazards, (ii) safety impact rating based on the controllability and severity of a hazard, (iii) threat scenarios derived from the failure mode, (iv) attack feasibility rating only if the rating of core factors from the attack potential method have been represented as input information, (v) risk determination if attack feasibility rating has been represented, and (vi) risk treatment decision, i.e., the recommendation and instantiation of security architecture patterns, including the specification of pattern requirements.

### 2.4.3 Reasoning on Safety and Security Consequences

The third research question of this thesis (**RQ3**) asks whether the introduction of lightweight semantics enables the identification of conflicts between safety and security analysis. To answer this question, we propose reasoning rules to derive safety and security consequences based on the security and safety architecture patterns represented in our DSL, respectively. Our work is inspired by [61] that extended safety architecture patterns to include security considerations, in particular security consequences that may be exploited by intruders.

The instantiation of a safety architecture pattern may cause security consequences. Each component of a safety architecture pattern is safe critical and may lead to harm if an

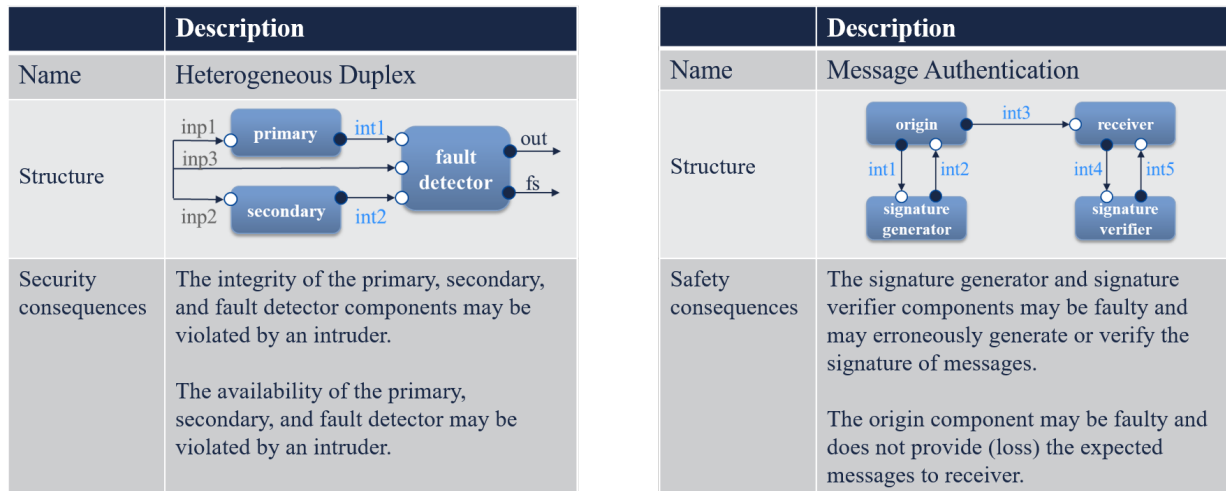


Figure 2.10: Examples of safety and security consequences caused by architecture patterns.

intruder is successful in violating its security properties. Consequently, the components of a safety architecture pattern shall be taken into account during the security analysis. The left side of Figure 2.10 describes two security consequences caused by the Heterogeneous Duplex pattern. The integrity and availability of **primary**, **secondary**, and **fault detector** shall be satisfied such that the Heterogeneous Duplex pattern can work as intended. The following security artifacts may be derived from the security consequences described in Figure 2.10: (i) each safety pattern's component becomes an asset, where its security properties are part of the damage scenario. The security property may be derived from the failure mode of the loss scenario associated with **primary**. The description of the damage scenario may be obtained from the hazard led by loss scenario associated with **primary**, and (ii) the threat scenario associated with the asset, where the threat may be derived from the asset's property using the STRIDE methodology, such as integrity  $\rightarrow$  tampering.

Similarly, the instantiation of a security architecture pattern may cause safety consequences. This thesis focuses on security for safety, therefore the instantiation of a security architecture patterns may include safety-related components. The malfunction behavior of such components may lead to hazards, and shall be taken into account during the safety analysis. The right side of Figure 2.10 describes two safety consequences caused by the Message Authentication pattern. Faults triggering erroneous failures in either **signature generator** or **signature verifier** may block safety-relevant information to be read by **receiver**, and thus preventing **receiver** from properly functioning. Similarly, faults triggering loss failures in **origin** will prevent **receiver** from even receiving any input information from **origin**. Safety artifacts may be derived from the safety consequences caused by security architecture patterns.

Our article [45] provides reasoning rules for computing safety and security consequences based on the instantiation of security and safety architecture patterns, respectively.

In summary, our reasoning rules enable the automation of safety and security consequences caused by the instantiation of security and safety architecture patterns, respectively.

Specifically, our reasoning rules compute (i) assets, damage scenarios, and threat scenarios as a direct consequence due to the instantiation of a safety architecture pattern, and (ii) loss scenarios (STPA) or faults and failures (FTA) as a direct consequence due to the instantiation of a security architecture pattern.

## 2.5 Connecting Lightweight Semantics with Formal Verification

This thesis uses formal verification to investigate how the safety property of the system may be violated by attacks, and how security mechanisms specified based on pattern requirements may be used to mitigate such attacks and ensure the safety property. Safety properties may be derived from safety goals. As an illustrative example, we consider a CACC system and the safety goal “*prevent unintended safety distance violation between vehicles in the platoon*”. In general, a safety property denotes “*nothing bad ever happens*”. Specifically, we derive a safety property from the above safety goal as “*no crash between vehicle platoons ever happens*”. We consider attacks against CACC systems and a security mechanism derived from pattern requirements to investigate whether the security mechanism is effective in mitigating such attacks. The pattern requirements may be computed in an automated fashion with the help of the introduced lightweight semantics. This investigation is crucial to answer the fourth research question of this thesis (**RQ4**), which investigates the benefits of connecting the introduced lightweight semantics with formal verification.

Our article [46] proposes a formal framework for the safety and security verification of CACC platoons. Our framework formalizes the behavior of a CACC platoon model in Maude. Our model enables the specification of vehicles (e.g., their roles, positions and speed values) and vehicle strategies for executing platooning depending on the vehicle state, such as joining and emergency states. Our framework specifies a parametric intruder model that subverts communication channels to carry out attacks. The capabilities of the intruder model are either blocking messages from a communication channel or injecting messages into communication channels. Based on the intruder model, we specify five attacks against vehicle platoons that may violate the defined safety property for the CACC platoon. These attacks consider an intruder positioned between the platoon leader and the follower members. The followers are vehicles that depend on the leader for guidance. They follow the leader’s decisions, such as adhering to a specific speed. The intruder may carry out attacks to either block messages from the leader or impersonate the leader to inject messages.

The user of our framework can utilize Maude’s `search` command to verify the safety property of the CACC platoon model in situations where an intruder has the potential to block or inject messages. The left side Figure 2.11 illustrates an initial state specified with two vehicles (a leader and a follower), the instantiation of an intruder that blocks legitimate messages from the leader and inject messages with malicious speed values. The right side of Figure 2.11 illustrates the search pattern, i.e., a crash between the follower and the leader.

When it comes to the utilization of security mechanisms, we examine two scenarios:

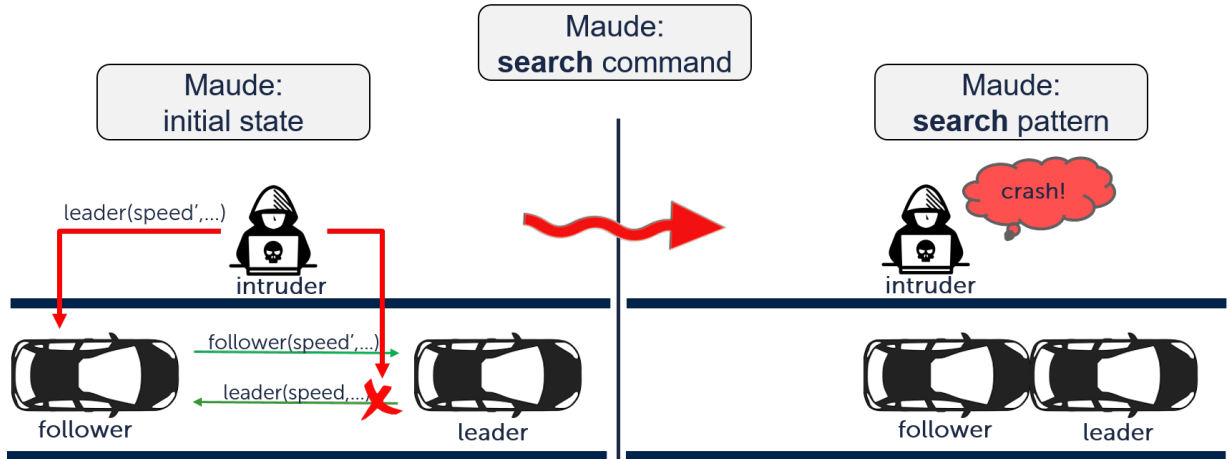


Figure 2.11: Illustration of both initial state and search pattern of our framework.

(i) the absence of security mechanisms specified in the CACC platoon model, a.k.a. the baseline scenario, and (ii) the inclusion of a security mechanism specified to mitigate attacks targeting the CACC platoon system. In the next paragraphs, we discuss the selection and positioning of the security mechanism in the CACC platoon model.

During the design of the CACC architecture, one or more security architecture patterns may be selected to mitigate threat scenarios that may violate the security properties of components of the CACC system. We consider a threat scenario that has the potential to compromise the integrity property of the follower systems. This threat scenario involves tampering with the speed messages produced by the leader to manipulate the followers into increasing their speed values. This manipulation may result in a front-to-rear crash, i.e., violation of the safety property. *Security monitor* is a security architecture pattern to mitigate threat scenarios that may violate the integrity property of the system. Acting as a runtime monitoring and enforcement mechanism, the security monitor intercepts incoming or outgoing messages and verifies their compliance with predefined security policies. If a policy violation occurs, the security monitor takes appropriate enforcement actions, which may include blocking the incoming or outgoing messages. Our article [45] provides details on the instantiation of the security monitor pattern.

We specify the security monitor pattern in the CACC platoon model. We consider the security requirements associated with this pattern, which encompass requirements regarding its (i) placement, (ii) security policies, and (iii) enforcement actions. To this end, we specify an instance of the security monitor for each follower in the platoon. These monitors are strategically (i) positioned to intercept and analyze the messages received from the leader. We specify plausibility checks to (ii) evaluate the validity and integrity of incoming messages. The incoming message is (iii) blocked if the plausibility check detects a malicious message (i.e., violation of the security policy). We evaluate the effectiveness of the security monitor against five different attacks. Our article [46] provides a comprehensive overview of the effectiveness achieved by the specified security monitors.



In our research, we have identified benefits that arise from the integration of the introduced lightweight semantics with formal verification. Firstly, when connecting lightweight semantics with formal verification, the pattern requirements play an important role in assisting formal verification experts in specifying security mechanisms. This is particularly evident in terms of considering the suitable placement of these security mechanisms within the system. Moreover, formal verification experts can define how these mechanisms should be instantiated based on the pattern requirements. This encompasses the specification of essential elements such as security policies and enforcement decisions for the security monitor. The introduced lightweight semantics provide a framework for capturing these requirements, enabling a more systematic and comprehensive approach to specify security mechanisms. Additionally, once the security mechanism has been specified based on the lightweight semantics, formal verification experts are able to verify the system's properties. This verification process involves analyzing and assessing the system behavior in both scenarios: with the presence of the security mechanism and without it. By considering these distinct scenarios, formal verification experts can thoroughly evaluate the impact and effectiveness of the security mechanism on the system's properties.

In summary, we provide a formal framework to verify the safety property (derived from a safety goal) of a CACC platoon system. Our framework enables the specification of security attacks (based on an intruder model) that may violate the safety property. It enables the specification of security mechanisms that may be derived from pattern requirements. Our framework assesses the effectiveness of the specified security mechanism in mitigating attacks against CACC platoon systems.

## 2.6 Discussion

We provide lightweight semantics to specify a vocabulary, along with its semantics, for safety artifacts resulting from HARA, FTA, and STPA. While this thesis primarily focuses on STPA, we also investigate how to use the results of a FTA to recommend safety architecture patterns. A fault tree includes a minimal cut set consisting of failures (a.k.a. basic events) often represented as leaf nodes. Our reasoning rules takes such basic events into account to recommend safety architecture patterns. For example, in our article [45] (see also the fault tree depicted in Figure 2.4), we considered a very simple fault tree with only one layer of failure events triggering a top event (hazard), where a single pattern may be recommended to address the minimal cut set. In a real-world scenario, a fault tree may consists of several layers between the basic events and the top event. Moreover, there may be several minimal cut sets at the lowest level (not just one as exemplified in [45]). Consequently, we overlook the information situated between the basic events and the top event. This decision could result in the recommendation of several safety architecture patterns to address the minimal cut sets and the top event. A more cost-effective approach would be to analyze the information preceding the top event and recommend a pattern to address it, thus addressing all the information in the lower layers (including the minimal cut sets). We left to future work the modification of our reasoning rules to address this issue. The solution may be

relatively straightforward, as it would entail incorporating the causal relationship between the minimal cut sets and the information from other layers until reaching the top event as part of our input. This thesis considers the failure mode erroneous or loss from either failures (FTA) or loss scenarios (STPA). These two failure modes are well-recognized within the research community. It is worth noting that the industry may consider additional failure modes beyond erroneous and loss. Our DSL may have the capability to specify further failure modes, particularly for establishing connections with safety architecture patterns. That is, the specification of further failure modes becomes feasible when we can connect the problem addressed by the pattern with the specific failure mode such that the pattern can be recommended. The investigation of further potential failure modes is left to future work.

We provide lightweight semantics to specify a vocabulary, along with its semantics, for security artifacts resulting from TARA. In this thesis, our DSL focuses on the specification of security architecture patterns to mitigate threat scenarios that violate the security properties of data asset, which encompass either input and output data of assets. We recognize the importance of security architecture patterns to mitigate threat scenarios against either the software or hardware of assets. Those threats are particularly relevant in the context of automated driving systems. Examples of such patterns are authenticated secure boot (for software) and hardware security module (for hardware). Our DSL may be extended to enable the specification of these security architecture patterns. That is, the pattern attributes may be extended to include the threat scope, such as tampering of software, where the authenticated security boot may be applied. Our reasoning rules would require corresponding adaptations to accommodate these extended attributes and specifications. Our DSL and reasoning rules may be extended to align with the guidelines recommended by the UN Regulation No. 155 [62]. This regulation outlines security requirements that shall be met by Original Equipment Manufacturers (OEMs) for all new vehicle types for type approval from July 2024. Regulation 155 provides a table consisting of a list of threats and possible mitigation solutions specifically relevant to automotive systems. Consider a public element GNSS located outside the system boundary and the asset localization component situated within the system boundary. According to R155, one potential threat for this scenario is “Spoofing of messages by impersonation GNSS messages”, and the corresponding mitigation is “The vehicle shall verify the authenticity and integrity of messages it receives”. This mitigation is very high-level and does not explicitly specify which security mechanism shall be implemented to satisfy messages authenticity and integrity. For this reason, this thesis opted not to strictly follow the R155 mapping, as we aimed to be more precise on which security mechanism shall be implemented. However, some OEMs may choose to strictly follow the R155 recommendations. In such cases, one may extend our DSL and reasoning rules to align with these specific recommendations. This thesis follows the safety-informed security approach [35] and provides a mapping from safety artifacts to security artifacts inspired by related work approaches. As a result, this mapping provides the instantiation of the safety impact rating. Typically, the financial, operation and privacy impact are also expected to calculate the overall risk associated with a threat scenario. These impact values are important to OEMs to deciding, e.g., which assets to prioritize during the development of security mechanisms. Our DSL may be extended to

---

include predicates that specify financial, operation and privacy impact, where users would be required to provide this information as input. Our DSL expects manual input by the user to specify artifacts regarding attack feasibility rating. This manual input is required because this activity demands expertise from a security engineer. We envision that the automation of the attack feasibility rating activity may be possible if a database is provided as input. OEMs may have such a database from their previous projects. The risk value can be automatically computed if both impact rating and attack feasibility rating are provided. In this thesis, we provide security architecture patterns regardless of the risk value. In a real-world scenario, OEMs or suppliers consider the risk value when determining to reduce the risk by implementing security architecture patterns. This decision is tailored for each specific OEM or supplier. Our reasoning rules can be augmented to include conditions for when a security architecture pattern shall be recommended based on the risk value.

# Chapter 3

## Supplementary Related Work

In addition to the related work that will be discussed in the forthcoming chapters, this chapter provides supplementary details on relevant related work that has served as a foundation for this thesis.

**From Safety Artifacts to Security Artifacts.** The field of safety and security co-analysis has a vast literature, which serves as the foundation for our mapping of safety artifacts to security artifacts. We present an overview of key approaches that have influenced the development of our mapping. STPA and Six Step Model [63] is an approach that integrates safety and security artifacts in the context of automated driving systems. This approach is built upon the Six Step Model (SSM) approach [64], which was proposed by the same authors. The STPA and SSM approach consists of six steps, namely (i) system's functional architecture, (ii) system's structure architecture, (iii) safety artifacts, (iv) security artifacts, (v) safety countermeasures, and (vi) security countermeasures. The first step consists of identifying system functions. The second step consists of organizing the system into several layers of decomposition (i.e., a structure architecture), where the relationships between the systems and functions are identified, determining which systems implement specific functions. The third step consists of identifying safety artifacts (e.g., hazards and loss scenarios) and the relationship between safety artifacts and systems. HARA and STPA are performed to identify such safety artifacts. The identified safety artifacts are considered as input for the security analysis. The fourth step consists of identifying safety countermeasures and their relationships with safety artifacts are established. The fifth step consists of identifying security artifacts (e.g., threats and attack paths) and their relationships with safety countermeasures. TARA is performed to identify security artifacts. Finally, in the last step of the STPA and Six-Step Model, security countermeasures are identified along with their relationships to the security artifacts. The SSM approach enables the traceability of various artifacts. For example, it allows for the identification of threats that can impact safety countermeasures and establishes the relationships between security artifacts and safety artifacts. The latter can be viewed as how security artifacts may trigger safety artifacts. The Safety-Aware Hazard Analysis and Risk Assessment (SAHARA) approach, introduced in [65], expands on the HARA analysis outlined in ISO 26262 [10]

---

by incorporating security threats that may impact safety. SAHARA utilizes the STRIDE methodology [26], which is a well-established threat modeling technique developed by Microsoft. STRIDE encompasses six categories of threats, namely **S**poofing, **T**ampering, **R**epudiation, **I**nformation disclosure, **D**enial of service, and **E**levation of privilege. These threats are derived by SAHARA based on the security objectives that the system is expected to fulfill. For example, tampering can be identified as a threat to the integrity property. The Bosch engineers [66] proposed an approach for deriving security artifacts of TARA from safety artifacts computed by a HARA analysis. Specifically, the Bosch approach recommends that (i) assets are derived from safety goals, (ii) threats are derived from the violation of safety goals, (iii) damage scenarios are derived from hazards, and (iv) impact rating values are derived from severity and controllability of ASIL.

Our method, introduced in [42], for mapping security artifacts from safety artifacts is inspired by the aforementioned approaches. Similar to the Bosch and STPA and Six Step Model approaches, our method assumes that a safety analysis has been conducted as a preliminary step. Based on the outcomes of the safety analysis, our method derives security artifacts. To this end, we agree with the Bosch approach in terms of deriving damage scenarios from hazards and determining impact rating values based on the severity and controllability of ASIL. However, we differ from the Bosch approach regarding the derivation of assets from safety goals. We find it challenging to derive such assets from safety goals as recommended by Bosch. This challenge arises primarily because safety goals are specified as high-level requirements without explicitly mentioning concrete system architecture elements, i.e., which elements are required to be protected to avoid hazards. Consequently, our method aligns with the principles of STPA and Six Step Model, considering the outputs of STPA alongside HARA. In other words, our method derives assets from the loss scenarios computed by STPA. Our method derives the security properties that the asset shall satisfy from the associated failure mode of a loss scenario. In line with the recommendations of ISO 21434 and similar to the SAHARA approach, our primary approach for modeling threats is based on the STRIDE methodology. We determine the threat category based on the desired security property of the asset. In our work, we specifically focus on tampering and denial of services threats that may compromise the integrity and availability of safety-critical topics (i.e., assets), respectively. We also consider that spoofing threats have the potential to compromise the integrity of safety-critical functions.

**Attacks Against Automated Driving Systems.** We present an overview of several attacks targeting automated driving systems, including attacks targeting automated driving vehicles adopting service-oriented architectures. These attacks have served as a source of inspiration for us to formalize our intruder model described in [42].

A recent systematization of knowledge article [67] provides a comprehensive overview of the state-of-the-art of the literature. The article analyzed 53 articles and taxonomize them based on security critical aspects, including attacks targeting vehicle sensors. The article “Drift with devil” [68] describes how an intruder may manipulate location information by spoofing GPS radio signals. This attack is effective even against localization components

using multi-sensor fusion. The authors discovered scenarios where the spoofed GPS signals become the primary input source in the fusion process, overriding other sensor data. LiDAR sensor signals may be spoofed to remove obstacles on the road [69]. LiDAR technology is used to create 3D maps of the surrounding environment and identify objects, including vehicles ahead of the automated driving vehicle. This spoofing attack [69] injects additional 3D points in a different location, but in the same direction as an object in front of the automated driving vehicle. The attack is effective because LiDAR's default mode records only one return signal per ray direction. As a result, the automated driving vehicle's LiDAR may fail to detect the actual target objects and instead identify the manipulated object injected by the intruder. Camera signals can be manipulated to deceive the perception component by spoofing video frames [70, 71]. This manipulation can involve disrupting the video stream by terminating it or replacing the legitimate stream with a malicious one. The vulnerability of this attack lies in the absence of authentication mechanisms between the camera and the client (e.g., perception component). The client accepts requests from any source MAC address that matches the camera's MAC address. In addition, the lack of encryption in the transmission of the video stream allows an intruder to intercept the camera feed, reconstruct video frames, and decode the payload to access the image content. An intruder may carry out spoofing attacks to manipulate the range and velocity measured by the automated driving vehicle's radar sensor [72]. Through this attack, the intruder can induce an emergency braking response or deceive the vehicle into engaging in sudden acceleration. An intruder may exploit vulnerabilities within the Bluetooth stack to lock the vehicle's brakes [20]. In the work by [22], the authors investigate potential security issues in the service discovery mechanism of vehicle Service-Oriented Architecture (SOA), specifically in SOA systems utilizing the SOME/IP protocol. The authors describe an attack that allows an intruder to perform Man-in-the-Middle attacks between publishers and subscribers. This attack violates the integrity property of the published data, potentially resulting in erroneous behavior from subscribers. In the study conducted by [73], the authors investigate potential cases of overprivilege among publishers and subscribers in the Apollo system [19]. They propose a tool called AVGuardian, which can identify several instances of overprivilege in the Apollo 5.0 code base. These instances include scenarios where (a) the gnss driver may abuse a publish-overprivileged field in the transformation topic to manipulate the estimated position of an observed obstacle on the road, and (b) the compensator may exploit a publish-overprivileged field in the PointCloud topic to remove a detected obstacle from the road.

Our intruder model, proposed in [42], presents a formalized representation of the primary capabilities required by an intruder to carry out the aforementioned attacks at the architectural level. These capabilities encompass the ability to conduct (i) spoofing attacks through an external interface (e.g., from sensors) (ii) and (b) Man-in-the-Middle attacks through an internal interface (e.g., between internal components), thereby compromising the security properties of safety-critical topics in automated driving vehicles adopting service-oriented architectures. It is worth noting that the attacks exploiting overprivileged instances can be viewed as a specific instance of the Man-in-the-Middle attack. These attacks leverage the vulnerability of overprivileged instances to manipulate the communication and

---

compromise properties of safety-critical components within automated driving systems.

**Safety Architecture Patterns.** Safety architecture patterns have been widely used in the design of safety-critical systems, such as automated driving systems. These patterns provides a systematic framework for incorporating safety mechanisms into the system architecture. Several articles have been published on the topic of safety patterns, including [74, 14, 13, 75, 76, 61]. We present the two key articles that have had the greatest influence on the development of our automated methods for safety architecture patterns.

The PhD thesis by Armoush [13] focuses on the application of safety architecture patterns in the context of safety-critical systems. Armoush investigates several safety architecture patterns for addressing safety artifacts in safety-critical systems. The thesis proposes a set of safety architecture patterns that may be used to tolerate either hardware or software faults. Hardware redundancy is used to tolerate hardware faults and design diversity is used to tolerate software faults. Examples are the Homogeneous Duplex pattern, which tolerates hardware faults, and the Acceptance Voting pattern, which tolerates software faults. To provide a structured and systematic approach for selecting appropriate safety architecture patterns, Armoush proposes a recommendation system that considers the safety integrity level of identified hazards. This systematic system is an extension of the procedure proposed by the IEC 61508 standard. Armoush's system of recommendations contains seven categories: Not Recommended, Weakly Not Recommended, No Recommendation, Recommended, Moderately Recommended, and Highly Recommended. The assignment of each category is based on an integer value derived from the techniques employed by the safety architecture pattern, such as fault detection and voter mechanisms. Furthermore, the safety architecture patterns are presented in a pattern template, which includes pattern information regarding, e.g., structure and implementation. The structure aspect of the template provides guidelines on how the pattern should be instantiated within the system architecture. The implementation aspect aids in extracting the pattern requirements specific to a particular system. The article [74] investigates fault tolerant patterns in the context of avionics systems. The authors investigate different levels of criticality and fault tolerant pattern required for such systems. A key aspect in selecting an appropriate fault-tolerant pattern is understanding the failure modes of system functions, such as loss of function or malfunction (a.k.a. erroneous). Specific features may be developed into pattern components to avoid loss of function or malfunction. These features include watchdog timers, which identify function loss, and built-in tests (also referred to as plausibility checks), which identify malfunctions. The authors acknowledge the challenge of developing a built-in test feature that can detect 100% of malfunctions. The detection coverage of the built-in test feature is estimated to be around 95%. The article further explores various safety architecture patterns, including fault-tolerant patterns that incorporate redundant components. These patterns are designed to ensure fail operational, fail passive, or fail-safe capabilities. The outcome is a comprehensive table that outlines the number of failures (if any) that a safety architecture pattern can avoid while ensuring either fail operational, fail passive, or fail-safe capabilities.

Our articles [39, 40, 45] provide automated methods for recommending safety architecture patterns. These recommendations are determined by four factors, including ASIL class, fault mode, and failure mode. We consider the findings of [13] when establishing criteria for recommending patterns based on the ASIL class and fault dimension. Furthermore, our recommendation process for failure mode is influenced by the results of [74]. Building upon the insights from [74], our article [45] takes a step further by including conditions to recommend patterns based on their fail operational, fail passive, or fail-safe capabilities. The choice of these capabilities is left to the user. Additionally, the user may also select the type of safety architecture pattern, whether the recommended pattern shall be fault detection or fault tolerant pattern.

**Security Architecture Patterns.** Safety architecture patterns are generally more established than security architecture patterns. Safety has been an industry concern for a longer time compared to security. As a result, safety-related articles, standards and regulations have been developed over the years, leading to the establishment of safety architecture patterns. Since security architecture patterns are relatively new in comparison to safety architecture patterns, there is no established set of security architecture patterns available as discussed by [77]. Establishing a set of security architecture patterns presents challenges due to the dynamic nature of security threats, with new (inside and outside) threats emerging regularly [78]. Therefore, the scientific community is still in the early stages of reaching a consensus on the definition of security architecture patterns for application in safety-critical systems. Some articles have been published on the topic of security architecture patterns, such as [78, 24, 27, 28, 29]. We present the two key articles that have had the greatest influence on the development of our automated methods for security architecture patterns.

According to [78], having a clear understanding of security building blocks is essential for using security architecture patterns. Examples of security building blocks are data flows, decision points and enforcement points. *Data flows* (a.k.a. information flow) denote the points at which data enters and exits a system. Keeping track of data flows is crucial for identifying sensitive or safety-critical data that requires protection from a security point of view. *Decision points* denote locations within the system where security-related decisions are made, while *enforcement points* are the locations where security-related enforcement decisions are made. It is crucial to clearly identify these points, especially for specifying requirements and transferring such requirements from the concept phase to the implementation and verification phase. In line with this, the authors emphasize the importance of establishing a connection between security requirements and pattern components to ensure traceability throughout the software life cycle, i.e., security requirements shall be linked to their corresponding security architecture patterns, allowing for a traceable path from the initial requirements to the final deployed software. The DSL proposed by this thesis (see, e.g., article [43]) captures relevant data flows by conducting attack path enumerations. These data flows are used as a basis for identifying suitable locations to instantiate security architectures. Decision and enforcement points are tailored to security architecture patterns. For example, for the Firewall pattern, the firewall component serves as both the decision



and enforcement point. Our DSL also enables the specification of security requirements and their allocations to pattern components. A set of security architecture patterns for connected and automotive systems are proposed by [27]. The authors introduce a template to describe such patterns, similar to the one considered by this thesis. With this template, several security architecture patterns are instantiated, categorized based on their applicability to either inward-facing communications or outward-facing communications. These communication types align with the concepts explored in this thesis, where inward communications represent internal interfaces and outward-facing communications represent external interfaces. The authors advocate for the use of the STRIDE methodology to align security properties with potential threats. By employing the STRIDE descriptors, suitable patterns can be identified. Similarly, this thesis (see, e.g., article [43]) incorporates both the security property and its associated threat when recommending security architecture patterns. The template proposed by the authors also include the consequences of using security architecture patterns. We follow a similar approach in [45] to state the safety consequences associated with the utilization of security architecture patterns.

# Chapter 4

## Knowledge Representation and Reasoning for Safety System Architectures

Chapter 4 proposes a Domain-Specific Language (DSL) for specifying system architecture artifacts, safety artifacts, and safety architecture patterns. Safety reasoning rules have been specified to enable the automation of safety architecture patterns. The DSL and safety reasoning rules have been implemented as a dedicated logic programming tool, which has been validated using examples taken from the automotive domain.

**Contributing article:** Yuri Gil Dantas, Antoaneta Kondeva, and Vivek Nigam: Less Manual Work for Safety Engineers: Towards an Automated Safety Reasoning with Safety Patterns. ICLP Technical Communications 2020: 244-257

**Copyright information:** The article [39] is licensed under a Creative Commons Attribution 4.0 International license (<https://creativecommons.org/licenses/by/4.0/>). <https://doi.org/10.4204/EPTCS.325.29>.

**Author contributions:** The concept for the publication was jointly developed by Yuri Gil Dantas, Antoaneta Kondeva, and Vivek Nigam. They jointly performed a literature review to identify the gaps and challenges faced by safety engineers when selecting safety architecture patterns for safety-critical systems. Through the literature review, the authors recognized that safety engineers use certain conditions (e.g., ASIL class) for selecting and recommending safety architecture patterns. The outcome of the literature review enabled the authors in specifying semantically-rich (safety) architecture patterns for automated driving systems. In terms of tool, Vivek Nigam gave the idea of using a logic programming system (i.e., DLV) to enable the automated recommendation of safety architecture patterns. Yuri Gil Dantas took the lead in implementing large parts of the tool developed in DLV and carried out the experiments presented in the article. Yuri Gil Dantas and Vivek Nigam

jointly wrote the first draft of the article. Antoaneta Kondeva assisted them in improving the article. Yuri Gil Dantas handled subsequent revisions and corrections.

# Less Manual Work for Safety Engineers: Towards an Automated Safety Reasoning with Safety Patterns

Yuri Gil Dantas

Antoaneta Kondeva

Vivek Nigam

fortiss GmbH  
Research Institute of the Free State of Bavaria  
Guerickestraße 25  
80805 München, Germany

dantas@fortiss.org

kondeva@fortiss.org

nigam@fortiss.org

The development of safety-critical systems requires the control of hazards that can potentially cause harm. To this end, safety engineers rely during the development phase on architectural solutions, called safety patterns, such as safety monitors, voters, and watchdogs. The goal of these patterns is to control (identified) faults that can trigger hazards. Safety patterns can control such faults by e.g., increasing the redundancy of the system. Currently, the reasoning of which pattern to use at which part of the target system to control which hazard is documented mostly in textual form or by means of models, such as GSN-models, with limited support for automation. This paper proposes the use of logic programming engines for the automated reasoning about system safety. We propose a domain-specific language for embedded system safety and specify as disjunctive logic programs reasoning principles used by safety engineers to deploy safety patterns, e.g., when to use safety monitors, or watchdogs. Our machinery enables two types of automated safety reasoning: (1) identification of which hazards can be controlled and which ones cannot be controlled by the existing safety patterns; and (2) automated recommendation of which patterns could be used at which place of the system to control potential hazards. Finally, we apply our machinery to two examples taken from the automotive domain: an adaptive cruise control system and a battery management system.

## 1 Introduction

The development of safety-critical systems, such as vehicles, aircraft and medical devices aims to achieve two goals: (1) to develop systems that cannot cause any harm, and (2) to convince regulatory bodies about the safeness of the system by demonstrating compliance to safety standards [18, 17].

To achieve the first goal, safety engineers perform safety analysis to ensure that systems cannot cause any harm. For example, *Hazard Analysis* [18, 16] identifies the main hazards that shall be controlled. Other safety techniques, e.g., FTA [16], STPA [21], FMEA [16], HAZOP [8], break down the identified main hazards into component hazards (a.k.a component failures), i.e., faults that can trigger main hazards. Safety engineers commonly use *safety architectural patterns* [5, 22, 23] to control the identified component hazards (or hazards for short) thus controlling the main hazards. To achieve the second goal, safety engineers shall develop a safety case [18, 24] for the system under development. The purpose of the safety case is to both (a) ensure that all hazards have been analyzed and (b) answer why a safety pattern has been deployed at a particular component to control which hazard.

Safety cases are often documented in textual form, or by models e.g., the Goal Structure Notation (GSN) [7]. These models, however, have limited support for automated reasoning [19]. It is not possible to automatically check whether safety arguments used in a safety case are correct, i.e., check whether all hazards have been controlled by, e.g., safety patterns. This is because the *safety reasoning* used to support

system safety is implicitly written textually thus lacking the precise semantics to enable automation [25]. As a result, correctness checks are performed manually, possibly leading to human errors.

Our vision is to build an incremental development process for system safety and security assurance cases using automated methods that incorporate safety and security reasoning principles. This paper is the first step towards achieving this vision. We provide safety reasoning principles with safety patterns used during the definition of system architecture for embedded systems. We specify these principles using logic and logic programming as they are suitable frameworks for the specification of reasoning principles as knowledge bases and using them for automated reasoning [4].

Our main contributions are threefold:

- **Domain-Specific Language (DSL):** We propose a DSL for safety reasoning with safety patterns. Our DSL includes (1) architectural elements, both functional components and logical communication channels; (2) safety hazards including guidewords used in typical analysis, *e.g.*, erroneous or loss of function; (3) a number of safety patterns including n-version programming, safety monitors, and watchdogs;
- **Reasoning Principles:** We specify key reasoning principles for determining when a hazard can be controlled or not, including reasoning principles used to decide when a safety pattern can be used to control a hazard. These reasoning principles are specified as Disjunctive Logic Programs [11] based on the DSL proposed;
- **Automation:** We illustrate the increased automation enabled by the specified reasoning principles using the logic programming engine DLV [20]. Our machinery enables two types of automated reasoning: (1) *Controllability*: which hazards can be controlled by the given deployed safety patterns and which hazards cannot be controlled. (2) *Safety Pattern Recommendation*: which safety patterns can be used and where exactly they should be deployed to control hazards that have not yet been controlled.

We validate our machinery<sup>1</sup> with two examples of safety-critical embedded systems taken from the automotive domain. The first example is an *Adaptive Cruise Control* system installed in a vehicle to adapt its speed in an automated fashion without crashing into objects in front and at the same time trying to maintain a given speed. The second example is a *Battery Management System* [22] responsible for ensuring that a vehicle battery is charged without risking it to explode by, *e.g.*, overheating. Our machinery infers a number of possible solutions involving different safety patterns that can be used to control identified hazards.

## 2 Motivating Examples

This section describes two examples from the automotive domain. We refer to these examples as Adaptive Cruise Control system (ACC) and Battery Management System (BMS). We use the ACC as a running example throughout the paper. We get back to the BMS example in Section 7.

**Adaptive Cruise Control (ACC).** Consider as a motivating example, a simplified ACC responsible for maintaining safe distance to objects in front of its vehicle. The ACC is a critical system as harm, *e.g.*, accidents, may occur if the ACC is faulty.

---

<sup>1</sup>All machinery needed to reproduce our results are publicly available: <https://github.com/ygdantas/safpat>

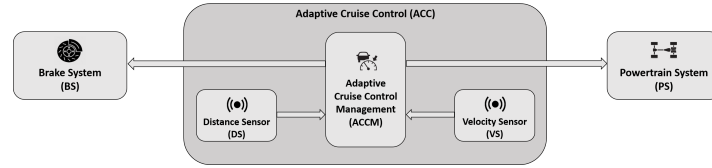


Figure 1: Adaptive Cruise Control (ACC) Functional Architecture

Figure 1 depicts the main functions composing the ACC. ACC uses information from two sensing functions: (1) distance sensor function (DS) that computes the distance to objects immediately in front; (2) velocity sensor function (VS) that computes the vehicle's current speed. The ACC Management function (ACCM) computes (adequate) acceleration and braking values for the vehicle which are sent to the power-train control (PS) and brake control functions (BS), respectively. Notice that PS and BS are not part of the ACC but interact with the ACC.

To address the safety of the ACC, safety analysis are carried out, such as Hazard Analysis, to determine main hazards. The main hazard is:

**H0<sub>acc</sub>**: The vehicle does not maintain a safe distance to any object in front.

We identify two hazards, **H1<sub>acc</sub>** and **H2<sub>acc</sub>**, that may lead to **H0<sub>acc</sub>**. The words *loss* and *erroneous* are used by safety engineers to describe hazards: *loss* is used when a hazard is triggered whenever a function is not working, and *erroneous* when a function is working but not correctly.

- **H1<sub>acc</sub>– Erroneous ACC**: ACC computes incorrect acceleration or braking values;
- **H2<sub>acc</sub>– Loss of ACC**: ACC is not functioning.

These hazards are subsequently further broken down to identify which sub-functions can trigger them using, *e.g.*, Fault Tree Analysis. The following hazards may lead to **H1**:

- **H1.1<sub>acc</sub>– Erroneous DS**: The DS computes an incorrect distance to the car in front;
- **H1.2<sub>acc</sub>– Erroneous VS**: The VS computes an incorrect velocity;
- **H1.3<sub>acc</sub>– Erroneous ACCM**: The ACCM computes wrong acceleration or braking values.

**Battery Management System (BMS).** We consider a simplified BMS responsible for controlling a rechargeable electric car battery [22]. The BMS is a critical system as harm, *e.g.*, battery explosions, may occur if it does not compute the charging state of the battery correctly.

Figure 2 depicts the main functions composing the BMS. The charging interface (CI) represents the interface at the charging car station. This interface is triggered while recharging the battery (BAT) of the car. BMS receives relevant information (*e.g.*, voltage and temperature values) from BAT so that it can compute the charging state of BAT. Depending on the state of BAT, BMS sends signals of activation or deactivation of the external charger to CI. These signals are sent through a CAN bus. CI is considered the only function accessible by external users (*e.g.*, drivers). To avoid that an intruder can access the CAN bus through CI, a firewall (FW) is placed between BMS and CI.<sup>2</sup> This decision, however, comes at a safety impact, as mentioned below. The main hazard considered here is:

**H0<sub>bms</sub>**: The BAT is overcharged leading to its explosion.

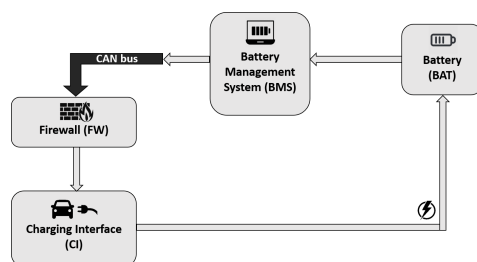


Figure 2: Battery Management System (BMS) Functional Architecture

We identify one erroneous hazard  $\mathbf{H1}_{bms}$  that may lead to  $\mathbf{H0}_{bms}$ .

- $\mathbf{H1}_{bms}$ – **Erroneous CI:** The CI sends charging signals when BAT is fully charged.

The following three hazards may lead to  $\mathbf{H1}_{bms}$ . We use the word *omission* as a specialization of the erroneous behavior whenever the corresponding function does not provide an output when such an output is expected, *e.g.*, not outputting a fail-safe signal.

- $\mathbf{H1.1}_{bms}$ – **Erroneous BMS:** The BMS sends wrong signals to CI;
- $\mathbf{H1.2}_{bms}$ – **Erroneous CAN:** The CAN bus sends wrong signals to CI;
- $\mathbf{H1.3}_{bms}$ – **Omission FW:** The FW incorrectly blocks signals from BMS.

Hazards are also associated with severity class denoting the level of harm it can cause. Severity classes range over *no effect*, *minor*, *major*, *fatal*, and *catastrophic*. The hazards described in this section are classified as *catastrophic*, which means that they shall be strongly controlled.

### 3 Preliminaries

**Safety Architectural Patterns.** In the architectural level, a number of safety patterns are typically used for embedded system safety [23, 5]. Examples of such patterns are Heterogeneous Duplex Redundancy (HDR), Triple Modular Redundancy (TMR), N-Version Programming (NProg), Safety Monitors (SafMon), and Watchdog (WD).

The goal of these patterns is to control some type of hazards provided some conditions are satisfied. WDs are used to detect when there is loss of function, thus controlling hazards associated with a *loss* of function. SafMons are used to check whether a function is computing correctly, thus controlling hazards associated with *erroneous* functions. HDR and TMR are used to control hazards by increasing the redundancy of existing hardware, thus reducing the overall fault rate. They can also be used to increase the redundancy of paths in the system in case messages are lost or incorrectly computed. NProgs are used control hazards associated with possibly *erroneous* software functions by increasing the redundancy of such functions.

**Answer-Set Programming and Disjunctive Logic Programs.** We assume that the reader is familiar with Answer-Set Programming (ASP) and provide only a brief overview here. Let  $\mathcal{K}$  be a set of propositional variables. A *default literal* is an atomic formula preceded by *not*. A propositional variable and a

<sup>2</sup>We refer the reader to [22] for more insights on why adding a FW between BMS and CI makes the system more secure.

default literal are both *literals*. A rule  $r$  is an ordered pair  $Head(r) \leftarrow Body(r)$ , where  $Head(r) = \ell$  is a literal and  $Body(r) = \{\ell_1, \dots, \ell_n\}$  is a set of literals. Such a rule is written as  $\ell \leftarrow \ell_1, \dots, \ell_n$ . An *Answer-Set Program* (LP) is a set of rules. An interpretation  $M$  is an *answer set* of a LP  $P$  if  $M' = least(P \cup \{not A \mid A \notin M\})$  and  $M' = M \cup \{not A \mid A \notin M\}$ , where *least* is the least model of the *definite logic program* obtained from the program  $P$  by replacing all occurrences of *not A* by a new atomic formula *not A*.

The interpretation of the default negation *not* assumes a *closed-world* assumption. That is, we assume to be true only the facts that are explicitly supported by a rule. For example, the following program  $P$  with three rules has two answer-sets  $\{a, c\}$  and  $\{b\}$ :

$$a \leftarrow not\ b \quad b \leftarrow not\ a \quad c \leftarrow a$$

DLV is an engine implementing disjunctive logic programs [11] based on ASP semantics [12]. In particular, a rule may have disjunction in its head, e.g.,  $a_1 \vee \dots \vee a_m \leftarrow \ell_1, \dots, \ell_n$ , where  $a_i$  for  $0 \leq i \leq m$  are atomic formulas. For example, consider the program  $P_1$  with the two clauses  $a \vee b$  and  $c \leftarrow a$ . It has the same two answer-sets as the program  $P$ . If a rule's head is empty, i.e.,  $m = 0$ , then it is a constraint. For example, if we add the clause  $\leftarrow b$  to  $P_1$ , then the resulting program has only one answer-set  $\{a, c\}$ .

In the remainder of this paper, we use the DLV notation writing  $:-$  for  $\leftarrow$  and  $\vee$  for  $\vee$ . For example, the program  $P_1$  is written as  $a \vee b$  and  $c :- a$ .

## 4 Basic DSL: Functional, Hardware and Safety Patterns

This section introduces our domain-specific language, called SafPat, for enabling automated safety reasoning with safety patterns. Tables 1 and 2 describe SafPat's main elements, i.e., key terms and predicates. Table 1 describes the language used to specify functional and hardware architecture, and safety analysis, while Table 2 describes the predicates used to specify selected safety patterns. We illustrate SafPat by using the ACC example described in Section 2.

**Example 1** *The functional architecture depicted in Figure 1 is specified by the following atomic formulas, or facts, using the notation of the DLV prover [20]:*

```
cp(acc). cp(accm). cp(ds). cp(vs). cp(bs). cp(ps).
subcp(accm, acc). subcp(ds, acc). subcp(vs, acc). ch(dsaccm, ds, accm).
ch(vsaccm, vs, accm). ch(accmbs, accm, bs). ch(accmps, accm, ps).
if(if1, [vsaccm, accmbs]). if(if2, [dsaccm, accmbs]).
```

*The fact  $ch(vsaccm, vs, accm)$  denotes the logical communication between the VS and the ACCM. The information flow  $if1$  denotes data flows from VS to BS. The facts below specify which functions are implemented as software, e.g., ACCM, and which as hardware, e.g., DS.*

```
sw(accm). hw(ds). hw(vs). hw(ps). hw(bs).
```

*Finally, the ACC hazards and their relations are specified by the following facts:*

```
hz(h1, acc, err, cat). hz(h2, acc, loss, cat). hz(h11, ds, err, cat).
hz(h12, vs, err, cat). hz(h13, accm, err, cat).
subHz(h11, h1). subHz(h12, h1). subHz(h13, h1).
```

*For example, the hazard  $HI.3_{acc}$  ( $h13$ ) is a sub-hazard of  $HI_{acc}$  ( $h1$ ).*



| <b>Functional, Hardware and Safety Analysis</b> |   |
|---|---|
| <b>Fact</b>                                     | <b>Denotation</b>   |
| $cp(id)$  | $id$ is a function in the system.   |
| $subcp(id_1, id_2)$                             | $id_1$ is a sub-function of the function $id_2$ .   |
| $ch(id, id_1, id_2)$                            | $id$ is a logical channel connecting an output of the function $id_1$ to an input of the function $id_2$ . Notice that it denotes a unidirectional connection.  |
| $if(id, \vec{ch})$                              | $id$ is an information flow following the channels in $\vec{ch}$ .  |
| $hw(id)$  | Function $id$ is implemented as hardware, <i>e.g.</i> , circuit connected to sensors.   |
| $sw(id)$  | Function $id$ is implemented as a software.   |
| $hz(id, id_c, tp, sv)$                          | $id$ is a hazard associated with the function $id_c$ is of type $tp$ , where $tp \in \{err, loss, omission, late, early\}$ , and severity $sv$ , where $sv \in \{minor, major, fatal, cat\}$ . $err, loss, omission, late,$ and $early$ denote, respectively, erroneous, loss of function, omission, late and early types of hazards. $minor, major, fatal, cat$ denotes, respectively, minor, major, fatal and catastrophic severity levels. |
| $subHz(id_1, id_2)$                             | $id_1$ is a hazard causing hazard $id_2$ .  |

Table 1: SafPat: a DSL for specifying functional, hardware and safety analysis.

Due to space limitations, we illustrate only the safMon pattern. The remaining patterns follow a similar reasoning. We refer the reader to [5, 23] for detailed description of these patterns.

The safMon pattern is depicted by all dashed elements in Figure 3 including channels. This safMon is associated to the function  $id_c$  and is used to detect whether  $id_c$  is computing erroneous values. To this end, it takes the values of  $id_c$ 's inputs ( $\vec{I}$ ) and outputs ( $\vec{O}$ ) to the function  $sm$  through the channels  $\vec{I}_{sm}$  and  $\vec{O}_{sm}$ . The channel  $fs$  connecting  $sm$  with  $id_c$  is used to send fail-safe commands whenever abnormal input-output relations are detected by  $sm$ .

In SafPat, one identifies a safMon by specifying the fact  $safMon(id, id_c, \vec{I}, \vec{O}, fs, \vec{I}_{sm}, \vec{O}_{sm}, sm)$ , containing all the information related to the safety monitor as described above.

## 5 Safety Reasoning using DLV

One of the main goals of safety engineers during the definition of a system architecture is to place suitable safety patterns so that the identified hazards can be controlled. This section demonstrates how much of this safety reasoning can be automated.

To this end, we introduce two new facts used to denote when a hazard is controlled or not:

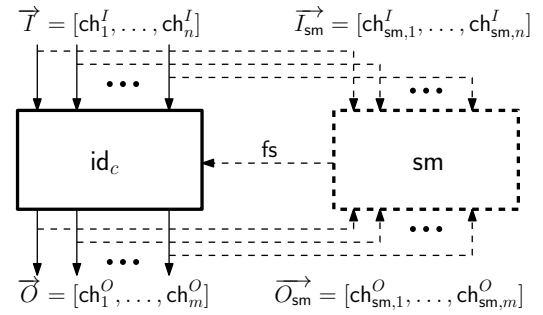


Figure 3: Safety Monitor Pattern

| <b>Safety Architectural Patterns</b>  |  |
|---|--|
| <b>Fact</b>   | <b>Denotation</b>  |
| HDR ( $id, id_c, I_c, id_{c'}, I_{vt_1}, I_{vt_2}, vt, vt_{out}, id_{out}$ )  | $id$ is a duplex redundancy associated with the function $id_c$ . $I_c$ is a channel from $id_c$ that might convey a fault message. $id_{c'}$ is a function possibly $id_c$ . $vt$ is a voter that receives data from $id_c$ and $id_{c'}$ through channels $I_{vt_1}$ , and $I_{vt_2}$ , respectively. The result from $vt$ is sent to $id_{out}$ through channel $vt_{out}$ .  |
| TMR ( $id, id_c, I_c, id_{c'}, id_{c''}, I_{vt_1}, I_{vt_2}, I_{vt_3}, vt, vt_{out}, id_{out}$ )                                  | $id$ is a triple modular redundancy associated with the function $id_c$ . $I_c$ is a channel from $id_c$ that might convey a fault message. $id_{c'}$ and $id_{c''}$ are functions possibly $id_c$ . $vt$ is a voter that receives data from $id_c$ , $id_{c'}$ and $id_{c''}$ through channels $I_{vt_1}$ , $I_{vt_2}$ , and $I_{vt_3}$ , respectively. The result from $vt$ is sent to $id_{out}$ through channel $vt_{out}$ .   |
| 2Prog ( $id, id_c, \vec{I}_{id_c}, \vec{O}_{id_c}, id_{c'}, \vec{I}_{vt_1}, \vec{I}_{vt_2}, \vec{VT}, \vec{VT}_{out}, id_{out}$ ) | $id$ is a 2-version programming associated with the function $id_c$ (a.k.a. version 1). $id_{c'}$ (a.k.a. version 2) is an identical function of $id_c$ . The inputs to $id_c$ and the outputs from $id_c$ are sent through channels $\vec{I}_{id_c}$ and $\vec{O}_{id_c}$ , respectively. $\vec{VT}$ is a list of voters that receive data from $id_c$ and $id_{c'}$ through channels $\vec{I}_{vt_1}$ and $\vec{I}_{vt_2}$ , respectively. The results from $\vec{VT}$ are sent to their respective functions $\vec{VT}_{out}$ through channels $id_{out}$ . |
| safMon( $id, id_c, \vec{I}, \vec{O}, fs, \vec{I}_{sm}, \vec{O}_{sm}, sm$ )  | $id$ is a safety monitor associated with the function $id_c$ . It uses the list of input and output channels $\vec{I}$ and $\vec{O}$ , respectively. The data of these channels are sent as input to $sm$ through the list of channels $\vec{I}_{sm}$ and $\vec{O}_{sm}$ . $fs$ is a channel from $sm$ to $id_c$ which sends a fail-safe signal whenever some inconsistency is detected.   |
| watchDog( $id, id_c, fs, I_{wd}, wd$ )  | $id$ is a watchdog associated with the function $id_c$ . It receives liveness messages from $id_c$ through channel $I_{wd}$ . $fs$ is a channel from $wd$ to $id_c$ which sends a fail-safe signal whenever some inconsistency w.r.t the expected messages is detected.  |

Table 2: SafPat: Language for Safety Architectural Patterns.

- $\text{ctl}(\text{id}_H, \text{id}_c, \text{tp}, \text{sv})$  and  $\text{nctl}(\text{id}_H, \text{id}_c, \text{tp}, \text{sv})$  denote that the hazard  $\text{id}_H$  of type  $\text{tp}$ , severity  $\text{sv}$  and associated with the function  $\text{id}_c$  can be, respectively, controlled and not controlled.

Before we specify controlled and not controlled hazards, we need to distinguish two types of hazards: *basic* hazards and *derived* hazards. A hazard is classified as *basic* if it does not have any sub-hazards, and *derived* otherwise. The following DLV rules specify this:

```
basic(H,CP,TP,SV) :- hz(H,CP,TP,SV), not has_subHz(H).
has_subHz(H) :- subHz(SH,H).
derived(H,CP,TP,SV) :- hz(H,CP,TP,SV), has_subHz(H).
```

We now use the closed-world semantics of DLV to specify controllability. A basic hazard is not controlled if there is no rule explicitly supporting its controllability, as specified by the rule:

```
nctl(H,CP,TP,SV) :- basic(H,CP,TP,SV), not ctl(H,CP,TP,SV) .
```

A derived hazard is not controlled if any one of its sub-hazards is not controlled as specified by the following rules:

```
nctl(H,CP,TP,SV) :- hz(H,CP,TP,SV), derived(H,CP,TP,SV),
                    hasNCTLSubHz(H,CP,TP,SV).
hasNCTLSubHz(H,CP,TP,SV) :- hz(H,CP,TP,SV), subHz(SH,H),
                             nctl(SH,SCP,STP,SSV).
```

**Example 2** Consider the hazards and sub-hazards relations in Example 1. The hazards  $\text{hz}(\text{h1}, \text{acc}, \text{err}, \text{cat})$  can be controlled if its three sub-hazards,  $\text{h11}$ ,  $\text{h12}$  and  $\text{h13}$ , can be controlled.

Safety patterns are commonly used to control hazards by, *e.g.*, adding redundancy to the system. Given our language SafPat, the reasoning principles used to do so can be easily captured by DLV rules. We list some reasoning principles for some of the patterns:

**WatchDog Pattern.** The following rule specifies that watch dog can be used to control hazard of type loss of function (*loss*).

```
ctl(ID,CP,loss,SV) :- hz(ID,CP,loss,SV), watchDog(_,CP,_,_,_).
```

**Safety Monitor Pattern.** The following rules specify intuitively that a hazard associated to a function CP of type erroneous can be controlled if a safety monitor is associated to CP provided not  $\text{inpNotCovSF}(\text{ID2})$  and not  $\text{outNotCovSF}(\text{ID2})$ : there are no input logical channels, *i.e.*, channels incoming to CP specified by  $\text{ch}(\text{CH}, \_, \text{CP})$ , not taken as input to the safety monitor, nor output channels *i.e.*, channels outgoing from CP specified by  $\text{ch}(\text{CH}, \text{CP}, \_)$ . The predicate  $\text{\#member}$ , *e.g.*,  $\text{\#member}(\text{CH}, \text{ICHs})$  specifies that CH is a member of list ICHs. You can safely ignore the fact *isexploration* which is only used for the automation as described in Section 6.

```
ctl(ID,CP,err,SV) :- hz(ID,CP,err,SV), safMon(ID2,CP,_,_,_,_,_),
                    not inpNotCovSF(ID2), not outNotCovSF(ID2).
inpNotCovSF(ID2) :- safMon(ID2,CP,ICHs,_,FS,_,_,_), ch(CH,_,CP),
                    CH != FS, not \#member(CH,ICHs), not isexploration.
outNotCovSF(ID2) :- safMon(ID2,CP,_,OCHs,_,_,_,_), ch(CH,CP,_,_),
                    not \#member(CH,OCHs), not \#member(CH,MIN),
                    not \#member(CH,MOUT), not isexploration.
```

**2-version programming.** This pattern is used to improve safety by adding software redundancy. Hence, it can only be associated with functions implemented as software as specified by the rule:

```
ctl(ID,CP,err,SV) :- hz(ID,CP,err,SV), 2Prog(ID2,CP,_,_,_,_,_,_),
    sw(CP), not inpNotCovNP(ID2).
```

Here `inpNotCovNP` is similar to `inpNotCovSF` explained above.

**HDR.** The HDR and TMR Voter patterns can be used for two different safety reasons: (1) to improve safety by hardware redundancy or (2) to improve safety by path redundancy. These are specified by the following rules, where omission is a type of error:

```
ctl(ID,CP,err,SV) :- hz(ID,CP,err,SV),
    hdr(ID3,_,_,_,_,_,VOTERCP,_,_), ch(_,CP,VOTERCP).
ctl(ID,CP,omission,SV) :- hz(ID,CP,omission,SV), cp(CP), cp(CP1), cp(CP2),
    ch(CHOUT,CP1,_), ch(CHIN,_,CP2), ch(CH,CP,_), if(IF,PATH),
    before(CH,CHIN,IF), before(CHOUT,CH,IF),
    hdr(IDPAT,CP1,_,CP2,_,_,_,_,_).
```

The second rule requires further explanation. The fact `before(CH,CHIN,IF)` specifies that CH appears before CHIN in the path PATH. The rule itself specifies that if there is an IF such that there is a hazard of type omission associated to a component CP in the information path PATH, then placing a HDR on a functions CP1 and CP2 before and after CP in the path can control an omission hazard. Intuitively, this is because Voters placed in this way can detect when safety critical messages are lost during transmission due to the omission of CP.

**Remark:** This paper specifically focuses on architectural principles. We focus on the architecture components and how such components interact with others through channels. Encoding other reasoning principles like, *e.g.*, the actual behavior of such components, are left to future work.

## 6 Automated Pattern Recommendation

This section builds on the principles specified to automate the recommendation of safety patterns. Our machinery enables a safety engineer to understand which options of patterns he can use to control hazards and decide which one is more suitable given factors, such as costs and hardware availability.

The recommendation machinery uses ASP/DLV semantics to enumerate design options by attempting to place safety patterns wherever they are applicable. In this way, each answer of our DLV specification corresponds to a recommended architecture. Some recommended architectures may be better than others, *e.g.*, controlling more hazards. From all obtained answers, the system can recommend to the safety engineer only the best architectures, *i.e.*, the ones that control the most number of hazards.

The recommendation system is activated by using facts of the form.

- `explore(N,Pat)` denoting that the system shall recommend the placement of at most N patterns of type Pat, where Pat is one of patterns described in Table 2.

For example, if `explore(1,safMon)`, the system attempts to add at most one additional safety monitor to a given architecture. Multiple such facts can be used to recommend different patterns at the same time. As a result, safety engineers can configure the pattern recommendation machinery to search for particular safety patterns that can control identified hazards.

We have implemented rules for recommending the patterns shown in Table 2. Due to space restrictions, we describe only some of them used for recommending safMon, TMR, and HDR.

The following DLV rule specifies the enumeration of placement or not of a safMon(nsafMon), associated with the function CP that is furthermore associated with a basic or not controlled hazard ID:

```
safMon(nuSafMon, CP, allInputs, allOutputs, nuSC, numin, numout, numcp) v
nsafMon(nuSafMon, CP, allInputs, allOutputs, nuSC, numin, numout, numcp)
:- cp(CP), hz(ID, CP, err, SV), basicOrNCTL(ID, CP, err, SV), explore(N, safMon).
```

We assume here that the constants starting with nu are fresh, *i.e.*, do not appear in the given architecture, thus used only for recommended safety patterns. Since it is enough to know to which function a safety monitor is associated to, we do not need to enumerate all the inputs and outputs of CP, but rather simply denote CP's inputs and outputs using, respectively, the fresh constants allInputs and allOutputs.

The rule above will attempt to place a safety monitor in any applicable location of the architecture. The following clause limits the number of safety monitors that can be recommended to be at most N. Here #count is a DLV aggregate predicate returning the size of a symbolic set defined by its argument.

```
:- #count{CP : safMon(nuSafMon, CP, _, _, _, _, _)} > N, explore(N, safMon).
```

Notice that whenever a pattern is recommended, the controllability reasoning described in Section 5 applies to infer which hazards are controlled by this pattern and which are not.

The reasoning principles described in Section 5 can be used to further constraint the number of recommendations. For example, a TMR used for hardware redundancy shall only be associated with components that are not software components as specified by the following rule:

```
tmr(nuTMR, CP1, CH1, nucp2, nucp3, nuchm1, nuchm2, nuchm3, nuvtcp, nucho, nucpo) v
ntmr(nuTMR, CP1, CH1, nucp2, nucp3, nuchm1, nuchm2, nuchm3, nuvtcp, nucho, nucpo)
:- cp(CP1), not sw(CP1), hz(HZ0, CP1, err, SV), ch(CH1, CP1, _), explore(N, tmr).
```

The next example illustrates the power of our language to specify pattern recommendation. It specifies conditions for recommending HDR patterns to achieve path redundancy.

```
hdr(nuHDR, CP1, CH1, CP2, nuchm1, nuchm2, nuvtcp, nucho, CP0)
v nhdr(nuHDR, CP1, CH1, CP2, nuchm1, nuchm2, nuvtcp, nucho, CP0)
:- hz(ID, CP, omission, SV), cp(CP), cp(CP1), cp(CP2), CP1 != CP,
    CP1 != CP2, CP1 != CP0, CP2 != CP0, ch(CHOUT, CP1, _), ch(CHIN, _, CP2),
    ch(CH, CP, _), ch(CH1, _, CP0), if(IF, PATH), before(CHOUT, CHIN, IF),
    before(CHOUT, CH, IF), before(CHIN, CH1, IF), explore(N, hdr).
```

We search for functions CP0, CP1 and CP2 and a channel CH1 where to place the HDR. The goal is to control a hazard associated with function CP by increasing path redundancy. To this end, CP1 needs to appear before CP in an information flow PATH that uses these functions. CP2 may either be equal to CP or located after CP in such a PATH. Thus, HDR can, in principle, detect when messages are omitted by CP. Whenever this happens, the HDR shall send a message to the function CP0 used only later in the information flow PATH.

A constraint similar to the one for safMon, constraints the number of TMR and HDR to be searched for. These constraints are omitted here.

## 7 Case Studies

This section illustrates the results of our automated safety reasoning for two case studies, namely Adaptive Cruise Control (ACC) and Battery Management System (BMS). We illustrate our results by depicting how the architectures of both ACC and BMS would appear on a layout when our machinery is used. The safety patterns suggested by our machinery are depicted as dark gray boxes, and the channels related (inputs or outputs) to such patterns are depicted as dashed arrows.

**Adaptive Cruise Control (ACC).** We identified an erroneous ( $\mathbf{H1}_{acc}$ ) and a loss ( $\mathbf{H2}_{acc}$ ) hazard on ACC, as described in Section 2. The erroneous hazard ( $\mathbf{H1}_{acc}$ ) is broken down into three sub-hazards, namely erroneous DS ( $\mathbf{H1.1}_{acc}$ ), erroneous VS ( $\mathbf{H1.2}_{acc}$ ), and erroneous ACCM ( $\mathbf{H1.3}_{acc}$ ).

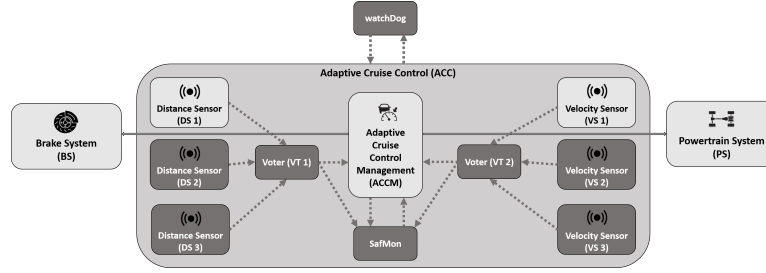


Figure 4: ACC Functional Architecture with safMon, TMR and WD

We run our recommendation machinery to automatically identify what safety patterns could be used to control the identified hazards. Our machinery yielded *five complete solutions* (*i.e.*, architectures) for controlling these hazards. For the sake of space, we only show one of those solutions. The architecture of the chosen solution is depicted in Figure 4. The subset of our DLV specification for this solution is shown below. It contains the predicates for the recommended safety patterns and controllability.

```
{safMon(nuSafMon, accm, allInputs, allOutputs, nuSC, numin, numout, numcp),
 tmr(nuTMR, ds, dsaccm, nucp2, nucp3, nuchm1, nuchm2, nuchm3, nuvtcp, nucho, nucpo),
 tmr(nuTMR, vs, vsaccm, nucp2, nucp3, nuchm1, nuchm2, nuchm3, nuvtcp, nucho, nucpo),
 watchDog(nuWD, acc, nuswd, nulvwd, nuwd), ct1(["hz", accLs], acc, loss, cat),
 ct1(["hz", ds], ds, err, cat), ct1(["hz", vs], vs, err, cat),
 ct1(["hz", accm], accm, err, cat), ct1(["hz", accEr], acc, err, cat)}
```

Our machinery recommended to use three safety patterns, *i.e.*, safMon, TMR, and watchDog, to control the identified hazards. The main difference w.r.t. the other solutions (omitted here) is 2Prog instead of safMon. To control the sub-hazards  $\mathbf{H1.1}_{acc}$  and  $\mathbf{H1.2}_{acc}$ , our machinery recommended to use TMR on DS and VS, respectively. The goal is to improve safety by hardware (*i.e.*, DS and VS) redundancy. The remaining sub-hazard  $\mathbf{H1.3}_{acc}$  can be controlled by placing a safMon on ACCM. The hazard  $\mathbf{H1}_{acc}$  is then controlled by using both TMR and safMon. Finally, our machinery recommended to use a watchDog on ACC to control the loss hazard  $\mathbf{H2}_{acc}$ .

**Battery Management System (BMS).** We identified an erroneous ( $\mathbf{H1}_{bms}$ ) hazard on CI, as described in Section 2. This erroneous hazard ( $\mathbf{H1}_{bms}$ ) is broken down into three sub-hazards, namely erroneous BMS ( $\mathbf{H1.1}_{bms}$ ), erroneous CAN ( $\mathbf{H1.2}_{bms}$ ), and omission FW ( $\mathbf{H1.3}_{bms}$ ). Typically, hazards on CAN buses can be controlled by replacement only. Hence, we assume that  $\mathbf{H1.2}_{bms}$  has already been controlled.

Our recommendation machinery yielded *four complete solutions* (i.e., architectures) to control  $\mathbf{H1}_{bms}$ ,  $\mathbf{H1.1}_{bms}$ , and  $\mathbf{H1.3}_{bms}$ . For the sake of space, we only show two of those solutions. The architecture of the chosen solutions are depicted in Figure 4. The DLV specification for those solutions is similar to the one presented in the ACC case study.

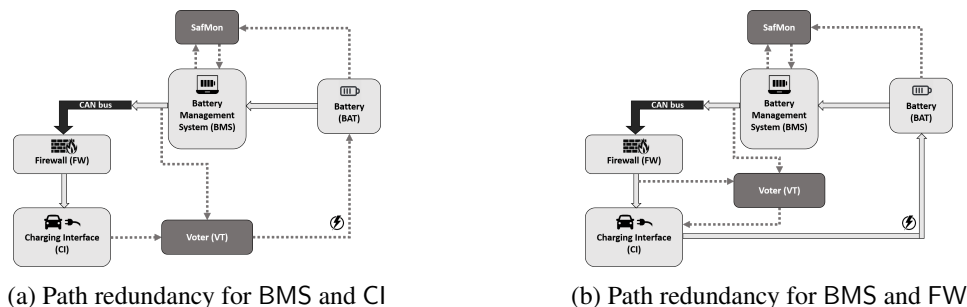


Figure 5: Battery Management System Functional Architecture with safMon and HDR

Our machinery recommended to use two safety patterns, i.e., safMon and HDR, to control the identified hazards. On both solutions, a safMon is placed together with BMS to control  $\mathbf{H1.1}_{bms}$ . For the ACC example, TMR is placed to improve safety by hardware redundancy. Here, HDR is placed to improve safety by path redundancy. The HDR solutions are depicted in Figures 5a and 5b control  $\mathbf{H1.3}_{bms}$ . They differ w.r.t which functions are composing the HDR. Figure 5a illustrates that BMS and CI sent redundant inputs to vt so that BAT has a higher chance of getting the expected input. That is, if CI does not send the input to BAT due to, e.g., an omission from FW, BAT receives the expected input from BMS through vt. Similarly, Figure 5b illustrates that BMS and FW sent redundant inputs to vt with CI as destination. Consequently, BAT should have a higher chance of getting the expected input from CI.

## 8 Related Work

**Failure Rates Computations.** An important analysis for safety is the computation of failure rates of the system and its sub-systems as it is a requirement for safety-critical systems to have (very) low failure rates. The automation of this computation has been subject of some previous work [15, 2]. In particular, for a given architecture and given sub-system fault rates, the failure rate of the system is computed. Our work on reasoning with safety patterns complements the work above as we consider the design of the architecture itself, which is part of the input used by the work above.

**Safety Case Models.** GSN [7] is a model for specifying safety cases. Safety cases are tree-like structures containing different types of nodes denoting, e.g., Goals, Strategies, Contexts, Assumptions of a safety case. As the exact meaning of each node is specified textually (inside the node), models written in GSN enables little automation. There are, however, work that provide more structure to GSN models and others providing means for some automation [6]. We describe some approaches below. [13] proposes patterns encoding typical safety reasoning principles, such as those using FTA, FMEA, STPA. While these reasoning patterns provide some structure to GSN models, they suffer from the same automation limitations of GSN described above. On the one hand, our work complements this work by specifying reasoning principles based on safety patterns, which was not considered in [13]. On the other hand, we believe that it is possible to encode some of the reasoning principles described in [13] and consider not only safety reasoning with patterns but the other types of reasoning described in [13]. [9, 10] propose

automated quantitative evaluation methods for GSN models that associated to Goal nodes with values for belief, disbelief and uncertainty. It is not clear from this work how these values are related to the quality of safety argument. We believe that the encoding of our reasoning principles can profit from this work to make the relation between the quality of the safety argument and the belief values more explicit.

**Safety Reasoning using Logic Programming.** Logic programming has been used in the past for safety reasoning. For example, [14] provides decision support for air traffic control systems by specifying landing criteria in complex landing situations by using Defeasible Logic Programming (DeLP). [26] outlines a method for safety assessment of medical devices also based DeLP. An interesting work is presented in [3] on the formalization of automotive standard requirements [18] to enable automatic reasoning about compliance with the standard. We take a similar approach to these works as we also use logic programming and engine to support safety engineers in the designing system architecture. However, we do not consider here reasoning with uncertain and incomplete knowledge as in the work above using DeLP. As described above, we are considering extending the type of safety reasoning encoded to also include uncertainty [9, 10]. DeLP is a method we could consider for modeling such arguments.

## 9 Conclusion

This paper establishes the first steps towards automated safety (and security) for embedded systems. We propose a domain-specific language, called SafPat, for safety reasoning on the architectural level using safety patterns. We encode typical safety reasoning principles as disjunctive logic programs, using these specification for increasing automated reasoning, namely, on determining controllability and recommending patterns.

We are currently investigating a number of future directions. We are considering other types of safety reasoning, *e.g.*, reasoning with uncertainty. Further, as illustrated by the BMS case study, there are a number of co-analysis reasoning deriving from the use safety and security patterns. It seems possible to build on the grounds established by this paper to carry out such reasoning in an automated fashion.

The increased automation provided by our methods seems to support incremental methods for safety (and in the future security). It is possible to identify, *e.g.*, which hazards are no longer controlled whenever there is an incremental change to the system. We are currently investigating how to improve the proposed automated reasoning for this purpose. Finally, we plan to integrate our machinery into the Model-Based Engineering Tool AutoFOCUS3 [1]. The goal is to enable safety engineers to use our automated reasoning with models written in AutoFOCUS3. This will also enable the use of automated methods for building safety cases modeled in GSN [6].

**Acknowledgment.** This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 830892. Nigam is partially supported by CNPq grant 303909/2018-8.

## References

- [1] *AF3 – AutoFOCUS 3*. Available at <https://af3.fortiss.org/>.
- [2] *Fault Tree Analysis – FT +*. Available at <https://tinyurl.com/faulttreetan>.
- [3] J. P. C. Ardila, B. Gallina & G. Governatori (2018): *Lessons Learned while Formalizing ISO 26262 for Compliance Checking*. In: *2nd Workshop on TeReCom - Tech. for Regulatory Compliance*, pp. 5–16.



- [4] C. Baral (2010): *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.
- [5] C. Kreiner, C. Preschern, N. Kajtazovic (2013): *Security Analysis of Safety Patterns*. In: *20th Conference on Pattern Languages of Programs, PLoP '13, USA*, pp. 12:1–12:38.
- [6] C. Cârlan, V. Nigam, A. Tsalidis & S. Voss (2019): *ExplicitCase: Tool-support for Creating and Maintaining Assurance Arguments Integrated with System Models*. In: *WoSoCer*, doi:10.1109/ISSREW.2019.00093.
- [7] GSN Community (2011): *GSN Community Standard Version 1*. Available at [http://www.goalstructuringnotation.info/documents/GSN\\_Standard.pdf](http://www.goalstructuringnotation.info/documents/GSN_Standard.pdf).
- [8] F. Crawley & B. Tyler, editors (2015): *HAZOP: Guide to Best Practice*.
- [9] L. Duan, S. Rayadurgam, M. P. E. Heimdahl, A. Ayoub, O. Sokolsky & I. Lee (2014): *Reasoning About Confidence and Uncertainty in Assurance Cases: A Survey*. In: *FHIES*, 9062, Springer, pp. 64–80, doi:10.1007/978-3-319-63194-3\_5.
- [10] J. Dürrwang, K. Beckers & R. Kriesten (2017): *A Lightweight Threat Analysis Approach Intertwining Safety and Security for the Automotive Domain*. In: *SAFECOMP*, doi:10.1007/978-3-319-66266-4\_20.
- [11] T. Eiter, G. Gottlob & H. Mannila (1997): *Disjunctive Datalog*. *ACM Trans. Database Syst.* 22(3), doi:10.1145/116825.116838.
- [12] M. Gelfond & V. Lifschitz (1990): *Logic Programs with Classical Negation*. In: *ICLP*, pp. 579–597.
- [13] M. Gleirscher & C. Cârlan (2017): *Arguing from Hazard Analysis in Safety Cases: A Modular Argument Pattern*. In: *HASE*, pp. 53–60, doi:10.1109/HASE.2017.15.
- [14] S. A. Gómez, A. Goron & A. Groza (2014): *Assuring Safety in an Air Traffic Control System with Defeasible Logic Programming*. In: *15th Argentine Symposium on Artificial Intelligence, ASAI*.
- [15] P. Helle (2012): *Automatic SysML-Based Safety Analysis*. In: *ACES-MB*, p. 1924, doi:10.1145/2432631.2432635.
- [16] SAE International (1996): *Standard ARP 4761: Guidelines and Methods for Conducting the Safety Assessment*. Available at <https://www.sae.org/standards/content/arp4761/>.
- [17] SAE International (2011): *ARP 4754a: Guidelines for Development of Civil Aircraft and Systems*. Available at <https://www.sae.org/standards/content/arp4754a/>.
- [18] ISO (2011): *ISO 26262, Road vehicles Functional safety - Part 6: Product Development: Software Level*. Available at <https://www.iso.org/standard/43464.html>.
- [19] A. Kondeva, C. Carlan, H. Ruess & V. Nigam (2019): *On Computer-Aided Techniques for Supporting Safety and Security Co-Engineering*. In: *WoSoCer*, doi:10.1109/ISSREW.2019.00095.
- [20] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri & F. Scarcello (2006): *The DLV System for Knowledge Representation and Reasoning*. *ACM Trans. Comput. Logic* 7, pp. 499–562, doi:10.1145/1149114.1149117.
- [21] N. Leveson & J. Thomas (2018): *STPA Handbook*.
- [22] H. Martin, Z. Ma, Ch. Schmittner, B. Winkler, M. Krammer, D. Schneider, T. Amorim, G. Macher & Ch. Kreiner (2020): *Combined automotive safety and security pattern engineering approach*. *Reliability Engineering & System Safety* 198(2), pp. 1–35, doi:10.4018/jrse.2012040101.
- [23] H. L. V. De Matos, A. M. da Cunha & L. A. V. Dias (2014): *Using Design Patterns for Safety Assessment of Integrated Modular Avionics*. In: *DASC*, doi:10.1109/DASC.2014.6979473.
- [24] Defence UK Ministry (2007): *Safety Management Requirements for Defence Systems*. Available at <https://www.skybrary.aero/bookshelf/books/344.pdf>.
- [25] V. Nigam, A. Pretschner & H. Ruess (2018): *Model-Based Safety and Security Engineering*. Available at <https://arxiv.org/abs/1810.04866>. White Paper.
- [26] Gomez S.A., Groza A. & Chesnevar C.I. (2014): *An Argumentative Approach to Assessing Safety in Medical Device Software Using Defeasible Logic Programming*. In: *Meditech*.

# Chapter 5

## Integrating Safety Architecture Patterns into MBSE: A Plugin for Automated Pattern Synthesis

Chapter 5 focuses on the integration of the developed logic programming tool [39, 45], designed for reasoning about safety architecture patterns, into a model-based systems engineering (MBSE) framework. Within this chapter, a plugin for safety architecture pattern synthesis within the MBSE context is proposed. The validation of this plugin has been conducted using an example taken from the automotive domain.

### Contributing articles:

- Yuri Gil Dantas, Tiziano Munaro, Carmen Cârlan, Vivek Nigam, Simon Barner, Shiqing Fan, Alexander Pretschner, Ulrich Schöpp, and Sergey Tverdyshev: A Model-based System Engineering Plugin for Safety Architecture Pattern Synthesis. *MODELSWARD 2022*: 36-47
- Yuri Gil Dantas, Tiziano Munaro, Carmen Cârlan, Vivek Nigam, Simon Barner, Shiqing Fan, Alexander Pretschner, Ulrich Schöpp, and Sergey Tverdyshev: A Toolchain for Synthesizing and Validating Safety Architectures. *SN Computer Science 2023*: Volume 4, Article number: 335.

### Copyright information:

- Article [40]: Science and Technology Publications Lda (SCITEPRESS), 2024. <https://doi.org/10.5220/0010831700003119>.
- The article [41] is reproduced with permission from Springer Nature. License number: 5798741234318. Print rights of the Version of Record are provided for; electronic rights for use only on institutional repository as defined by the Sherpa guideline ([www.sherpa.ac.uk/romeo/](http://www.sherpa.ac.uk/romeo/)) and only up to what is required by the awarding institution.

**Author contributions:** The idea of developing a Model-Based Systems Engineering (MBSE) plugin for safety architecture pattern synthesis was brought up by Yuri Gil Dantas and Vivek Nigam. They had several discussions on how the interfaces with the backend (previously developed by Yuri Gil Dantas and Vivek Nigam [39]) should be designed. Yuri Gil Dantas implemented the MBSE plugin after several discussions with MBSE researchers, including Tiziano Munaro and Simon Barner. Yuri Gil Dantas took the lead in writing the initial draft of the article. The co-authors assisted Yuri Gil Dantas in improving the article. Yuri Gil Dantas handled subsequent revisions and corrections. Yuri Gil Dantas won the best student paper award with this article [40].

The second article [41] is the result of a collaboration with MBSE researchers, in particular, Tiziano Munaro and Simon Barner. This article is a successor of article [40], where the MBSE plugin has been improved. Yuri Gil Dantas prepared the structure of the article. Yuri Gil Dantas took the lead in organizing the structure of the article and made substantial contributions to various sections, excluding Section 5 (Feature Degradation Synthesis) and Section 6 (Simulation) written by Simon Barner and Tiziano Munaro, respectively. The other co-authors assisted Yuri Gil Dantas in improving the article.

# A Model-based System Engineering Plugin for Safety Architecture Pattern Synthesis

Yuri Gil Dantas<sup>1</sup>, Tiziano Munaro<sup>1</sup>, Carmen Carlan<sup>1</sup>, Vivek Nigam<sup>2</sup>, Simon Barner<sup>1</sup>, Shiqing Fan<sup>2</sup>, Alexander Pretschner<sup>1,3</sup>, Ulrich Schöpp<sup>1</sup> and Sergey Tverdyshev<sup>2</sup>

<sup>1</sup>fortiss GmbH, Munich, Germany

<sup>2</sup>Huawei Technologies Düsseldorf GmbH, Düsseldorf, Germany

<sup>3</sup>Technische Universität München, Munich, Germany

**Keywords:** Model-based System Engineering, Safety Architecture Patterns, Automation, Tooling.

**Abstract:** Safety architecture patterns are abstract representations to address faults in the system architecture. In the current state of practice, the decision of which safety architecture pattern to deploy and where in the system architecture is carried out manually by a safety expert. This decision may be time consuming or even lead to human errors. This paper presents Safety Pattern Synthesis, a tool for automating the recommendation of safety architecture patterns during the design of safety-critical systems: 1) Safety Pattern Synthesis recommends patterns to address faults in the system architecture (possibly resulting in more than one architectural solution), 2) the user selects the system architecture with patterns based on, e.g., the criteria provided by Safety Pattern Synthesis, and 3) Safety Pattern Synthesis provides certain requirements that shall be considered in the overall safety engineering process. The proposed tool has been implemented as a plugin in the model-based system engineering tool called AutoFOCUS3. Safety Pattern Synthesis is implemented in Java while using a logic-programming engine as a backend to reason about the safety of the system architecture. This paper provides implementation details about Safety Pattern Synthesis and its applicability in an industrial case study taken from the automotive domain.

## 1 INTRODUCTION

Safety architecture patterns, such as the Homogeneous Duplex or the Triple Modular Redundancy patterns, are deployed to avoid harm due to faults triggering failures, such as erroneous function or loss of function (Avizienis et al., 2004). An advantage of making use of such patterns in practice is that their goal and development are well understood, and even recommended by standards (ISO26262, 2018)(IEC61508, 2010).

In the current state of practice, the decision of which safety architecture pattern to deploy in a given system architecture is done with limited computer-aided support. Currently, an expert (e.g., a safety engineer) determines which pattern to deploy and where by carrying out a manual safety analysis. As the complexity of systems grows, it becomes challenging for experts to make these decisions due to, e.g., time consuming and human error issues. Moreover, since these decisions are normally made in early stages of design, potential errors or sub-optimal designs may

result in high development delays and costs.

Current safety-critical systems are characterized by an ever-increasing number of highly interdependent requirements, functions and subsystems. The safe integration of critical and non-critical components onto a shared execution platform is very important to enable the certification of such systems. While integration platforms such as time-space partitioning hypervisors or dedicated hardware units that support the segregation of critical tasks are available today, such constitute a large configuration space that adds additional complexity to determine an architecture configuration that satisfies all constraints (in particular in terms of safety and performance).

The inherent abstraction introduced by Model-Based System Engineering (MBSE) has the potential to meet these challenges, as it has been shown in e.g., an model based engineering approach for mixed-criticality systems (Barner et al., 2017), the model-based architecture exploration approach introduced by Eder *et al.* (Eder et al., 2018b)(Eder et al., 2018a)(Eder et al., 2020), or in the approaches by

(Amorim et al., 2017)(Martin et al., 2020) that use MBSE to address the complexity of safety architecture design using architecture patterns. A key limitation of the existing approaches for safety architecture patterns is that it does not provide the type of automation successfully applied in other development phases, such as in optimizing deployment strategies (Eder et al., 2018b).

*Our goal is to provide safety engineers with computer-aided support for selecting safety architecture patterns in an automated fashion.* To this end, we have developed a plugin within the MBSE tool AutoFOCUS3 (Aravantinos et al., 2015) to enable the model-driven approach using safety architecture patterns. We refer to the developed plugin as Safety Pattern Synthesis. Safety Pattern Synthesis relies on MBSE practices to automatically recommend safety architecture patterns for tolerating faults in the system architecture. The intended outcome of Safety Pattern Synthesis is to reduce the effort from safety engineers during a safety analysis, in particular by assisting with the selection of patterns to ensure the required functional safety of the system architecture. To the best of our knowledge, Safety Pattern Synthesis is the first MBSE tool that enables the automated recommendation of safety architecture patterns.

The development of Safety Pattern Synthesis has been motivated by concrete use-cases provided by our industry partner. In this paper, we illustrate how Safety Pattern Synthesis has been used for the development of safe Highway Pilot (HWP) features, such as Adaptive Cruise Control (ACC) and Emergency Brake (EB) functions.

The remainder of this paper is structured as follows: Section 2 presents some background information to help the reader to understand the results presented in the paper. Section 3 describes Safety Pattern Synthesis, including its architecture and implementation details. Safety Pattern Synthesis is validated in Section 4 using an industry use-case. Finally, we conclude the paper by pointing out to related and future work in Sections 5 and 6.

## 2 BACKGROUND

### 2.1 Model-based Engineering in AutoFOCUS3

AutoFOCUS3 (AF3) is a model-based open source tool and research platform for safety-critical embedded systems (fortiss GmbH, 2020). AF3 builds on the Eclipse platform and supports the design, devel-

opment and validation of safety-critical embedded systems in many development phases, including architecture design, implementation, and hardware and software integration.

The tool’s metamodel (Aravantinos et al., 2015)(Barner et al., 2018) provides multiple viewpoints to describe the different aspects of the system under design. The *logical architecture* represents an implementation-agnostic specification of the system’s behaviour. The *technical viewpoint* includes a series of models. The *task and partition architectures* represent the hardware-independent interaction of software tasks and their aggregation into partitions, and the *platform architecture* describes the system’s hardware including its properties and topology. Finally, the distinct models are linked by means of allocations, defining, *e.g.*, the deployment of tasks to hardware units.

### 2.2 Safety Concepts

We briefly review some basic safety concepts to set the terminology used in the remainder of the paper. The definition of the safety concepts described below are mainly taken from (Avizienis et al., 2004).

A *hazard* is a situation that can cause harm to users or businesses. A *failure* is an event that when occurs results in a deviation of the expected behavior of a function. An *error* is a deviation of the expected system behavior. A *fault* is the hypothesized cause of an error. A failure triggered by a fault may lead to a hazard. Normally, failures are associated with a set of predefined Guidewords that characterize intuitively the semantics of such failures. Examples of Guidewords are *loss* and *erroneous* that denote, respectively, a failure due to the loss of a function, *i.e.*, a function not operating whatsoever, and a failure due to an erroneous function behavior, *e.g.*, a function not computing correctly some output value. This paper refers to hazards, faults, and failures as *safety elements*. A *component* is a part of a system that implements a function and consists of software units or hardware parts. Components may be assigned to an Automotive Safety Integrity Level (ASIL), *i.e.*, the level of safety assurance required ranging among QM, A, B, C, D, where D is the highest assurance level.

A *safety architecture pattern* (safety pattern for short) is an architectural solution for tolerating faults in the system architecture. A *fault detection pattern* deactivates the system in the presence of a failure (triggered by a fault) by either transitioning the system to a safe state (*e.g.*, informing the driver to take over the vehicle control) or shutting down the system. A *fault tolerant pattern* ensures that the system will

continue to operate in the presence of a failure by providing a redundant component to take over. Fault tolerant patterns improve the availability of the system given the redundant component. A fault tolerant pattern may also transition the system to a safe state or shutdown the system in the presence of a failure in the redundant component.

Examples of safety patterns are Homogeneous Duplex, Heterogeneous Duplex, Triple Modular Redundancy, Simplex Architecture, Acceptance Voting, and Monitor Actuator (Armoush, 2010)(Preschern et al., 2013b)(Biondi et al., 2020)(Bak et al., 2009).

*Homogeneous Duplex pattern* is a fault tolerant pattern that addresses hardware faults by duplicating the primary hardware component. Similarly, *Heterogeneous Duplex pattern* is a fault tolerant pattern that also addresses hardware faults by duplicating the primary hardware component. However, the primary and the redundant components shall be designed and implemented independently from each other. Heterogeneous Duplex pattern may also address software faults as long as the software running in redundant component is implemented using a different design. *Triple Modular Redundancy pattern* is a fault tolerant pattern that addresses hardware faults by tripling the primary hardware component. *Simplex Architecture pattern* is a fault tolerant pattern that addresses software faults by providing a simple and reliable version of the primary component. *Acceptance Voting pattern* is a fault tolerant pattern that addresses software faults by providing diverse redundancy implementations of the primary software component. *Monitor Actuator pattern* is a fault detection pattern mainly known for addressing hardware faults, but it can also address software faults through the use of plausibility checks.

### 2.3 SafPat (Backend)

A framework called SAFPAT (Dantas et al., 2020) has been recently proposed for automating the recommendation of safety patterns. SAFPAT receives as input the designed system architecture and safety elements. SAFPAT performs changes in the system architecture by adding safety patterns in an automated fashion. SAFPAT has been implemented in DLV, a logic programming language based on the Answer Set Programming paradigm (Leone et al., 2006). SAFPAT is the backend of the plugin proposed by this paper.

SAFPAT consists of a domain-specific language (DSL) for embedded systems and reasoning principles that enable the automated recommendation of which patterns and where in the system architecture they shall be deployed. These reasoning principles are automated by the DLV engine.

**DSL.** The DSL enables the specification of architectural elements (*e.g.*, components and channels), safety elements (*e.g.*, faults), and safety patterns. SAFPAT currently supports the following safety patterns: *Acceptance Voting*, *Homogeneous Duplex*, *Heterogeneous Duplex*, *Monitor-Actuator*, *Simplex Architecture*, and *Triple Modular Redundancy*.

Table 1 illustrates how SAFPAT provides semantically-rich description of safety patterns. To this end, we provide an example of a template which is similar to pattern templates appearing in the literature (Armoush, 2010)(Sljivo et al., 2020). We instantiate the template with the Homogeneous Duplex pattern. Specifically, Table 1 provides a high-level description of the pattern and its specification in SAFPAT. The assumptions described in the table are not meant to be comprehensive. By *assumptions*, we refer to requirements that shall be satisfied to ensure the safety patterns work as intended.

**Reasoning Principles.** SAFPAT provides means to reason about the safety of the system architecture, in particular to recommend safety patterns. SAFPAT consists of reasoning principle rules to determine when (a) a failure is avoided, (b) a fault is tolerated, and (c) a hazard is controlled (a.k.a. mitigated). Specifically, (a) *a failure is avoided* if a suitable safety pattern is deployed, (b) *a fault is tolerated* if all failures triggered by that fault are avoided, and (c) *a hazard is controlled* if the fault triggering failures leading to that hazard is tolerated.

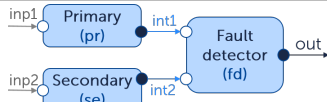
Whenever a safety pattern is recommended (see next paragraph), the rules for (a), (b), and (c) apply to infer which hazards have been controlled. SAFPAT only outputs architectural solutions where all hazards (received as input) have been controlled.

SAFPAT specifies reasoning rules for automating the recommendation of safety patterns. These rules specify conditions for when a particular pattern can be recommended to avoid failures triggered by faults.

The following are the main conditions specified by SAFPAT when recommending the Homogeneous Duplex pattern (`homogeneousDuplex`):

- there is a fault FT in the hardware component PR that triggers a failure FL leading to hazard HZ;
- `homogeneousDuplex` is suitable for addressing hardware faults;
- `homogeneousDuplex` is suitable for avoiding FL's type of failure (*i.e.*, erroneous or loss);
- `homogeneousDuplex` is suitable for addressing the ASIL of HZ. Since the Homogeneous Duplex pattern is suitable for ASIL D, it is also suitable for lower levels, *i.e.*, ASIL A, B and C.
- the safety mechanism type (*i.e.*, fault tolerant or

Table 1: Instantiation of the Homogeneous Duplex pattern. The assumptions are not meant to be comprehensive.

|                            | Description   | SAFPAT Specification  |
|----------------------------|---|---|
| Pattern name               | Homogeneous Duplex Pattern  | NAME=homogeneousDuplex;   |
| Structure                  |    | NAME=homogeneousDuplex;<br>COMPONENTS=[pr,se,fd];<br>INPUT_CH=[inp1,inp2];<br>INTERNAL_CH=[int1,int2];<br>OUTPUT_CH=[out];  |
| Intent                     | This pattern is fault tolerant, suitable for both addressing high criticality hazards (ASIL D) (Armouh, 2010) and tolerating hardware faults.   | TYPE_PAT=fault_tolerant;<br>TYPE_ASIL=d;<br>TYPE_CP=[hardware];<br>TYPE_FAIL=[erroneous,loss];  |
| Problem addressed          | This pattern tolerates faults by avoiding failures of type erroneous or loss.   |   |
| Assumptions (requirements) | The primary and the secondary components shall be identical.<br>The primary and the secondary components shall be allocated to different hardware units.<br>The fault detector shall be verified. | TYPE_ASSUMPTION=are_identical;<br>COMPONENTS=[pr,se];<br>TYPE_ASSUMPTION=are_decoupled;<br>COMPONENTS=[pr,se];<br>TYPE_ASSUMPTION=are_verified;<br>COMPONENTS=[fd]; |

detection) of heterogeneousDuplex matches the type of the safety mechanism chosen by the user.

SAFPAT may provide multiple architectural solutions as output, with different safety patterns for each solution. For example, possible patterns for tolerating a software fault include the use of either the Acceptance Voting pattern or the Heterogeneous Duplex pattern. The user may select the most suitable system architecture with patterns based on some criteria such as the ones described in Section 3.1.

SAFPAT also specifies requirements for ensuring safety integrity w.r.t. the allocation of software components to hardware components. For example, consider an allocation of a software component SW1 into a hardware component HW1. SAFPAT's reasoning rules check whether the ASIL of SW1 is higher than the ASIL of HW1. If this condition is true, SAFPAT provides a requirement to allocate SW1 to a hardware component with the same ASIL of SW1.

We refer the interested reader to (Dantas et al., 2020) for the detailed description about SAFPAT, including its reasoning principles rules.

### 3 SAFETY PATTERN SYNTHESIS

Safety Pattern Synthesis is a plugin of the model-based system engineering tool AutoFOCUS (AF3) for recommending safety patterns. Figure 1 depicts the artifacts that are used by Safety Pattern Synthesis

(system architecture and safety elements) and which artifacts are produced (system architecture with patterns and requirements).

- **System Architecture:** This artifact consists of the designed system architecture. The designed architecture includes the task and platform architecture for the system, and the allocation of tasks to hardware units. In addition, architectural components (tasks or hardware units) may be assigned to the ASIL that such components shall be implemented. We assume that the system architecture is designed following an model-based engineering approach, *e.g.*, developed in AF3.
- **Safety Elements:** This artifact consists of the results from a safety analysis<sup>1</sup> carried out by a safety engineer on the designed system architecture. This consists of hazards, faults and failures. Faults are associated with architectural components (tasks or hardware units) in the system architecture. As shown in Section 3.1, we have developed a wizard for defining these safety elements in AF3.
- **Safety Pattern Synthesis:** The developed plugin recommends safety architecture patterns based on the system architecture and safety elements. The reasoning on which pattern to select and where to place the selected pattern in the architecture is

<sup>1</sup>In our example described in Section 4, the safety analysis was carried out using the STPA method

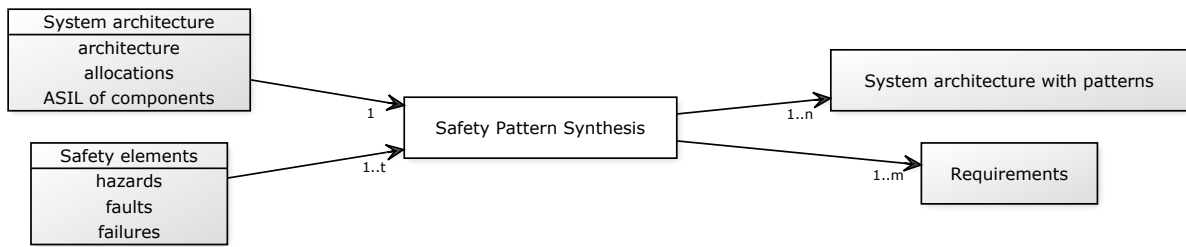


Figure 1: Safety Pattern Synthesis: Simplified diagram. Gray boxes are artifacts received as input or generated for output.

performed with the help of SAFPAT (described in Section 2.3). Safety Pattern Synthesis generates the following artifacts:

- **System Architecture with Patterns:** Safety Pattern Synthesis provides a list of modified AF3 architecture models with patterns. For example, possible solutions for tolerating a hardware fault in the platform architecture include the use of either the Homogeneous Duplex pattern or the Triplex Modular Redundancy pattern. The user of Safety Pattern Synthesis shall then select the most suitable architecture for the system. Section 3.1 provides some criteria to assist the user in selecting the architecture.
- **Requirements:** Safety Pattern Synthesis provides requirements that shall be implemented during system development. In particular, Safety Pattern Synthesis provides requirements for (a) the recommended safety patterns, *i.e.*, requirements to ensure that the recommended safety patterns work as intended (examples of such requirements are described in Table 1), and (b) safety integrity, *e.g.*, to ensure that the allocation of tasks to hardware units complies with the ASIL assigned to tasks.

The following section describes the high-level architecture of Safety Pattern Synthesis, including implementation details and how an user interacts with Safety Pattern Synthesis.

### 3.1 High-level Architecture

The architecture of Safety Pattern Synthesis is illustrated by Figure 2. Safety Pattern Synthesis has been developed, as part of the AF3 framework, in Java (frontend) and DLV (backend), and currently works under Linux and Windows.

Safety Pattern Synthesis receives the input artifacts described above. The architecture of Safety Pattern Synthesis consists of the following components:

- **Safety Elements Wizard:** This component provides a wizard to enable users to define the safety

elements obtained from a safety analysis. The component requires an interface to the designed system architecture so that it can assign faults to components (*i.e.*, tasks or hardware units). Figure 4 illustrates our wizard to define safety elements. Firstly, the user defines a hazard consisting of the hazard’s description and its ASIL. Secondly, the user selects the (possibly) faulty components (either tasks for software faults or ECUs for hardware faults). Thirdly, for each selected fault, the user defines the type of failure (either erroneous or loss) triggered by the fault.<sup>2</sup> Finally, the user defines which type of safety mechanism (either fault detection or fault tolerant) shall tolerate the identified faults, and consequently address the identified hazard.

- **Model-to-Model Transformation from AF3 to SAFPAT:** To make use of SAFPAT for safety pattern recommendation, we implemented a model-to-model transformation from AF3 (which is implemented in Java) to SAFPAT (which is implemented in DLV). We transform AF3 system architecture models and safety element models (specified in the wizard) to SAFPAT models. Figure 3 illustrates a model-to-model transformation from a task architecture modeled in AF3 to SAFPAT. The translation is implemented with the help of the EmbASP framework (EmbASP, 2018). EmbASP enables the Java representation of predicates that are specified in a DLV program. We represent in Java each relevant predicate that can be used in the DSL of SafPat. This includes the representation of architectural elements designed in AF3 and safety elements. Technically, we represent each relevant predicate by a Java class.
- **SafPat:** This component reasons about the safety of the system architecture. Explained in Section 2.3, this component is the backend of Safety

<sup>2</sup>Currently, Safety Pattern Synthesis supports two Guidewords, erroneous and loss, which are most commonly used in methodologies such as HAZOP. We plan in the future to incorporate other Guidewords such as early and late.



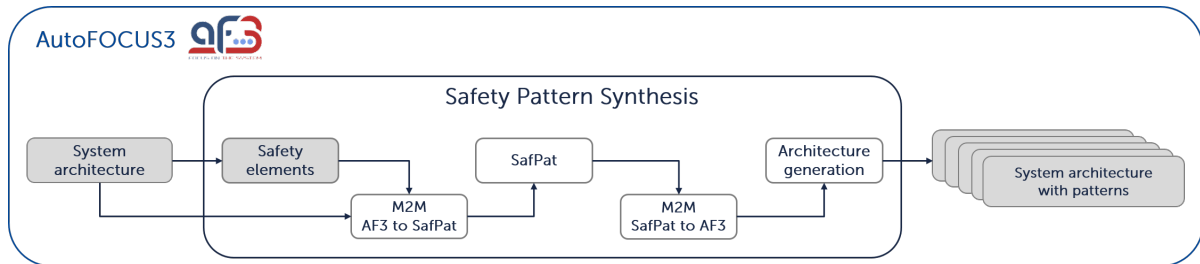


Figure 2: Architecture of Safety Pattern Synthesis. Gray boxes are artifacts either received as input or generated for output. Safety Pattern Synthesis generates requirements for each **System architecture with patterns** (output artifact).

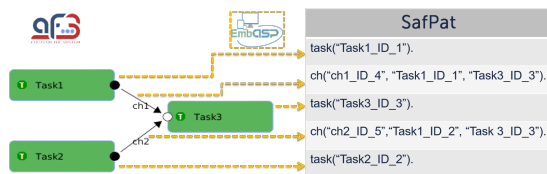


Figure 3: Illustration of a model-to-model transformation from a task architecture modeled in AF3 (left side) to SAF-PAT (right side) with the help of EmbASP (illustrated by the yellow arrows). The IDs of tasks and channels in the AF3 model are omitted in the figure.

Pattern Synthesis. Once the representation is fully realized, Safety Pattern Synthesis (via EmbASP) invokes SafPat by sending the translated system architecture and safety elements. Based on the specified reasoning principles, SAFPAT attempts to deploy safety patterns wherever they are applicable to tolerate faults. SAFPAT may return a list of modified architectures with patterns. SAFPAT also outputs requirements (a.k.a. assumptions).

- Model-to-Model Transformation from SAFPAT to AF3:** This component translates the results obtained from SAFPAT to AF3 models enabling the representation of the system architecture with safety patterns in AF3. The translation is obtained with the help of the EmbASP framework that enables the parsing of DLV facts to Java. The translation is similar to the one illustrated in Figure 3 (but in reverse order, *i.e.*, from SAFPAT to AF3). To identify the changes in the system architecture, SAFPAT makes explicit all the changes made in the architecture by using a prefixing scheme. This prefixing scheme is used for new channels and new components, and for channels (from the baseline architecture) to be removed.
- Architecture Generation:** This component provides a list of system architectures with safety patterns represented as AF3 models to be selected by the user. This component implements a wizard for visualizing the solutions obtained from SAFPAT, including a number of criteria to assist the user in selecting the most suitable architecture for sys-

tem. Currently, Safety Pattern Synthesis supports the following criteria:

- Number of new components** describes the number of diverse application components required by the safety pattern. For example, Heterogeneous Duplex pattern requires one redundant diverse component. The user can make a decision based on, *e.g.*, the number of application components to be developed.
- Number of replica components** describes the number of redundant components required by the safety pattern. A user can make a decision based on, *e.g.*, the number of replica that need to be introduced.
- Number of pattern support components** describes the number of non-application components required by the safety pattern. Examples of non-application components are the fault detector of the Heterogeneous Duplex pattern and the monitor of the Simplex Architecture pattern. A user can make a decision based on, *e.g.*, the overhead introduced by the safety pattern and on the number of pattern support components to be developed.
- Simplified (a.k.a. degraded)** describes whether the modified architecture with patterns contains a simplified application component. For example, the redundant component of the Simplex Architecture pattern is a simpler version of the primary component. A user can make a decision based on, *e.g.*, the minimum available fidelity level of components.

Safety Pattern Synthesis provides two wizards for visualizing the architectural solutions with patterns. One wizard with a spider chart view for showing the criteria based on the recommended patterns. The other wizard implements a table view that provides more detail, in addition to criteria, such as which safety pattern was recommended and additional requirements. For example, the highlighted solution in Figure 5 deploys one instance of the Heterogeneous Duplex pattern

(htd) and one instance of the Homogeneous Duplex pattern (hmd). Once the user has chosen the system architecture, the user selects and exports an architecture by clicking on the “Select” and “Export” buttons, respectively.

The exported architectural solution (**System architecture with patterns**) will be shown the modelling view of AF3. An example of the exported solution is presented in Section 4.2.

**Remark:** Figure 5 shows the additional requirements for ensuring (a) the recommended safety pattern work as intended, and (b) the safety integrity of the system. The highlighted solution in Figure 5 contains 15 requirements, including the ones for the Homogeneous Duplex pattern described in the pattern template (see Table 1). Once an architectural solution has been exported, the implementation of the additional requirements shall be carried during the system development.

### 3.2 Download

We have built a binary of Safety Pattern Synthesis to ease the use of the plugin. The binary of Safety Pattern Synthesis can be download here (Safety Pattern Synthesis, 2021). It also contains a video illustrating how one can run Safety Pattern Synthesis to select safety patterns in an automated fashion.

## 4 CASE STUDY

We consider an industrial use case taken from the automotive domain. We describe the system architecture and selected safety elements (*e.g.*, faults) that will serve as input artifacts for Safety Pattern Synthesis. We then run Safety Pattern Synthesis to recommend safety patterns for tolerating the identified faults.

### 4.1 Use Case

**System Architecture.** We consider an industrial use-case, namely the Highway Pilot (HWP). The *nominal function* of a HWP is predominantly defined as the longitudinal and lateral control of a vehicle’s movement up to a given maximum speed to realize a trajectory under consideration of the limitations given by the lane, other vehicles, and the ego vehicle itself.

This specification of the system’s nominal function can be further broken down into *functional requirements*. The HWP shall

1. **Req 1:** not cause the ego vehicle to exceed its maximum velocity,

2. **Req 2:** keep the ego vehicle either at a set speed or adapt its speed to a leading vehicle,
3. **Req 3:** keep the ego vehicle at the center of the current lane,
4. **Req 4:** include a *stop & go* functionality, and
5. **Req 5:** inform the driver about its status.

As the HWP takes over the complete Dynamic Driving Task (DDT) as well as Object and Event Response (OEDR), the system is classified as a highly safety-critical, ASIL-D rated, level 3 Automated Driving System (ADS) according to the SAE J3016 standard (SAEJ3061, 2012)(ISO26262, 2018).

HWP has been designed in AutoFOCUS3 (AF3) as part of the fortissimo<sup>3</sup> demonstrator platform. The HWP architecture consisting of both task and platform architectures are illustrated in Figure 6.

The *Sensor Data Fusion* processes data generated by a front-mounted sensors to determine the distance of leading vehicles or obstacles as well as the ego vehicle’s position within the lane. The *Adaptive Cruise Control (ACC)* and *Lane Keeping Assistance (LKA)* provide longitudinal and lateral control, respectively – each according to the HWP’s nominal function. The *Emergency Brake (EB)* provides longitudinal control in case a collision with an obstacle in front of the vehicle is deemed unavoidable. Here, the goal is not collision avoidance, but mitigation. The system is always active as such situations can arise both during manual driving as well as due to a fault of the HWP. The *Motion Control* coordinates the desired vehicle states given by the driving functions and controls the torque applied by the servo steering, and the throttle position accordingly.

The hardware dedicated to the execution of the HWP is specified by means of hierarchical platform architecture models. A simplified version of the platform architecture is illustrated in Figure 6.

Table 2 describes the allocation of task to hardware units, as well as the ASIL requirements for each component (*i.e.*, the ASIL that each component shall be implemented).

**Safety Elements.** For the sake of our evaluation, we consider the following safety elements that can be identified from a safety analysis. The identified safety elements are not meant to be comprehensive.

Table 3 describes the identified hazards HZ1 and HZ2. We assigned ASIL B to HZ2 as we consider the exposure of HZ2 as low probability of happening. These hazards may happen on the occurrence or presence of failures triggered by the following faults:

<sup>3</sup><https://www.fortiss.org/en/research/living-lab/detail/fortissimo>

**Hazard**

Description: Unintended emergency brake.

ASIL Level: ASIL B

**Faults and Failures**

| Component               | Type | Fault                               | Failure   |
|-------------------------|------|-------------------------------------|-----------|
| Adaptive Cruise Control | Task | <input type="checkbox"/>            | n/a       |
| Emergency Brake         | Task | <input checked="" type="checkbox"/> | ERRONEOUS |
| Lane Keeping Assistant  | Task | <input type="checkbox"/>            | n/a       |
| Motion Control          | Task | <input type="checkbox"/>            | n/a       |
| Sensor Data Fusion      | Task | <input type="checkbox"/>            | n/a       |
| AI Cores                | ECU  | <input type="checkbox"/>            | n/a       |
| AI SoC                  | ECU  | <input type="checkbox"/>            | n/a       |
| GPU                     | ECU  | <input type="checkbox"/>            | n/a       |
| ISP SoC                 | ECU  | <input type="checkbox"/>            | n/a       |
| MCU                     | ECU  | <input type="checkbox"/>            | n/a       |
| Vector Cores            | ECU  | <input checked="" type="checkbox"/> | LOSS      |

**Safety Mechanism**

Type of Mechanism: Fault Tolerant

Add

Figure 4: Screenshot of the wizard for defining safety elements.

Table 2: Allocation and ASIL of components. Note that last allocation does not comply with the safety integrity level required (*i.e.*, ASIL D task allocated to ASIL B hardware unit). The intended outcome is to show that Safety Pattern Synthesis can provide requirements to ensure the safety integrity w.r.t. allocations.

| Task                           | Hardware Unit                       |
|--------------------------------|-------------------------------------|
| Sensor Data Fusion<br>[ASIL B] | Host SoC (GPU)<br>[ASIL B]          |
| ACC<br>[ASIL D]                | MCU<br>[ASIL D]                     |
| LKA<br>[ASIL D]                | MCU<br>[ASIL D]                     |
| EB<br>[ASIL B]                 | Host SoC (Vector<br>Cores) [ASIL B] |
| Motion Control<br>[ASIL D]     | AI SoC<br>[ASIL B]                  |

Table 3: Identified hazard for the HWP system.

| Hazard | Description  | ASIL |
|--------|--|------|
| HZ1    | The vehicle violates the safety distance to other road users or objects on the road. | D    |
| HZ2    | Unintended emergency brake.  | B    |

- FT1: A software fault occurs in the algorithm implemented by the task implementing ACC causes the provided target deceleration value not be high enough to reach the safety distance between vehicles. The failure of type *erroneous* triggered by fault FT1 may lead to hazard HZ1.
- FT2: A hardware fault occurs in the hardware unit to which EB is allocated. This causes EB not to provide the target deceleration needed to avoid a front-end collision. The failure of type *loss* triggered by fault FT4 may lead to hazard HZ2.

## 4.2 Results

Consider the system architecture and the safety elements described in Section 4.1. We run Safety Pattern Synthesis to determine patterns that can tolerate the identified faults and address the identified hazards.

Safety Pattern Synthesis recommended twelve solutions for tolerating the identified faults. The recommended safety patterns are described in Table 4. Considering that both the system architecture has been loaded, and the identified safety elements have been annotated to architectural elements, *we accomplished the results depicted in Table 4 with a few clicks only.*

As an example, consider the task architecture from **Solution 10** illustrated in Figure 7, where the *Heterogeneous Duplex pattern* is applied to ACC and the *Monitor Actuator pattern* is applied to EB. Both ACC (V2) and Fault Detector tasks are created to tolerate the software faults that may be present in ACC (V1). That is, if one fault is detected in ACC (V1), the function ACC will continue operating using the outputs from ACC (V2). This solution contains additional requirements to be implemented to ensure that the pattern works as intended such as ACC (V1) and ACC (V2) tasks shall be developed using different design, and for safety integrity such as ACC (V2) and Fault Detector tasks shall be developed using ASIL D requirements to comply with the safety integrity of ACC (V1), as shown in Table 2.

The Monitor Actuator task is created to tolerate hardware faults of type loss in the hardware unit where EB is allocated (*i.e.*, Vector Cores). To this end, the Monitor Actuator task shall implement a timeout algorithm to detect failures of type loss triggered by a fault in Vector Cores. This requirement is provided by Safety Pattern Synthesis to be realized during the system development.

Safety Pattern Synthesis provides requirements to ensure safety integrity w.r.t. allocations. Consider the

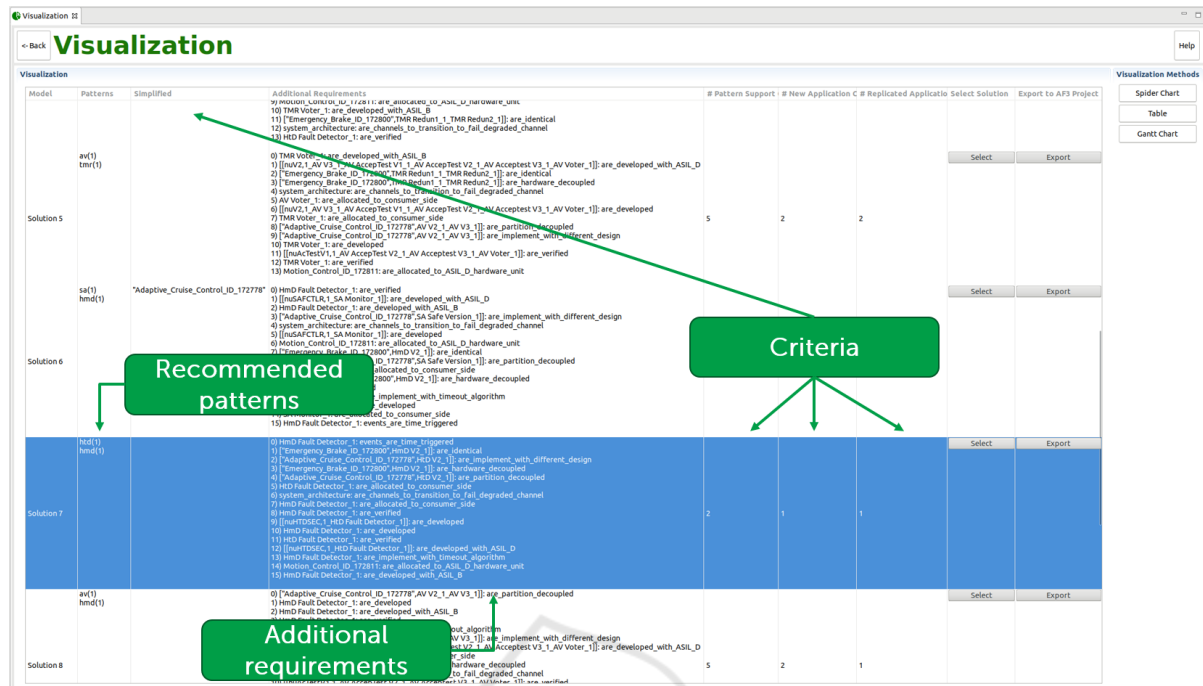


Figure 5: Screenshot of the wizard for visualizing architectural solutions with patterns. It includes the recommended patterns, the additional requirements to be implemented during the system development, and some criteria to assist the user in selecting the most suitable architecture for the system.

allocation of the Motion Control [ASIL D] to AI SoC [ASIL B] in Table 2. This allocation does not comply with the safety integrity ASIL D given that AI SoC is ASIL B. Safety Pattern Synthesis provides a requirement to ensure the safety integrity, *i.e.*, Motion Control shall be allocated to an ASIL D hardware unit.

### 4.3 Discussion

We discuss some issues that are left out of the scope of this work, but that are important for Safety Pattern Synthesis deployment in industry.

**Handling Potential Design Option Explosion.** Safety Pattern Synthesis may provide a considerably high number of solutions to be selected by the user, as shown in Table 4. Currently, Safety Pattern Synthesis provides four criteria (see Section 3.1) to assist the user with this decision. We are investigating further criteria used, for example, by AutoFOCUS3 for design space exploration (Eder et al., 2020), such as, the performance overhead caused by safety patterns, implementation cost, and the hardware resource usage required by safety patterns. These criteria can be used to rank more precisely design options eliminating non-optimal ones, thus reducing options.

**Towards Incremental Development.** In this work, we consider only one development loop. That is, once safety elements are identified a user can make use of Safety Pattern Synthesis to deploy safety patterns into the system architecture. It remains to be investigated how Safety Pattern Synthesis can be extended to support incremental development where several loops are involved. For example, consider an architecture with safety patterns, possibly recommended by Safety Pattern Synthesis, and a new unhandled fault. Currently Safety Pattern Synthesis would recommend new safety patterns without modifying the existing ones. This may not lead to optimal solutions as it does not exploit synergies between patterns, *e.g.*, a safety pattern that provide more fault tolerance can subsume other weaker patterns. It seems possible to use Safety Pattern Synthesis in a search mechanism procedure where pattern recommendations are withdrawn by backtracking and new more optimized architectures are recommend.

**Scalability.** Safety Pattern Synthesis reduces the problem of pattern recommendation to a logical theory (specified by SAFPAT) that is NP-complete in general. This does not necessary mean that Safety Pattern Synthesis cannot be used in practice as specialized engines, such as SMT-solvers, have been used in industry projects (Eder et al., 2020) for other

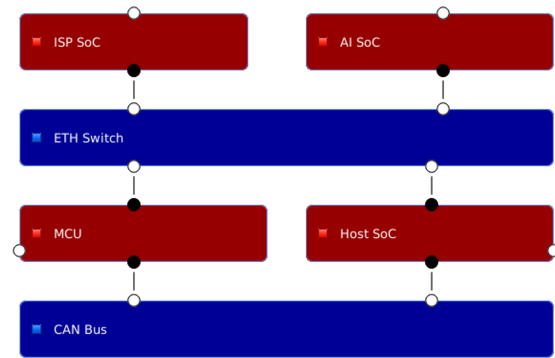
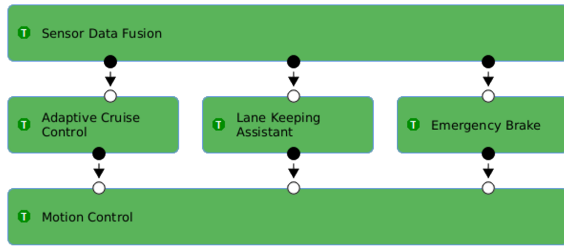


Figure 6: Highway Pilot (HWP): Task (left side) and platform (right side) architectures.

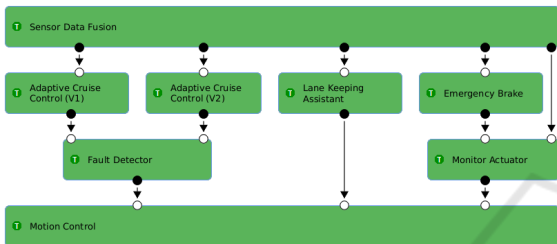


Figure 7: HWP Task architecture with safety patterns.

design space exploration problems that are also NP-hard. Moreover, given that the focus of Safety Pattern Synthesis is on development time and not runtime, Safety Pattern Synthesis’s performance requirements ranges on hours (and even days). However, a more dedicated study shall be carried out to determine exactly the Safety Pattern Synthesis’s scalability. We are aiming to achieve this by using more realistic examples provided by our industry collaboration. The results shown in Section 4.2 only took a few seconds to be computed, but the computation may change depending on, *e.g.*, the size of the system architecture and the number of safety elements.

## 5 RELATED WORK

A catalog of safety architecture patterns for safety-critical systems have been presented in (Douglass, 2012)(Armoush, 2010)(Preschern et al., 2013a). In particular, (Armoush, 2010) has proposed a pattern template for providing a consistent representation for safety architecture patterns. This template has been instantiated with several patterns for tolerating hardware and software faults. Safety Pattern Synthesis currently supports a subset of such patterns.

We have been inspired by (Martin et al., 2020) that proposed a pattern-based approach providing guidance w.r.t. selection of safety (and security) patterns. A key difference to our work is that we propose a

tool for automating the recommendation of safety patterns, while in (Martin et al., 2020) the recommendation of patterns was done in a manual fashion. This approach also includes guidance for selecting security patterns (*e.g.*, firewall) to address security problems (*i.e.*, threats), and for clarifying possible safety conflicts when deploying such patterns. For example, one may deploy a firewall to mitigate identified threats. The deployed firewall may, however, lead to new system faults if it erroneously blocks legitimate messages. We are investigating how to include security aspects into Safety Pattern Synthesis by extending the work by (Dantas et al., 2020) to reason about the security of system architectures to automate the recommendation or security patterns.

Approaches combining MBSE with safety analysis have been proposed by, *e.g.*, (Papadopoulos et al., 2011)(Belmonte and Soubiran, 2012). For example, the HiP-HOPS tool (Papadopoulos et al., 2011) has been proposed to semi-automate the safety analysis process (using FTA and FMEA techniques) on system architectures. HiP-HOPS enables a user to annotate the given architecture with data describing how individual components can fail. HiP-HOPS examines the data and automatically identifies a list of system faults that shall be later addressed by safety mechanisms. Our work complements (Papadopoulos et al., 2011) by providing means to automate the recommendation of such safety mechanisms to tolerate identified faults. We are interested in extending Safety Pattern Synthesis to enable the automatic identification of system faults by, *e.g.*, using tools like HiP-HOPS.

(Eder et al., 2017) proposed a design space exploration approach to enable the allocation of software components into hardware units in a semi-automated fashion. This approach takes into account the structure of system architectures (incl., software components and hardware units), and a DSL to formalize requirements (*e.g.*, timing, memory consumption) w.r.t. the design space exploration problem. This DSL is

Table 4: Solutions recommended by our plugin.

| Solution | Recommended Safety Patterns  |
|----------|--|
| 0        | <i>Simplex Architecture</i> for tolerating the software fault FT1, <i>i.e.</i> , the pattern is applied to ACC. <i>Heterogeneous Duplex</i> for tolerating the hardware fault FT2, <i>i.e.</i> , the pattern is applied to EB.                                       |
| 1        | Two instances of the <i>Heterogeneous Duplex pattern</i> , where the first instance tolerates the software fault FT1 ( <i>i.e.</i> , pattern is applied to ACC), and the second instance tolerates the hardware fault FT2 ( <i>i.e.</i> , pattern is applied to EB). |
| 2        | <i>Acceptance Voting</i> for fault FT1. <i>Heterogeneous Duplex</i> for fault FT2.   |
| 3        | <i>Simplex Architecture</i> for fault FT1. <i>Triple Modular Redundancy</i> for fault FT2.   |
| 4        | <i>Heterogeneous Duplex</i> for fault FT1. <i>Triple Modular Redundancy</i> for fault FT2.   |
| 5        | <i>Acceptance Voting</i> for fault FT1. <i>Triple Modular Redundancy</i> for fault FT2.  |
| 6        | <i>Simplex Architecture</i> for fault FT1. <i>Homogeneous Duplex</i> for fault FT2.  |
| 7        | <i>Heterogeneous Duplex</i> for fault FT1. <i>Homogeneous Duplex</i> for fault FT2.  |
| 8        | <i>Acceptance Voting</i> for fault FT1. <i>Homogeneous Duplex</i> for fault FT2.   |
| 9        | <i>Simplex Architecture</i> for fault FT1. <i>Monitor Actuator</i> for fault FT2.  |
| 10       | <i>Heterogeneous Duplex</i> for fault FT1. <i>Monitor Actuator</i> for fault FT2.  |
| 11       | <i>Acceptance Voting</i> for fault FT1. <i>Monitor Actuator</i> for fault FT2.   |

specified as a first-order logic language that can be automated by solving techniques such as Satisfiability Modulo Theories (SMT) (de Moura and Bjørner, 2008). The combination of the proposed DSL with the designed system architectures enabled the applicability of the semi-automated design space exploration for allocating software components into hardware units. This approach has been extended to enable a synthesis of the topology of technical platforms together with a deployment (Eder et al., 2018b). The approach has been implemented as a feature of AF3. Safety Pattern Synthesis currently deals with requirements (in terms of allocation constraints) provided by SAFPAT in a manual fashion. We believe that we can combine our work with (Eder et al., 2018b)(Eder et al., 2017) to implement these requirements in an automated fashion.

## 6 CONCLUSION

This paper presented Safety Pattern Synthesis – a plugin for automating the recommendation of safety patterns within the model-based system engineering tool AutoFOCUS3. Safety Pattern Synthesis is guided by the results of a safety analysis. It takes as input information on how faults may trigger identified hazards in the system architecture. Guided by this information, Safety Pattern Synthesis automatically recommends safety patterns to address the identified hazards. Safety Pattern Synthesis also recommends requirements (w.r.t. the recommended patterns and safety integrity) that shall be implemented during the system development.

Safety Pattern Synthesis has been developed with the intention of reducing the effort required by safety engineers while carrying out a safety analysis on safety-critical systems such as autonomous vehicles.

## ACKNOWLEDGEMENTS

We thank Christoph Ainhauser and Sandro Nüesch for their help in the early phase of this work.

## REFERENCES

- Amorim, T., Martin, H., Ma, Z., Schmittner, C., Schneider, D., Macher, G., Winkler, B., Krammer, M., and Kreiner, C. (2017). Systematic Pattern Approach for Safety and Security Co-engineering in the Automotive Domain. In Tonetta, S., Schoitsch, E., and Bitsch, F., editors, *SAFECOMP 2017*.
- Aravantinos, V., Voss, S., Teufel, S., Hölzl, F., and Schätz, B. (2015). AutoFOCUS 3: Tooling concepts for seamless, model-based development of embedded systems. In *ACES-MB*, pages 19–26.
- Armoush, A. (2010). *Design Patterns for Safety-Critical Embedded Systems*. PhD thesis, RWTH Aachen University.
- Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. E. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33.
- Bak, S., Chivukula, D. K., Adekunle, O., Sun, M., Caccamo, M., and Sha, L. (2009). The system-level simplex architecture for improved real-time embedded system safety. In *15<sup>th</sup> IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, pages 99–107. IEEE Computer Society.
- Barner, S., Chauvel, F., Diewald, A., Eizaguirre, F., Hagen, Ø., Migge, J., and Vasilevskiy, A. (2018). *Modeling and Development Process*, pages 87–161. CRC Press.

- Barner, S., Diewald, A., Migge, J., Syed, A., Fohler, G., Faugère, M., and Gracia Pérez, D. (2017). DREAMS Toolchain: Model-Driven Engineering of Mixed-Criticality Systems. In *Proceedings of the ACM/IEEE 20<sup>th</sup> International Conference on Model Driven Engineering Languages and Systems (MODELS '17)*, pages 259–269. IEEE.
- Belmonte, F. and Soubiran, E. (2012). A model based approach for safety analysis. In Ortmeier, F. and Daniel, P., editors, *Computer Safety, Reliability, and Security - SAFECOMP 2012 Workshops: Sassur, ASCoMS, DESEC4LCCI, ERCIM/EWICS, IWDE, Magdeburg, Germany, September 25-28, 2012. Proceedings*, volume 7613 of *Lecture Notes in Computer Science*, pages 50–63. Springer.
- Biondi, A., Nesti, F., Cicero, G., Casini, D., and Buttazzo, G. C. (2020). A safe, secure, and predictable software architecture for deep learning in safety-critical systems. *IEEE Embed. Syst. Lett.*, 12(3):78–82.
- Dantas, Y. G., Kondeva, A., and Nigam, V. (2020). Less manual work for safety engineers: Towards an automated safety reasoning with safety patterns. In *ICLP*.
- de Moura, L. M. and Bjørner, N. (2008). Z3: An Efficient SMT Solver. In Ramakrishnan, C. R. and Rehof, J., editors, *TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer.
- Douglass, B. P. (2012). *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*.
- Eder, J., Bahya, A., Voss, S., Ipatiov, A., and Khalil, M. (2018a). From deployment to platform exploration: Automatic synthesis of distributed automotive hardware architectures. In *MODELS 2018*, *MODELS '18*, page 438–446.
- Eder, J., Bayha, A., Voss, S., Ipatiov, A., and Khalil, M. (2018b). From deployment to platform exploration: Automatic synthesis of distributed automotive hardware architectures. In Wasowski, A., Paige, R. F., and Haugen, Ø., editors, *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018*, pages 438–446. ACM.
- Eder, J., Voss, S., Bayha, A., Ipatiov, A., and Khalil, M. (2020). Hardware architecture exploration: automatic exploration of distributed automotive hardware architectures. *Software and Systems Modeling*.
- Eder, J., Zverlov, S., Voss, S., Khalil, M., and Ipatiov, A. (2017). Bringing DSE to Life: Exploring the Design Space of an Industrial Automotive Use Case. In *MODELS 2017*, pages 270–280. IEEE Computer Society.
- EmbASP (2018). EmbASP. Available at <https://www.mat.unical.it/calimeri/projects/embasp/>.
- fortiss GmbH (2020). AutoFOCUS 2.19. Available at <https://www.fortiss.org/en/publications/software/autofocus-3>.
- IEC61508 (2010). IEC 61508, Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 7: Overview of techniques and measures. Available at <http://www.cechina.cn/eletter/standard/safety/iec61508-7.pdf>.
- ISO26262 (2018). ISO 26262, road vehicles — functional safety — part 6: Product development: software level. Available at <https://www.iso.org/standard/43464.html>.
- Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. (2006). The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562.
- Martin, H., Ma, Z., Schmittner, C., Winkler, B., Krammer, M., Schneider, D., Amorim, T., Macher, G., and Kreiner, C. (2020). Combined automotive safety and security pattern engineering approach. *Reliab. Eng. Syst. Saf.*, 198:106773.
- Papadopoulos, Y., Walker, M., Parker, D., Ruede, E., Hamann, R., Uhlig, A., Graetz, U., and Lien, R. (2011). Engineering failure analysis and design optimisation with HiP-HOPS. *Journal of Engineering Failure Analysis*, 18(2):590–608.
- Preschern, C., Kajtazovic, N., and Kreiner, C. (2013a). Building a safety architecture pattern system. In van Heesch, U. and Kohls, C., editors, *Proceedings of the 18th European Conference on Pattern Languages of Program, EuroPLOP 2013*, pages 17:1–17:55. ACM.
- Preschern, C., Kajtazovic, N., and Kreiner, C. (2013b). Security analysis of safety patterns. *PLoP*, pages 12:1–12:38.
- SAEJ3061 (2012). SAE J3061: Cybersecurity guidebook for cyber-physical vehicle systems. Available from <https://www.sae.org/standards/content/j3061/>.
- Safety Pattern Synthesis (2021). Safety Pattern Synthesis. Available at <https://download.fortiss.org/public/MODELSWARD2022/SafetyPatternSynthesis.zip>.
- Slijivo, I., Uriagereka, G. J., Puri, S., and Gallina, B. (2020). Guiding assurance of architectural design patterns for critical applications. *J. Syst. Archit.*, 110:101765.



# A Toolchain for Synthesizing and Validating Safety Architectures

Yuri Gil Dantas<sup>1</sup> · Tiziano Munaro<sup>1</sup> · Carmen Carlan<sup>3</sup> · Vivek Nigam<sup>2</sup> · Simon Barner<sup>1</sup> · Shiqing Fan<sup>2</sup> · Alexander Pretschner<sup>1,4</sup> · Ulrich Schöpp<sup>1</sup> · Sergey Tverdyshev<sup>2</sup>

Received: 3 July 2022 / Accepted: 13 January 2023  
© The Author(s), under exclusive licence to Springer Nature Singapore Pte Ltd 2023

## Abstract

Autonomous vehicles handle complicated tasks that may lead to harm when performed incorrectly. These harms, in particular when caused by system faults, may be avoided by the deployment of safety architectural patterns, such as the Heterogeneous Duplex pattern. Our goal is to provide safety engineers with computer-aided support for synthesizing architectures with safety architecture patterns. To this end, we build on our previous work in which we proposed a model-based system engineering plugin to enable the model-driven approach using safety architecture patterns. This article proposes a toolchain for synthesizing the structure and switching logic of safety architectures, as well as for validating them through simulation-based fault-injection. We validate our toolchain using an industrial use-case for autonomous driving systems, namely, a Highway Pilot system.

**Keywords** Model-based system engineering · Toolchain · Safety architecture patterns · Reconfiguration · Simulation

## Introduction

Autonomous vehicles consist of several functions for sensing the environment and for driving with little or no intervention from human drivers. Examples of such functions are Sensor Data Fusion, Adaptive Cruise Control, and Lane Keeping Assist. These functions enable, respectively, the perception of road objects, the adjustment of speed values and the safe distance from road objects, and lane detection to keep the vehicle centered in a detected lane. Autonomous driving

functions are implemented as software and deployed over hardware units, such as Electronic Control Units (ECUs).

Autonomous driving functions may cause harm (e.g., injury or even death) to drivers or passengers if performed incorrectly. Root causes of such harm include (systematic) software faults and (random) hardware faults. At the architecture level, safety engineers make use of safety architecture patterns to tolerate software and hardware faults [1, 2]. Examples of safety architecture patterns are Heterogeneous Duplex and Triple Modular Redundancy.

In our previous article [3], we have proposed a model-based system engineering (MBSE) plugin to assist safety engineers with the selection of safety architecture patterns.

---

This article is part of the topical collection “Advances on Model-Driven Engineering and Software Development” guest edited by Luís Ferreira Pires and Slimane Hammoudi.

---

✉ Yuri Gil Dantas  
dantas@fortiss.org

Tiziano Munaro  
munaro@fortiss.org

Carmen Carlan  
ccarlan@ecr.ai

Vivek Nigam  
vivek.nigam@huawei.com

Simon Barner  
barner@fortiss.org

Shiqing Fan  
shiqing.fan@huawei.com

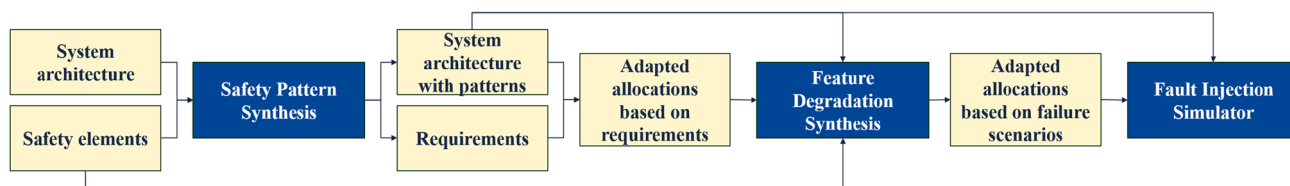
Alexander Pretschner  
pretschner@fortiss.org

Ulrich Schöpp  
schoepp@fortiss.org

Sergey Tverdyshev  
sergey.tverdyshev@huawei.com

- <sup>1</sup> fortiss GmbH, Munich, Germany
- <sup>2</sup> Huawei Technologies Düsseldorf GmbH, Düsseldorf, Germany
- <sup>3</sup> Edge Case Research, Munich, Germany
- <sup>4</sup> Technische Universität München, Munich, Germany





**Fig. 1** Workflow of the proposed toolchain. Blue boxes illustrate the tools of the proposed toolchain and yellow boxes illustrate either input artifacts or output artifacts. The input artifact *System*

The selected system architecture with patterns (here also called safety architecture) includes requirements with respect to allocations of redundant components, i.e., the primary and the redundant component shall be allocated to different hardware units. These allocation requirements shall be taken into account to ensure that the pattern works as intended, in particular to avoid common hardware faults. Once these allocations have been determined, one needs to validate whether the safety architecture is capable of switching to the redundant component if a fault has been identified in the primary component. To this end, this article proposes a toolchain for synthesizing the structure and switching logic of safety architectures, as well as for validating them by means of simulation-based fault-injection.

The workflow of the proposed toolchain is illustrated in Fig. 1. Our toolchain consists of (i) **Safety Pattern Synthesis** [3]: A safety architecture patterns (incl., fault tolerant patterns) to address safety artifacts (incl., software and hardware faults) identified during a safety analysis of the autonomous vehicle system architecture. **Safety Pattern Synthesis** also provides (allocation) requirements for the recommended patterns. (ii) **Feature Degradation Synthesis**: A plugin for determining (potentially degraded) allocations for failure scenarios caused by hardware faults in the hardware units, where pattern components (e.g., redundant components) are allocated. (iii) **Fault Injection Simulator**: A simulator tool for validating the fault-tolerance of the autonomous vehicle system under test by means of fault-injection, in particular **Fault Injection Simulator** validates the switching logic of the system architecture with patterns.

This article is an extended version of our MODELSWARD 2022 article [3]. The main extensions are the **Feature Degradation Synthesis**, and the **Fault Injection Simulator** as part of our toolchain for synthesizing and validating safety architectures. We validate our toolchain using an industrial use-case for autonomous driving systems, namely, a Highway Pilot system. This article considers a far more complex Highway Pilot system than the the simplistic Highway Pilot system considered by our MODELSWARD 2022 article.

The remainder of the article is structured as follows. Section "**Background**" briefly presents background information

architecture consists of multiple views, including task architecture, platform architecture, and initial allocation of tasks to ECUs

to help the reader to understand the results described in the article. Section "**Case Study**" describes the Highway Pilot system architecture and selected results computed by our safety analysis. Our toolchain and the evaluation results are described in Sects. "**Safety Pattern Synthesis**", "**Feature Degradation Synthesis**", and "**Simulation**". After a discussion of the related work in Sect. "**Related Work**", we conclude in Sect. "**Conclusion**".

## Background

### AutoFOCUS3

AutoFOCUS3 [4] is an open source model-based systems engineering tool and research platform for safety-critical embedded systems that builds on the Eclipse Modeling Framework [5].

Following the SPES modeling approach and engineering methodology [6], the tool enables graphical modelling of designs and implementations of safety-critical systems in terms of the modeling viewpoints [7, 8] summarized below. Besides a model of the vertical and horizontal structure, all viewpoints provide a rich set of annotations that are mainly used to describe non-functional properties of the respective model element.

- The *requirements viewpoint* provides a simple metamodel to capture textual requirements roughly following the Volere template.
- The *logical viewpoint* [7] provides a metamodel for the platform-independent specification of systems that consists of a hierarchical network of components, whose data interface is described using typed logical input and output ports that may be connected using directed channels. Logical components may contain behaviour specifications in the form of timed state automata or (semantically equivalent) code specifications [7].
- The physical hardware platform metamodel of the *technical viewpoint* [8] enables to describe the resources of hierarchical networked computer systems. *Execution units* such as electronic control units, processors, proces-

**Table 1** Architecture pattern description template

| Field                      | Description   |
|----------------------------|---|
| Pattern name               | Name of this pattern.   |
| Structure                  | Block diagram of this pattern.  |
| Intent                     | Textual description of the purpose of this pattern.   |
| Problem addressed          | Textual description of the problem the pattern addresses.   |
| Assumptions (requirements) | Assumptions necessary for using this pattern, i.e., to ensure that the pattern works as intended. |

processor cores denote the processing elements provided by the architecture. The model also captures further resources such as memories and communication devices in terms of *transmission units* (e.g., networks and busses) as well as their interconnection. The physical hardware platform metamodel is accompanied by a *task architecture* metamodel that can be used to describe platform-specific aspects of software-based implementations of logical components (e.g., memory requirements, worst-case execution time, etc.). It consists of a flat network of tasks whose data-dependencies are defined in terms of signals.

- The *deployment viewpoint* is used to trace relationships between model elements located in different viewpoints in terms of *allocations*, e.g., to relate logical component to tasks, and tasks to execution units, respectively.

Based on these models, AutoFOCUS3 provides computer-aided support for engineering tasks in the design and validation phase.

A major focus of the tool are prototypical implementations for the automated optimization and analysis of system architectures, in particular for the synthesis of task-to-hardware deployments [9–11], the exploration of platform architectures [12, 13], as well as mixed-critical system architecture [14, 15] product-lines [16]. The different analyses are available from AutoFOCUS3's *Design Space Exploration* perspective [17] from the where they can be configured and executed, and which enables to visualize the determined solutions [18]. AutoFOCUS3 also provides a functional simulation [7] that can be used to validate designs in early development stages, and supports the standardized functional mockup interface (FMI) for co-simulation.

## Safety Terminology

System safety is defined in ISO 26262 as “absence of unreasonable risk”, where the risk is computed based on the the probability of the occurrence of hazardous events

and their severity. A hazardous event is a hazard occurring in a certain operational situation, leading to harm, where a hazard is “a potential source of harm caused by malfunctioning behaviour of the item” [19]. Hazardous events are characterized by severity, probability of exposure regarding operational situations, and controllability. Based on these characteristics of the identified hazardous events, the Automotive Safety Integrity Level (ASIL) of the system or system components is determined. ASIL specifies the ISO 26262 requirements and safety measures to be considered during the development of the system or system component under consideration, with the scope to eliminate unreasonable risk, where D is the most stringent and A the least stringent level. ISO 26262 addresses functional safety, which considers hazardous events caused by E/E systematic and random faults. A system fault is defined in ISO 26262 as an “abnormal condition that can cause an element or an item to fail”, whereas a system failure is the “termination of an intended behaviour of an element or an item due to a fault manifestation”. Avizienis et al. [20] propose a classification of failures, such as “halt failures”, i.e., “loss of function”, or “content failures”, e.g., “erroneous output”.

System faults may be systematic, originating in software, or random, originating in hardware. Faults that are root causes for hazardous events shall be prevented, tolerated or removed. Fault tolerance avoids the occurrence of system failures in the presence of faults. To tolerate faults, safety engineers may use different safety measures, such as safety architecture patterns, which we present in the following paragraph. For the sake of presentation, we refer to hazards, faults and failure as safety artifacts.

**Safety Architecture Patterns** Safety architecture patterns [1, 2, 21] (safety patterns for short) are abstract solutions to tolerate software/hardware faults in the system architecture. Safety architecture patterns are described in an abstract form and they are implementation-agnostic. Safety patterns are often described using a pattern template [1, 22]. An example of a pattern template is depicted in Table 1.

A subset of safety patterns are the so-called fault tolerant patterns. A *fault tolerant pattern* ensures that the system will continue to operate when a fault is detected by providing a redundant component to take over. Fault tolerant patterns improve the availability of the system given the redundant component. A fault tolerant pattern may also transition the system to a safe state or shutdown the system in the presence of a failure in the redundant component. Examples of fault tolerant patterns are Acceptance Voting, Homogeneous Duplex, Heterogeneous Duplex, Triple Modular Redundancy, and Simplex Architecture [1, 23–25].

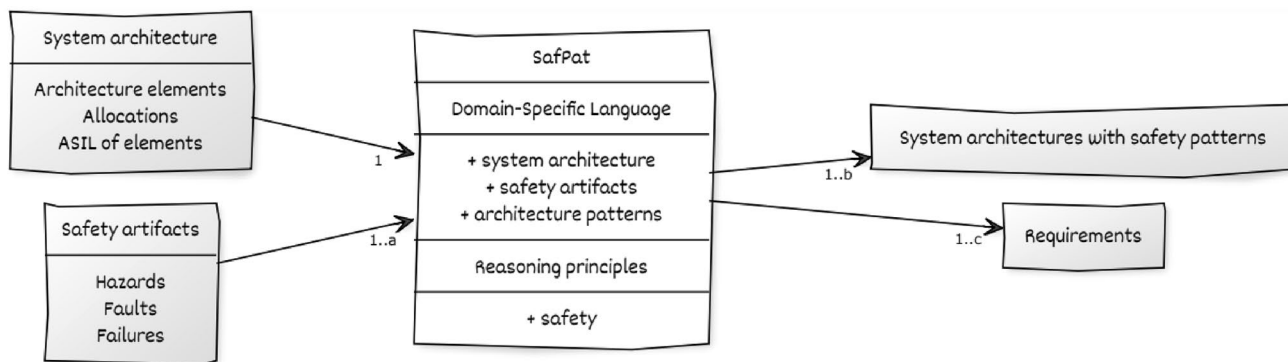


Fig. 2 Inputs and outputs artifacts received and generated by SafPat

Table 2 Instantiation of the Homogeneous Duplex pattern

| Field                      | Description   | SAFPAT Specification  |
|----------------------------|---|---|
| Pattern name               | Homogeneous Duplex Pattern  | NAME=homogeneousDuplex;   |
| Structure                  |   | COMPONENTS=[pr,se,fd];<br>INPUT_CH=[inp1,inp2];<br>INTERNAL_CH=[int1,int2];<br>OUTPUT_CH=[out];   |
| Intent                     | This pattern is fault tolerant, suitable to address high criticality hazards (ASIL D) [1] and tolerate hardware faults.   | TYPE_PAT=fault.tolerant;<br>TYPE_ASIL=d;<br>TYPE_FAULT=[hardware];<br>TYPE_FAIL=[erroneous,loss];   |
| Problem addressed          | This pattern tolerates faults by avoiding failures of type erroneous or loss.   |   |
| Assumptions (requirements) | The primary and the secondary components shall be identical.<br>The primary and the secondary components shall be allocated to different hardware units.<br>The fault detector shall be verified. | TYPE_ASSUMPTION=are.identical;<br>COMPONENTS=[pr,se];<br>TYPE_ASSUMPTION=are.decoupled;<br>COMPONENTS=[pr,se];<br>TYPE_ASSUMPTION=are.verified;<br>COMPONENTS=[fd]; |

The assumptions (a.k.a. requirements) are not meant to be comprehensive

### SafPat

SafPat [26] is a command-line engine for recommending safety architecture patterns for addressing hardware and software faults in autonomous vehicle system architectures. SafPat has been implemented in DLV [27] (DataLog with Disjunction), a logic programming language based on ASP.

The inputs and outputs received and generated by SafPat are illustrated in Fig. 2. The inputs consists of the system architecture and safety artifacts computed during a safety analysis, such as HARA and STPA.

SafPat implements a domain-specific language (DSL) for embedded systems, thus enabling the specification of architecture elements such as components, channels, tasks, and hardware units. Safetywise, the DSL enables the

specification of safety artifacts (e.g., hazards and faults) and safety patterns.

The specification of safety patterns is the key point towards enabling their automated recommendation. In fact, the DSL enables the specification of, e.g., the intent and problem addressed by the safety pattern. Table 2 illustrates the specification of the Homogeneous Duplex pattern.

SafPat implements safety principles to reason about the safety of the system architecture. In particular, SafPat implements safety reasoning principles specifying the conditions for when safety patterns can be recommended to tolerate faults, and consequently address hazards. For example, the conditions for the Homogeneous Duplex pattern (homogeneousDuplex) are specified below. Consider a fault FT of type TYPE\_FAULT associated to an architecture element (e.g., an ECU), where FT triggers a failure FL of type TYPE\_FAIL, and FL leads to a hazard of ASIL TYPE\_ASIL.

1. homogeneousDuplex is suitable for tolerating the fault type TYPE\_FAULT;
2. homogeneousDuplex is suitable for avoiding the failure type TYPE\_FAIL;
3. homogeneousDuplex is suitable for addressing the hazard type TYPE\_ASIL.

Whenever such conditions hold, SafPat recommends the Homogeneous Duplex pattern. SafPat may recommend multiple solutions for addressing the same fault. For example, possible solutions for addressing hardware faults are the deployment of either the Homogeneous Duplex pattern or the Triple Modular Redundancy pattern. SafPat computes all possible solutions for addressing the safety artifacts received as input. Note that SafPat only outputs solutions where all safety artifacts (received as input) have been addressed, i.e., the hazards have been addressed through the use of safety patterns tolerating all faults leading to such hazards.

SafPat outputs requirements for each recommended pattern to ensure that the pattern works as intended. Examples of requirements are “the pattern components shall be allocated to an ASIL D hardware unit” and “the primary and the secondary (a.k.a. redundant) components shall be allocated to different hardware units”. SafPat also outputs requirements to ensure safety integrity with respect to the allocation of software components to hardware components. For example, consider an allocation of a software component SW1 into a hardware component HW1. SafPat’s reasoning principles check whether the ASIL of SW1 is higher than the ASIL of HW1. If this condition holds, SafPat provides a requirement to allocate SW1 to a hardware component with the same ASIL of SW1.

## Case Study

### Highway Pilot

The use case considered in this work is a Highway Pilot (HWP), implementing an SAE Level 3 Autonomous Driving (AD) function. In accordance to the “Safety First for Automated Driving” document [28], the nominal function is to “realize a trajectory within given limits derived from lane, other objects, and ego-vehicle width with the given and normal performing actuators”. We have modeled the HWP system architecture in AF3.

For the sake of simplifying the use case while retaining its architectural and functional complexity, the considered HWP implements three main functions: Adaptive Cruise Control (ACC), Lane Keeping Assistant (LKA), and Emergency Brake (EB). Whereas the ACC provides longitudinal control, the EB provides longitudinal control in case a collision with an obstacle in front of the vehicle is deemed unavoidable, and the LKA provides lateral control. In Fig. 3, we present the task architecture we modeled in AF3 for the considered system, describing 15 existing software tasks and the information flow among them in terms of signals. Furthermore, in Fig. 4, we show the platform architecture of the system modeled in AF3, specifying the execution units of the system, and the network topology. The ACC, LKA, and EB are represented as different software tasks, running on certain execution units. In addition, there is a task implementing the Sensor Data Fusion function, processing sensor data in order to compute the distance to obstacles, and the position of the ego vehicle within the lane, and a Coordinator task, which implements the decision logic. In Table 3, we present the allocation of 5 out of the 15 existing software tasks to the hardware units. In the same table, we also show the ASIL assigned to each component.

### Safety Analysis

To identify the faults within the system that may cause hazardous events, in compliance to ISO 26262, in the context of this work, we first conducted a hazard analysis and then a safety analysis. Based on the outcome of the conducted hazard analysis, we identified three hazards, namely: *HZ01* Unintended lane departures while the HWP system is engaged and executing a “lane-centering” maneuver, *HZ02* Violation of safe distance to front vehicle while the HWP system is engaged, *HZ03* Unintended emergency braking while the HWP system is engaged. After a risk assessment analysis, we assigned these hazards as ASIL D. The safety goal is often defined as the negation of the hazard, such as *SG01* Prevent unintended lane departures while the HWP system is engaged and executing a “lane-centering” maneuver.

To determine the potential causes of hazards, we conducted a System-Theoretic Process Analysis (STPA). As in this work, we propose a solution for automatic identification of safety patterns that can be applied to tolerate safety-critical software and hardware faults within a given system, while conducting the STPA analysis, we mostly focus on the identification of such faults. For each component of the HWP system, we identified loss scenarios that describe how hardware or software faults in the respective component may lead to a hazard. In Table 4, we present two exemplary loss scenarios specifying how hardware and software faults in the ACC component may lead to hazard *HZ02*. To tolerate such faults, in Sect. “Safety Pattern Synthesis”, we present an engine for safety pattern synthesis.

### Safety Pattern Synthesis

This section describes Safety Pattern Synthesis [3], a plugin to assist safety engineers in selecting safety architecture patterns. In particular, we describe the workflow of Safety Pattern Synthesis as well as its activity diagram by providing some implementation details. Safety Pattern Synthesis has been implemented as a plugin in the model-based system engineering tool AutoFOCUS3 (AF3) [4], in particular Safety Pattern Synthesis has been integrated into the DSE perspective of AF3. Safety Pattern Synthesis currently supports the following safety architecture patterns: Acceptance Voting pattern, Homogeneous Duplex pattern, Heterogeneous Duplex pattern, Simplex Architecture pattern, and Triple Modular Redundancy pattern.

### Workflow Overview

Figure 5 illustrates the workflow of Safety Pattern Synthesis. The first step is the design of the system architecture. This

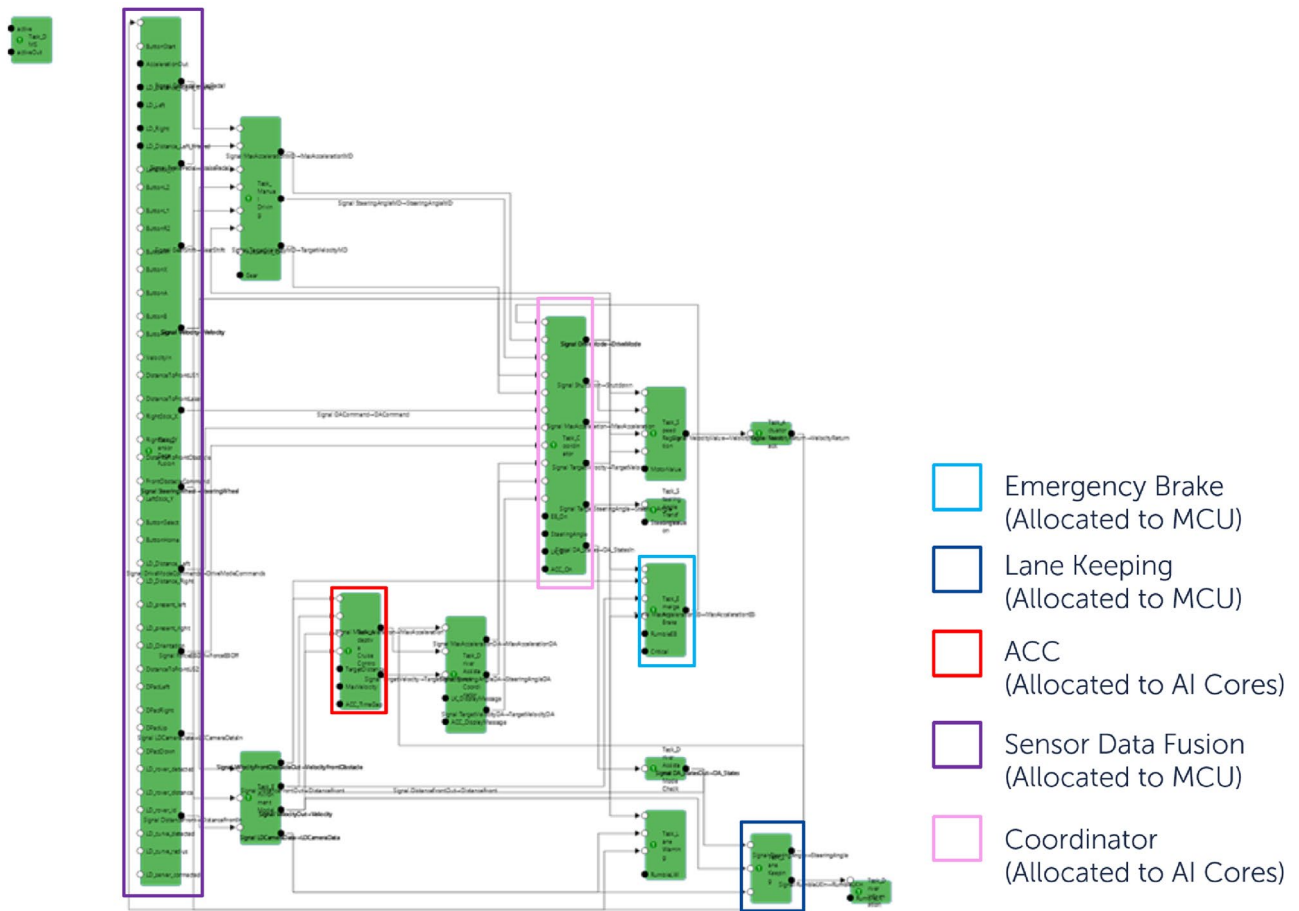


Fig. 3 Baseline task architecture of the HWP

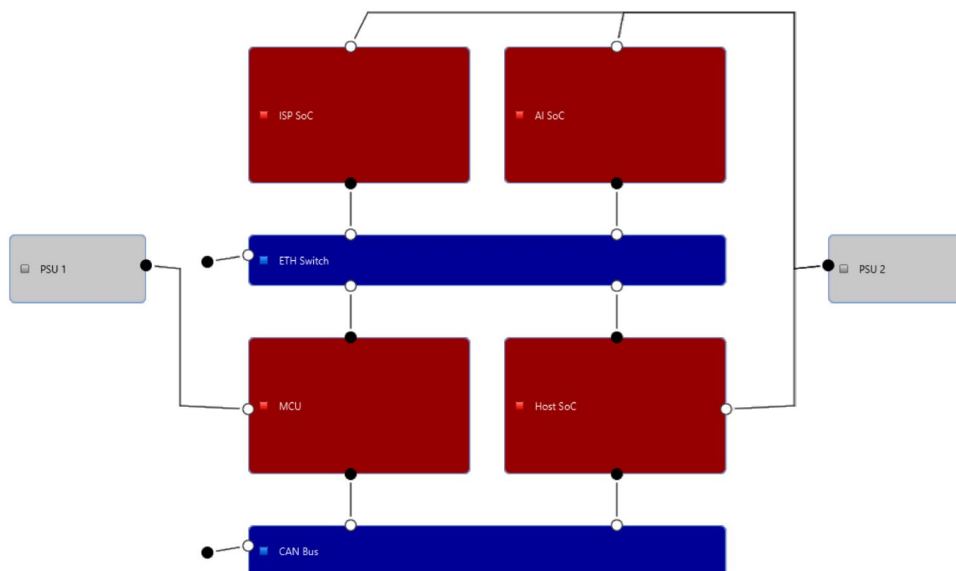


Fig. 4 Baseline platform architecture of the HWP

**Table 3** Allocation of tasks to hardware units

| Software task                    | Hardware unit                |
|----------------------------------|------------------------------|
| Adaptive Cruise Control [ASIL D] | Host SoC (AI Cores) [ASIL B] |
| Coordinator [ASIL D]             | Host SoC (AI Cores) [ASIL B] |
| Emergency Brake [ASIL D]         | MCU [ASIL D]                 |
| Lane Keeping [ASIL D]            | MCU [ASIL D]                 |
| Sensor Data Fusion [ASIL D]      | MCU [ASIL D]                 |

**Table 4** Exemplary loss scenario in adaptive cruise control (ACC)

| Component, fault | Loss scenario  |
|------------------|--|
| ACC, HW Fault    | When the HWP is enabled and the relative velocity and distance to an obstacle mean that a collision is imminent, a hardware fault occurs in the hardware unit to which the task implementing the ACC function is allocated, causing the target deceleration value needed to prevent a front-end collision not to be provided by the ACC task |
| ACC, SW Fault    | When the HWP is enabled and the relative velocity and distance to an obstacle mean that a collision is imminent, a software fault of type ERRONEOUS ALGORITHM in the task implementing the ACC function causes that the ACC provides a target deceleration value not high enough to prevent a front-end collision                            |

step is usually done by a system architect. Figure 5 illustrates a task architecture (a.k.a. software architecture) with tasks and signals connecting the tasks. The second step is to annotate the system architecture with safety-critical faults identified during the system safety analysis. Figure 5 illustrates a software fault in Task 3. **Safety Pattern Synthesis** receives as input both the system architecture and safety artifacts such as faults. **Safety Pattern Synthesis** automates the third step by recommending and instantiating suitable safety architecture patterns in the architecture. Figure 5 illustrates the instantiation of the Heterogeneous Duplex pattern (see tasks and signals in light green) for tolerating the software fault in Task 3. **Safety Pattern Synthesis** may recommend several solutions with different options for safety architecture patterns. The user

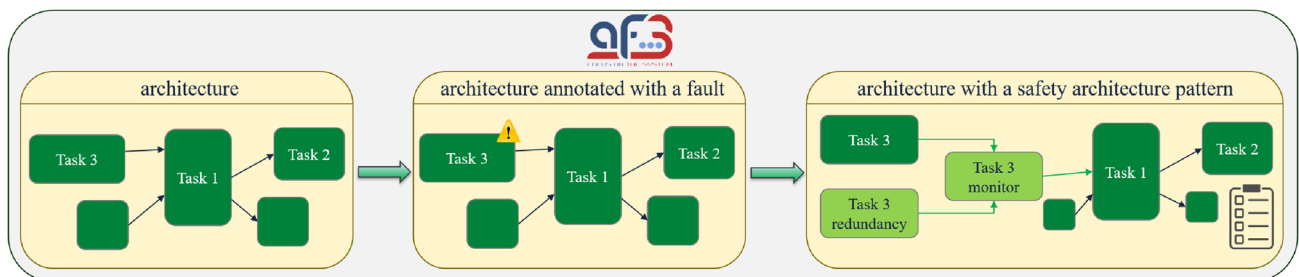
of **Safety Pattern Synthesis** shall select the most suitable solution for the system. As described in "**Activity Diagram and Implementation Details**", **Safety Pattern Synthesis** provides four criteria to assist the user with this decision. **Safety Pattern Synthesis** also provides requirements to ensure that implementation of each safety architecture pattern works as intended. These requirements shall be realized during the system development.

**Activity Diagram and Implementation Details**

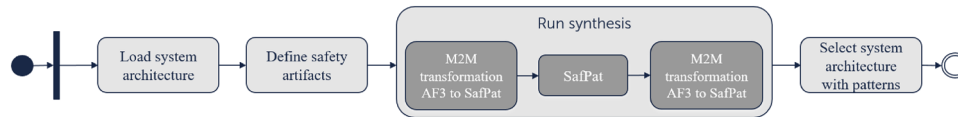
The implementation of **Safety Pattern Synthesis** consists of different modules implemented in Java. The exception is the **SafPat** module. **SafPat** [26] is the backend of the plugin that recommends safety architecture patterns. As described in Sect. "**SafPat**", **SafPat** has been implemented in DLV. We use the EmbASP framework [29] for integrating **SafPat** in **Safety Pattern Synthesis**.

The activity diagram of **Safety Pattern Synthesis** is illustrated in Fig. 6. For the sake of presentation, Fig. 6 also illustrates three important modules of **Safety Pattern Synthesis**, including **SafPat** (see dark gray boxes).

- **Load system architecture.** The user loads the designed system architecture in **Safety Pattern Synthesis**. The system architecture shall contain at least the following viewpoints: Logical architecture, task architecture, platform architecture, and allocation (a.k.a. deployment) entries. Optionally, the user may also assign ASILs (according to ISO 26262) to either components of the logical architecture or execution units of the platform architecture. We assume that ASILs assigned to components are safety requirements, e.g., component X shall be implemented with ASIL D requirements, while ASILs assigned to execution units are implemented safety requirements, e.g., execution unit X has been implemented with ASIL D requirements. **SafPat** takes this information into account when providing requirements.
- **Define safety artifacts.** The user defines in **Safety Pattern Synthesis** the safety artifacts obtained from the safety analysis. Each safety artifact shall contain the



**Fig. 5** Workflow of Safety Pattern Synthesis



**Fig. 6** Activity diagram of Safety Pattern Synthesis

following information: A hazard, fault(s) and failure(s). First, the user defines a hazard by providing a short description and assigning the ASIL of the hazard. Secondly, the user defines one or more faults associating each fault with an architecture element. To this end, **Safety Pattern Synthesis** provides a user interface that lists all tasks from the task architecture (software fault) and all execution units from the platform architecture (hardware fault). For each defined fault, the user shall define the failure type triggered by the fault. **Safety Pattern Synthesis** considers failures of type erroneous and loss. The user may read a defined safety artifact in **Safety Pattern Synthesis** as follows: *a fault (possibly more than one) triggers a failure that may lead to a hazard.*

- **Run synthesis.** Upon loading the system architecture and defining the safety artifacts, the user runs the plugin. Figure 6 illustrates the three main important modules of **Safety Pattern Synthesis** to enable the recommendation of safety architecture patterns.
  - *M2M transformation AF3 to SafPat.* This module implements a model-to-model transformation from AF3 to **SafPat**. That is, it transforms the model elements specified in the AF3 's language to the domain-specific language specified in **SafPat**. The transformation is implemented with the help of the EmbASP framework.
  - **SafPat.** The backend module implements reasoning rules to enable the recommendation of safety architecture patterns to address the safety artifacts received as input. The safety reasoning rules specify the conditions for when safety architecture patterns may be used to tolerate faults, and where such patterns shall be deployed in the system architecture. **SafPat** may recommend several different architecture solutions with patterns where all safety artifacts (received as input) have been addressed. **SafPat** also provides requirements for the recommended patterns (to ensure they work as intended), and for checking the safety integrity with respect to allocations (by checking the ASILs assigned to architecture elements).
  - *M2M transformation SafPat to AF3.* This module implements a model-to-model transformation from **SafPat** to AF3. That is, it transforms the architecture solutions with patterns specified in **SafPat** to AF3 's model elements. This transformation enables, e.g., the instantiation of safety architecture patterns in an automated fashion. The transformation is implemented with the help of EmbASP.
- **Select system architecture with patterns.** The user selects the system architecture from the architecture solutions recommended by **SafPat**. **Safety Pattern Synthesis** provides four criteria to help the user with this decision [3]:
  - *Number of new components* describes the number of diverse redundant components required by the safety architecture pattern. For example, Heterogeneous Duplex pattern requires one redundant diverse component. The user can make a decision based on, e.g., the number of diverse components to be developed.
  - *Number of replica components* describes the number of identical redundant components required by the safety architecture pattern. A user can make a decision based on, e.g., the number of replica that need to be introduced.
  - *Number of pattern support components* describes the number of non-application components required by the safety architecture pattern. Examples of non-application components are the monitor (a.k.a. fault detector) of the Heterogeneous Duplex pattern and the voter of the Triple Modular Redundancy pattern. A user can make a decision based on, e.g., the overhead introduced by the pattern and on the number of pattern support components to be developed.
  - *Simplified (a.k.a. degraded)* describes whether the modified architecture with patterns contains a simplified application component. For example, the diverse redundant component of the Simplex Architecture pattern is a simpler version of the primary component. A user can make a decision based on, e.g., the minimum available fidelity level of components.

The chosen architecture is exported by the user. The exported architecture solution consists of a modified baseline architecture with the recommended safety archi-

**Table 5** Solution with one instance of SA, three instances of HtD and one instance of AV

| Pattern | Task               | Description  |
|---------|--------------------|--|
| SA      | Sensor Data Fusion | SA for tolerating the software fault in Sensor Data Fusion. To this end, a safe and simple (a.k.a. degraded) version of Sensor Data Fusion shall be developed. A monitor shall be developed to identify faults in Sensor Data Fusion, including a timeout algorithm to detect faults triggering failures of type loss. If one fault is detected in Sensor Data Fusion, the Monitor switches to the safe and simple Sensor Data Fusion task. Requirements are created by <b>Safety Pattern Synthesis</b> to tolerate hardware faults such as the tasks implementing Sensor Data Fusion shall be allocated to different hardware units   |
| HtD     | ACC                | HtD for tolerating software faults in ACC. To this end, a redundant version of ACC shall be developed using a different software design, here called ACC V2. A Fault Detector (a.k.a. monitor) shall also be developed to identify faults, including a timeout algorithm to detect faults triggering failures of type loss. If one fault is detected in ACC, the Monitor switches to ACC V2 task. Requirements are created by <b>Safety Pattern Synthesis</b> to tolerate hardware faults by having ACC and ACC V2 allocated to different hardware units   |
| HtD     | LKA                | Similar to the description above about ACC   |
| HtD     | EB                 | Similar to the description above about ACC   |
| AV      | Coordinator        | AV for tolerating software faults in Coordinator. To this end, two additional diverse redundant implementations of the Coordinator shall be developed, here called Coordinator V2 and Coordinator V3. For each Coordinator task, a monitor shall be developed (using, e.g., plausibility checks) to identify faults in the outputs generated by the Coordinator. Each monitor shall also implement a timeout algorithm to detect failures of type loss. If the output is accepted by the Monitor (e.g., passes the plausibility check), the output is forwarded to a Voter, which selects the resulting output based on a voter policy such as majority. In order to also tolerate hardware faults, <b>Safety Pattern Synthesis</b> generates requirements that demand that Coordinator, Coordinator V2, Coordinator V3 shall be allocated to different hardware units |

ture patterns. In particular, **Safety Pattern Synthesis** modifies the task architecture by instantiating each of the recommended patterns. **Safety Pattern Synthesis** also exports a list of requirements that shall be realized to ensure (a) that recommended patterns work as intended and (b) the safety integrity with respect to allocations.

## Results

We run **Safety Pattern Synthesis** to recommend safety patterns to address the identified safety artifacts for the Highway Pilot system architecture.

**Safety Pattern Synthesis** computed **94** architecture solutions. **Safety Pattern Synthesis** computed such solutions in less than 10 min on an Intel Core i7-6700 (2.6 GHz) with 16 Gb RAM running Ubuntu Linux 20.4 LTS (in VirtualBox with VT-X/AMD-V virtualization and KVM para-virtualization). The computed solutions include several mixes of three safety architecture patterns, namely, Heterogeneous Duplex (HtD) pattern, Simplex Architecture (SA) pattern, and Acceptance Voting (AV) pattern. Table 5 shows one example solution where these three patterns are recommended.

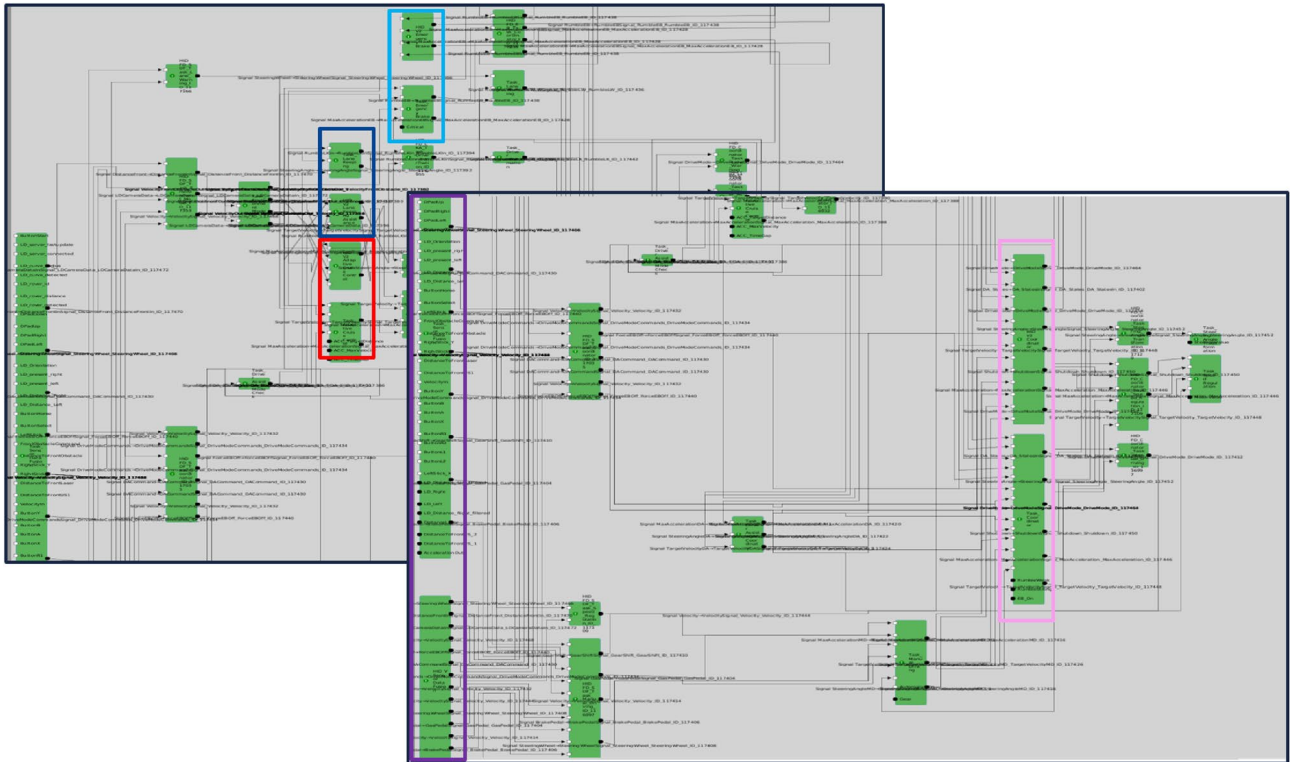
Figure 7 illustrates the exported AF3 model of the architecture solution with five instances of the HtD pattern that have been recommended to tolerate the identified hardware and software faults. Figure 8 illustrates an excerpt of the 45 requirements that have been generated for the above solution that contains five instances of the

HtD pattern. **Safety Pattern Synthesis** makes explicit via requirements when the safety integrity with respect to allocations is violated. Requirements **SafPat-14** and **SafPat-20** describe that the ACC and Coordinator tasks shall be allocated to ASIL D hardware units, instead of ASIL B hardware units as shown in Table 3.

## Feature Degradation Synthesis

This section describes **Feature Degradation Synthesis**, an automated workflow to analyze the degradation of functional features in failure scenarios related to hardware faults of execution units based on a formal model of the system. **Feature Degradation Synthesis** determines, based on a valid initial redundant task-to-execution unit allocation, alternative potentially degraded allocations for failure scenarios stemming from hardware fault of execution units. For the overall workflow of the described tool-chain, feature **Feature Degradation Synthesis** builds on the Heterogeneous Duplex Pattern (HtD) that may be instantiated by **Safety Pattern Synthesis** as described in Sect. "**Safety Pattern Synthesis**" as follows: For all instance of the HtD pattern in the given architecture, **Feature Degradation Synthesis** is used to determine the switching logic of the pattern's fault detector, i.e. when to switch between the primary to the secondary replica.





**Fig. 7** HWP architecture with five instances of the HtD pattern: Light blue highlights the instance for the EB task, dark blue highlights the instance for the LKA task, red highlights the instance for the ACC

task, purple highlights the instance for the Sensor Data Fusion task, and pink highlights the instance for the Coordinator task

## Workflow Overview

The approach underlying the presented tool-supported workflow is based on the work by Becker et al. [30]. It relies on an AutoFOCUS3 [4] input model that comprises the functional features of a system, software components realizing these features and an initial (redundant) deployment of the respective tasks to the hardware platform's execution units. The output is a reconfiguration graph that specifies the relocation of tasks for the different failure scenarios.

**Input model** Feature Degradation Synthesis relies on the following information that has to be provided in the input model:

- System Architecture with instantiated HmD patterns (generated by Safety Pattern Synthesis)
- Hardware failures of execution units (from safety analysis, see Sect. "Safety Analysis")
- Initial task-to-execution unit allocation with initial task deployment types (master/ replica)
- Priorities (penalty for function unavailability, reconfiguration cost)

In the presented tool-chain, the above information is provided in an AutoFOCUS3 model. The model is defined

in terms of the viewpoints introduced in Sect. "AutoFOCUS3" as well as the following extensions:

The *functional viewpoint* enables the specification of system functions, as well as the decomposition of functions into subfunctions in terms of a *Function Architecture*. Atomic functions are mapped to one or more components of the logical viewpoint, specifying concrete designs for each function in terms of component networks in the logical architecture. As shown in Fig. 9, in the presented toolchain, each function in the *Function Architecture* is annotated with following cost metrics related to the reconfiguration of the system:

- Reconfiguration cost: Cost for relocating a task related to this function from one execution unit to another
- Function unavailability penalty: Measure for the importance of the function (e.g., utility of for the user, criticality).

In order to enable the reconfiguration of the system in the event of execution unit failures, Feature Degradation Synthesis requires the redundant deployment of software tasks to be present in the initial system configuration. The allocation of active tasks (master) and their replica (standby) to execution units is defined in terms of a task redundancy mapping. Figure 10 shows an exemplary

| Type        | Status     | ID        | Title  |
|-------------|------------|-----------|--|
| Requirement | Identified | SafPat-00 | HtD Fault Detector_1: are_allocated_to_consumer_side                                 |
| Requirement | Identified | SafPat-01 | Task_Sensor_Data_Fusion_ID_145476: are_allocated_to_ASIL_D_hardware_unit             |
| Requirement | Identified | SafPat-02 | [[nuHTDSEC_4_HtD Fault Detector_4]]: are_developed                                   |
| Requirement | Identified | SafPat-03 | [[nuHTDSEC_3_HtD Fault Detector_3]]: are_developed_with_ASIL_D                       |
| Requirement | Identified | SafPat-04 | ["Task_Lane_Keeping_ID_145757",HtD V2_3]]: are_hardware_decoupled                    |
| Requirement | Identified | SafPat-05 | HtD Fault Detector_0: are_verified   |
| Requirement | Identified | SafPat-06 | HtD Fault Detector_2: events_are_time_triggered                                      |
| Requirement | Identified | SafPat-07 | ["Task_Sensor_Data_Fusion_ID_145476",HtD V2_1]]: are_implement_with_different_design |
| Requirement | Identified | SafPat-08 | ["Task_Sensor_Data_Fusion_ID_145476",HtD V2_1]]: are_hardware_decoupled              |
| Requirement | Identified | SafPat-09 | HtD Fault Detector_4: events_are_time_triggered                                      |
| Requirement | Identified | SafPat-10 | [[nuHTDSEC_1_HtD Fault Detector_1]]: are_developed                                   |
| Requirement | Identified | SafPat-11 | ["Task_Coordinator_ID_145374",HtD V2_0]]: are_implement_with_different_design        |
| Requirement | Identified | SafPat-12 | HtD Fault Detector_1: events_are_time_triggered                                      |
| Requirement | Identified | SafPat-13 | [[nuHTDSEC_4_HtD Fault Detector_4]]: are_developed_with_ASIL_B                       |
| Requirement | Identified | SafPat-14 | Task_Adaptive_Cruise_Control_ID_145725: are_allocated_to_ASIL_D_hardware_unit        |
| Requirement | Identified | SafPat-15 | HtD Fault Detector_4: are_allocated_to_consumer_side                                 |
| Requirement | Identified | SafPat-16 | HtD Fault Detector_3: are_allocated_to_consumer_side                                 |
| Requirement | Identified | SafPat-17 | ["Task_Emergency_Brake_ID_145777",HtD V2_4]]: are_implement_with_different_design    |
| Requirement | Identified | SafPat-18 | HtD Fault Detector_2: are_verified   |
| Requirement | Identified | SafPat-19 | ["Task_Lane_Keeping_ID_145757",HtD V2_3]]: are_implement_with_different_design       |
| Requirement | Identified | SafPat-20 | Task_Coordinator_ID_145374: are_allocated_to_ASIL_D_hardware_unit                    |
| Requirement | Identified | SafPat-21 | HtD Fault Detector_0: are_allocated_to_consumer_side                                 |
| Requirement | Identified | SafPat-22 | [[nuHTDSEC_2_HtD Fault Detector_2]]: are_developed                                   |
| Requirement | Identified | SafPat-23 | HtD Fault Detector_3: events_are_time_triggered                                      |
| Requirement | Identified | SafPat-24 | HtD Fault Detector_4: are_verified   |

Fig. 8 Excerpt of requirements generated for solution with five instances of the HtD pattern

task redundancy mapping where a *Task A* is allocated as active master on execution unit *Generic\_ECU1*, and its replica *Task A'* is allocated as passive replica on execution unit *Generic\_ECU2*. *Fault Detector* is the pattern support component of the HtD pattern that determines whether to switch from *Task A* to *Task A'* (e.g., to safeguard the architecture against hardware faults of *Generic\_ECU1*).

**Output model** To capture the output of the feature degradation synthesis, we build on the approach introduced in [8] and integrate a metamodel for reconfiguration graphs into the AutoFOCUS3 tool. A reconfiguration graph contains one *Configuration* node for each failure scenario to consider all combinations of hardware failures of the execution units (i.e., in an architecture with *n* execution units, there are  $2^n$  failure scenarios). The edges of the reconfiguration graph reference the execution unit whose failure triggers the transition from the source to the target *Configuration*.

Each of the *Configurations* contains a degraded deployment of tasks (as master or as standby) to the execution units that are still available in the respective failure scenario.

Figure 11 shows the reconfiguration graph for the example architecture with two execution units. The task deployment of the initial *Configuration* < 11 >, where all execution units are still available, is identical to the initial task redundancy mapping shown in Fig. 10. Figure 12 exemplarily shows the degraded task deployment for *Configuration* < 01 > where

| Model Element | Comment | Reconfiguration Cost | Unavailability Cost |
|---------------|---------|----------------------|---------------------|
| Function A    |         | 1                    | 4                   |
| Function B    |         | 5                    | 1                   |

Fig. 9 Function Architecture with reconfiguration cost and function unavailability penalty annotation

| Src            | Tgt     | Task Type    |
|----------------|---------|--------------|
| Fault Detector |         | Generic_ECU1 |
| Task A         | MASTER  |              |
| Task A'        | STANDBY |              |
| Task B         | MASTER  |              |

Fig. 10 Task redundancy mapping

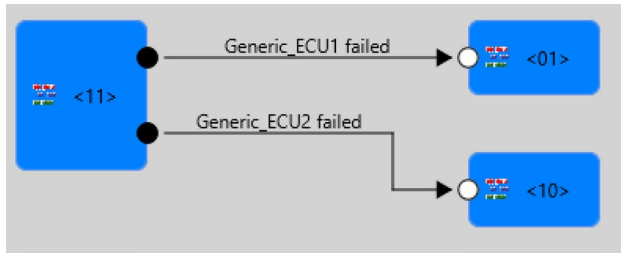


Fig. 11 Reconfiguration graph for an architecture with two execution units

| Src            | Tgt    | Task Type    |
|----------------|--------|--------------|
| Fault Detector |        | Generic_ECU1 |
| Task A         |        |              |
| Task A'        | MASTER |              |
| Task B         |        |              |

Fig. 12 Degraded task deployment for failure scenario < 01 >

execution unit *Generic\_ECU1* has failed (see edge from *Configuration < 11 >* to *Configuration < 11 >*). In *Configuration < 01 >*, *Task\_B* has been dropped to free resources favor of the replica *Task A'* (that has a higher unavailability penalty than *Task B*, see Fig. 9).

**Feature degradation synthesis AutoFOCUS3 extension**

For the presented toolchain, the AutoFOCUS3 [4] tool has been extended with a Feature Degradation Synthesis extension that has been integrated as an optimizing exploration into the DSE perspective. After the definition of the input model, the feature degradation synthesis can be launched as shown in Fig. 13.

After selecting the Z3Reconfiguration (SMT) solver type at the bottom of the optimization section, the Reconfiguration synthesis type can be enabled at the top of the synthesis pane. By means of the Optimize button, the feature degradation synthesis is started. It translates the input model into a Z3 formulization based on the work of [30], where it uses constraints to specify valid degradations of the system, e.g.:

- Initial deployment constraint: constraint encoding the input model
- No migration constraint: constraint ensuring that software tasks may only become a master on execution where they have initially been deployed (as master or replica)
- Single master active constraint: constraint ensuring that in each failure scenario, a task is deployed as a master at most once.

The synthesis is guided by the minimization of two objectives, namely

- Unavailability cost: cost for dropping tasks in a given failure scenario
- Reconfiguration cost: cost for activating task replica.

After the exploration has terminated, the results are converted back into AutoFOCUS3 models which may be exported to the original project.

**Results**

For evaluation, we consider the HWP architecture introduced in Sect. "Case Study" and apply Feature Degradation Synthesis to that solution yielded by Safety Pattern Synthesis that contains 5 instances of the HtD pattern (see Fig. 7). In the following, we focus on synthesizing the switching logic for the fault detectors that have been introduced for the Adaptive Cruise Control, the Emergency Brake, and the Coordinator task in order to tolerate hardware faults of type loss of the execution units *Core 1*, *Core 3*, and *Core 4* (contained by the *MCU*). The *DMS* task represents the implementation of a Driver Monitoring System function that may be dropped (possibly in conjunction with appropriate additional measures) in order to free computational resources for the tasks of higher criticality mentioned before.

In alignment with the deployment constraints resulting from the application of the selected patterns, we define the following task redundancy table for the underlying tasks and execution units.

In order to apply the Feature Degradation Synthesis, an initial task-to-hardware allocation has to be provided that defines on which execution unit task are initially active in the undegraded state of the system. Therefore, we extended the allocation defined in the baseline model such that the requirements on the deployment resulting from the application of safety patterns (see Fig. 8 are satisfied). Figure 14 shows the resulting task redundancy table where an active (master) replica and a passive (backup) replica of each consider task are allocated to different execution units.

In the next step, the model is imported into the Design Space Exploration perspective of AutoFOCUS3 and the

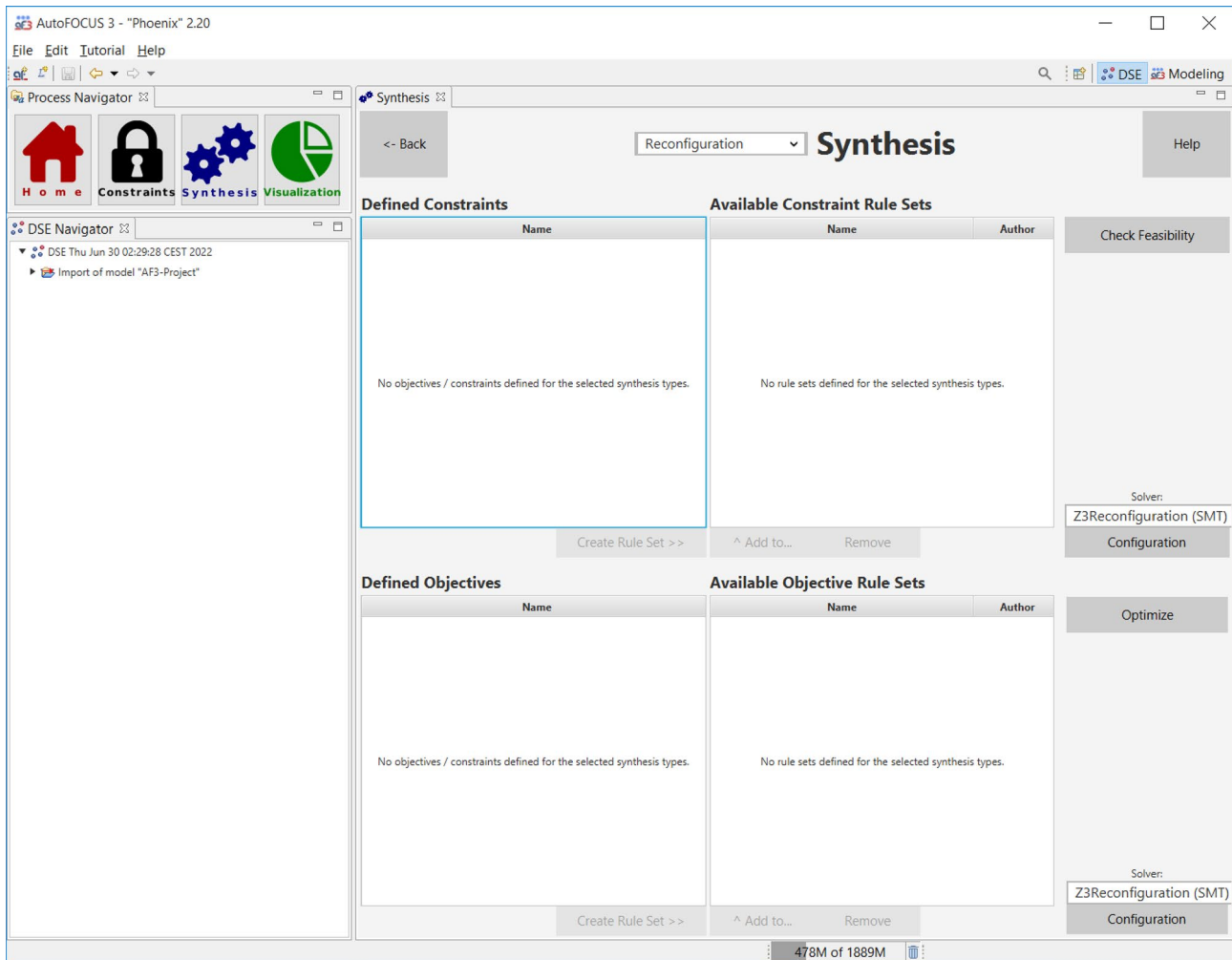


Fig. 13 Integration of Feature Degradation Synthesis into AutoFOCUS3 DSE perspective

Feature Degradation Synthesis is executed. As explained in Sect. "Workflow Overview", in this process, the AutoFOCUS3 model and the objective functions to maximize the utility of degraded configurations of the system while minimizing the reconfiguration cost is transformed into a constraint satisfaction problem. This formal representation is then passed to the Z3 SMT solver that determines the task-to-hardware allocations for all failure scenarios of the considered execution units.

Figure 15 exemplarily shows one the determined reconfiguration graphs that specify the switching logic for the considered fault detector component.

Each of the nodes of the graph defines a failure scenario that defines the degradation state of the tasks realizing the considered functions depending on the available execution units.

Configuration <111> corresponds to the initial system state where the execution units *Core 1*, *Core 3*, and *Core 4* of the *MCU* are healthy. In each of the configurations

<011>, <101>, and <110>, the execution unit depicted by the incoming edge has failed. Figure 16 exemplarily depicts the degraded task redundancy table for configuration <011> where *Core 1* of the *MCU* has failed.

In each of the subsequent configurations <100>, <010>, and <001>, a total of two execution units has failed. Since the safety pattern synthesis had constructed the underlying architecture based on a one fault assumption, in each of these configurations a critical task of the nominal channel would have to be dropped. Therefore, these configurations mark system states where nominal channel becomes unavailable, and the system has to switch to the degraded channel to meet the fail-operational requirements of the HWP functionality. Assuming sufficient independence between the execution units involved in the implementation of the feature degradation strategy, and therefore a very low probability of a dual hardware fault, the approach is considered as a good compromise between increase of the nominal channel's availability (i.e., in the case of single hardware faults), and

| Src   Tgt →   | Task Type: Core 1 | TaskType: Core 2 | Task Type: Core 3 | Task Type: Core 4 |
|---|-------------------|------------------|-------------------|-------------------|
| HTD Fault Detector_0_Task_Driver_Assistance_Mode_Check_ID_153526  |                   |                  |                   |                   |
| HTD Fault Detector_0_Task_Emergency_Brake_ID_153589               |                   |                  |                   |                   |
| HTD Fault Detector_0_Task_Lane_Warning_ID_153615                  |                   |                  |                   |                   |
| HTD Fault Detector_0_Task_Manual_Driving_ID_153148                |                   |                  |                   |                   |
| HTD Fault Detector_0_Task_Speed_Regulation_ID_153254              |                   |                  |                   |                   |
| HTD Fault Detector_0_Task_Steering_Angle_Transformation_ID_153277 |                   |                  |                   |                   |
| HTD Fault Detector_1_Task_Coordinator_ID_153186                   |                   |                  |                   |                   |
| HTD Fault Detector_1_Task_Environment_Model_ID_153449             |                   |                  |                   |                   |
| HTD Fault Detector_1_Task_Lane_Warning_ID_153615                  |                   |                  |                   |                   |
| HTD Fault Detector_1_Task_Manual_Driving_ID_153148                |                   |                  |                   |                   |
| HTD Fault Detector_1_Task_Speed_Regulation_ID_153254              |                   |                  |                   |                   |
| HTD Fault Detector_2_Task_Driver_Assistance_Coordinator_ID_153486 |                   |                  |                   |                   |
| HTD Fault Detector_3_Task_Driver_Assistance_Coordinator_ID_153486 |                   |                  |                   |                   |
| HTD Fault Detector_3_Task_Driver_Information_ID_153515            |                   |                  |                   |                   |
| HTD Fault Detector_4_Task_Coordinator_ID_153186                   |                   |                  |                   |                   |
| Task_Actuator Feedback  |                   |                  |                   |                   |
| Task_Adaptive Cruise Control                                      | MASTER            |                  |                   |                   |
| Task_Adaptive Cruise Control V2_2                                 |                   |                  |                   | STANDBY           |
| Task_Coordinator  |                   |                  |                   | MASTER            |
| Task_Coordinator V2_0   |                   |                  | STANDBY           |                   |
| Task_DMS  |                   |                  |                   |                   |
| Task_Driver Assistance Coordinator                                |                   |                  |                   |                   |
| Task_Driver Assistance Mode Check                                 |                   |                  |                   |                   |
| Task_Driver Information   |                   |                  |                   |                   |
| Task_Emergency Brake  |                   |                  | MASTER            |                   |
| Task_Emergency Brake V2_4   | STANDBY           |                  |                   |                   |
| Task_Environment Model  |                   |                  |                   |                   |
| Task_Lane Keeping   |                   |                  |                   |                   |
| Task_Lane Keeping V2_3  |                   |                  |                   |                   |
| Task_Lane Warning   |                   |                  |                   |                   |
| Task_Manual Driving   |                   |                  |                   |                   |
| Task_Sensor Data Fusion   |                   |                  |                   |                   |
| Task_Sensor Data Fusion V2_1                                      |                   |                  |                   |                   |
| Task_Speed Regulation   |                   |                  |                   |                   |
| Task_Steering Angle Transformation                                |                   |                  |                   |                   |

Fig. 14 Initial task redundancy table defining redundant deployment of Adaptive Cruise Control, Emergency Brake, and Coordinator task

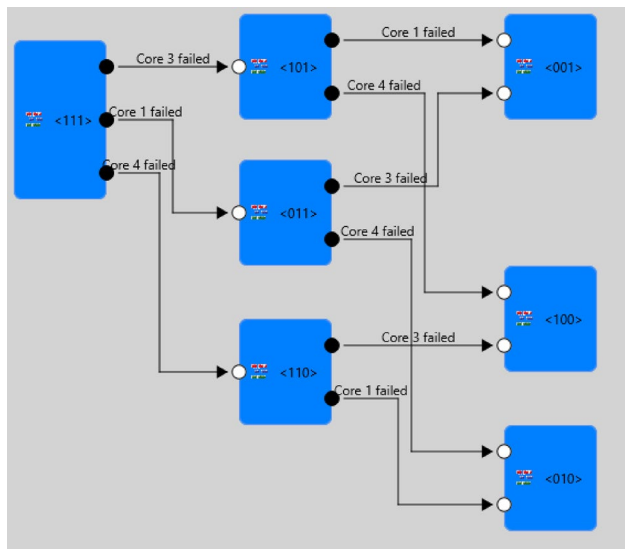


Fig. 15 Reconfiguration graph defining the switching logic of the Fault Detector tasks of the HTD patterns that instantiated for the Adaptive Cruise Control, Emergency Brake, and Coordinator task

the required redundancy (and therefore resource consumption and cost).

### Simulation

ISO 26262 recommends simulations as complementary method to provide evidence for the consistency and compliance of safety concepts with the safety goals and for their ability to avoid or mitigate hazards in both the design (cf. ISO 26262-3:2018) and the development (cf. ISO 26262-4:2018) phases. The standard also requires the verification of the functional performance, accuracy and timing of safety mechanisms. The here presented workflow includes simulation-based fault injection testing activities to examine real-time properties of the generated safety architecture candidates generated by Safety Pattern Synthesis and Feature Degradation Synthesis.

Section "Requirements" discusses the requirements for a simulation-based fault-injection framework used for the validation of safety architectures. Section "Approach" suggests a suitable approach for which Sect. "Implementation" introduces the tool support necessary for the evaluation of the generated safety concepts described in Sect. "Results".

| L Src.   Tgt. →   | ■ | ■ | ■ | ■ | ■ TaskType: Core 1 | ■ TaskType: Core 2 | ■ Task Type: Core 3 | ■ Task Type: Core 4 |
|---|---|---|---|---|--------------------|--------------------|---------------------|---------------------|
| HTD Fault Detector_0_Task_Driver_Assistance_Mode_Check_ID_153526  |   |   |   |   |                    |                    |                     |                     |
| HTD Fault Detector_0_Task_Emergency_Brake_ID_153589               |   |   |   |   |                    |                    |                     |                     |
| HTD Fault Detector_0_Task_Lane_Warning_ID_153615                  |   |   |   |   |                    |                    |                     |                     |
| HTD Fault Detector_0_Task_Manual_Driving_ID_153148                |   |   |   |   |                    |                    |                     |                     |
| HTD Fault Detector_0_Task_Speed_Regulation_ID_153254              |   |   |   |   |                    |                    |                     |                     |
| HTD Fault Detector_0_Task_Steering_Angle_Transformation_ID_153277 |   |   |   |   |                    |                    |                     |                     |
| HTD Fault Detector_1_Task_Coordinator_ID_153186                   |   |   |   |   |                    |                    |                     |                     |
| HTD Fault Detector_1_Task_Environment_Model_ID_153449             |   |   |   |   |                    |                    |                     |                     |
| HTD Fault Detector_1_Task_Lane_Warning_ID_153615                  |   |   |   |   |                    |                    |                     |                     |
| HTD Fault Detector_1_Task_Manual_Driving_ID_153148                |   |   |   |   |                    |                    |                     |                     |
| HTD Fault Detector_1_Task_Speed_Regulation_ID_153254              |   |   |   |   |                    |                    |                     |                     |
| HTD Fault Detector_2_Task_Driver_Assistance_Coordinator_ID_153486 |   |   |   |   |                    |                    |                     |                     |
| HTD Fault Detector_3_Task_Driver_Assistance_Coordinator_ID_153486 |   |   |   |   |                    |                    |                     |                     |
| HTD Fault Detector_3_Task_Driver_Information_ID_153515            |   |   |   |   |                    |                    |                     |                     |
| HTD Fault Detector_4_Task_Coordinator_ID_153186                   |   |   |   |   |                    |                    |                     |                     |
| Task_Actuator_Feedback  |   |   |   |   |                    |                    |                     |                     |
| Task_Adaptive_Cruise_Control                                      |   |   |   |   |                    |                    |                     |                     |
| Task_Adaptive_Cruise_Control_V2_2                                 |   |   |   |   |                    |                    |                     | MASTER              |
| Task_Coordinator  |   |   |   |   |                    |                    |                     | MASTER              |
| Task_Coordinator_V2_0   |   |   |   |   |                    |                    | STANDBY             |                     |
| Task_DMS  |   |   |   |   |                    |                    |                     |                     |
| Task_Driver_Assistance_Coordinator                                |   |   |   |   |                    |                    |                     |                     |
| Task_Driver_Assistance_Mode_Check                                 |   |   |   |   |                    |                    |                     |                     |
| Task_Driver_Information   |   |   |   |   |                    |                    |                     |                     |
| Task_Emergency_Brake  |   |   |   |   |                    |                    | MASTER              |                     |
| Task_Emergency_Brake_V2_4   |   |   |   |   |                    |                    |                     |                     |
| Task_Environment_Model  |   |   |   |   |                    |                    |                     |                     |
| Task_Lane_Keeping   |   |   |   |   |                    |                    |                     |                     |
| Task_Lane_Keeping_V2_3  |   |   |   |   |                    |                    |                     |                     |
| Task_Lane_Warning   |   |   |   |   |                    |                    |                     |                     |
| Task_Manual_Driving   |   |   |   |   |                    |                    |                     |                     |
| Task_Sensor_Data_Fusion   |   |   |   |   |                    |                    |                     |                     |
| Task_Sensor_Data_Fusion_V2_1                                      |   |   |   |   |                    |                    |                     |                     |
| Task_Speed_Regulation   |   |   |   |   |                    |                    |                     |                     |
| Task_Steering_Angle_Transformation                                |   |   |   |   |                    |                    |                     |                     |

Fig. 16 Task redundancy table for configuration <011> where Core 1 of the MCU has failed

### Requirements

An approach for simulation-based fault injection testing of safety mechanisms must have the ability to verify if a safety concept mitigates faults in both software and hardware. Therefore, a purley logical simulation is not sufficient: The simulation needs to consider properties of the underlying platform architecture that are relevant for the injection, containment, and propagation of faults. Such information, however, can only be leveraged if the allocation of software units (to partitions and processing elements) and signals (to transmission units) is taken into account as well.

Other requirements address the practical applicability of the approach, and stem particularly from the distributed development encountered across automotive supply chains: The necessity to exchange behavior specifications for the purpose of testing across organizations must not obstruct the protection of Intellectual Property (IP). Moreover, the heterogeneous tool landscape found in CPS engineering demands a high level of interoperability to reduce the overhead incurred by the testing activities, which, according to ISO 26262-3:2018, shall be performed recurringly throughout development.

### Approach

The co-simulation-based fault injection framework we introduced in [31] has been defined specifically for the early validation of system-level safety mechanisms. However, while that methodological framework addresses all requirements mentioned in Sect. "Requirements", no tool support is yet available.

In [31], we propose a co-simulation based on the Functional Mock-up Interface<sup>1</sup> (FMI) capable of accurately simulating the loss and erroneous behavior of both hardware and software components as well as the propagation of such faults. Hereby, each Functional Mock-up Unit (FMU) (representing a software unit) is executed by a standalone ROS<sup>2</sup> node. Hypervisor partitions and, in turn, processing elements are encoded by logically grouping these ROS nodes according to the allocations of software units. In contrast to prevalent FMI-based co-simulations, this ROS-based simulation architecture provides the flexibility necessary for simulating the loss of hardware and software components while accurately considering fault containment regions. Each transmission unit (both on-chip and off-chip) is simulated by a dedicated ROS node as well. By providing the

means to intercept and transform arbitrary signals (in both the time and value domains), these ROS nodes enable the simulation of erroneous behavior of hardware components and software units—despite their black-box nature.<sup>1,2</sup>

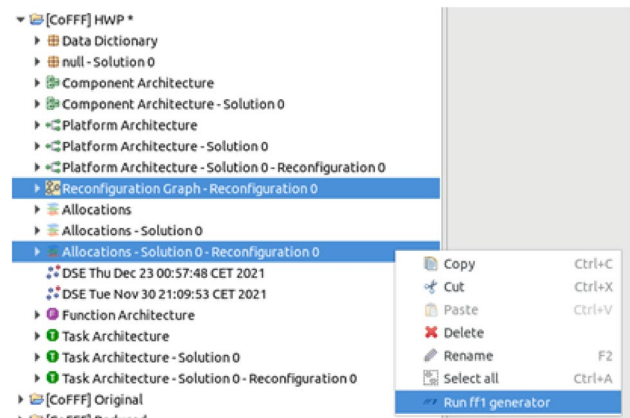
The approach presented in [31], however, requires the behavior of all components to be specified at the time of simulation—including pattern support components added to the task architecture by the safety pattern synthesis (see Sect. "SafPat"). While, for instance, voters (in the context of, e.g., an acceptance voting pattern) can be generated, fault detectors must be implemented manually. Requiring the implementation of such components would either nullify the benefits of the automated pattern synthesis, or impede the exploration of possible safety concepts by means of simulation-based fault injection. Hence, for the purpose of evaluating reconfiguration strategies, the fault detection is replaced by a timed trigger set to the expected worst case fault detection time interval (FDTI) as defined by ISO-26262-1:2018.

## Implementation

The approach described in Sect. "Approach" provides the means for the early evaluation of the dynamic and systemic properties of safety architectures. However, the effort of manually setting up the simulation framework for all candidates generated by Safety Pattern Synthesis and Feature Degradation Synthesis introduced in Sects. "Safety Pattern Synthesis" and "Feature Degradation Synthesis" is prohibitive. To address this lack of tool support, AutoFOCUS3 has been extended with an additional plugin which, given an exploration solution (or any other complete set of input models), (1) generates all necessary artefacts, (2) configures the simulation environment, and (3) provides the user with a fault injection interface.

For a set of input models to be complete, the following artefacts must be given:

- A *task architecture*, as introduced in Sect. "AutoFOCUS3". To be simulated, however, each task must be associated with a behavior specification. As we leverage the FMI standard, the simulation is not limited to behavior specifications modeled in AutoFOCUS3, but is capable of co-simulating an arbitrary set FMUs as long as the name and datatype of input and output ports on an FMU match the respective ports on the associated task. This requirement does not apply to pattern support components.
- A *platform architecture*. As described in Sect. "AutoFOCUS3" execution units can be modeled hierarchically



**Fig. 17** Screenshot of the AutoFOCUS3 user interface for the configuration of the simulation environment: Selecting the input artefacts enables the option to start the generator from the context menu

(in terms of nodes, tiles, and cores) and connected via transmission units (i.e., off-chip and on-chip networks).

- An *allocation table* (see Sect. "AutoFOCUS3") specifying the deployment of tasks to execution units and signals to routes across one or more transmission units. Crucially, the requirements generated by the safety pattern synthesis along with the safety architecture must already be reflected in the allocations.

Optionally, the plugin can consider the following input artefacts as well:

- A *partition architecture* representing the hypervisor-provided partitions used to meet the requirements generated by the safety pattern synthesis. Whenever a partition architecture is given, the allocation table must no longer directly specify the allocations of tasks to execution units. Instead, allocations of tasks to partitions and, in turn, of partitions to execution units are necessary to accurately replicate the fault containment regions.
- A *reconfiguration graph* as introduced in Sect. "Feature Degradation Synthesis", specifying the deployment type for tasks affected by the generated reconfiguration strategy.

As the allocation table references all of the aforementioned artefacts with exception of the reconfiguration graph, it is sufficient to select these two inputs from an AutoFOCUS3 project as shown in Fig. 17.

Given the aforementioned input models, the plugin generates the following output artefacts:

- A set of *behavior specifications* including the predefined FMUs associated with the given tasks. Redundant tasks introduced by the safety pattern synthesis might still lack

<sup>1</sup> <https://fmi-standard.org/>.

<sup>2</sup> <https://www.ros.org/>.

**Fig. 18** Overview over the functionality provided by the fault injection interface: The command-line tool is aware of the system architecture and can inject faults in processing elements, hypervisor-provided partitions, and single tasks

```
usage: inject.py [-h] [-s SYSTEM] [-t {e,p,t}] [-l | -n NAME] [-r] [-c CONFIGURATION] [-v | -q]

optional arguments:
  -h, --help            show this help message and exit
  -s SYSTEM, --system SYSTEM
                        specify the subject of the fault injection
  -t {e,p,t}, --type {e,p,t}
                        specify the type of target
  -l, --list            list all matching targets
  -n NAME, --name NAME
                        specify the name of the target
  -v, --verbose         output fault injection and reconfiguration status
  -q, --quiet          hide fault injection and reconfiguration status

  -r, --reconfigure    initiate a reconfiguration after the fault injection
  -c CONFIGURATION, --configuration CONFIGURATION
                        specify the current configuration using binary encoding
```

a behavior specification, in which case the FMUs specifying the nominal behavior are reused.

- The *configuration files* encoding the fault propagation across networks, the system's fault containment regions, and its reconfiguration strategy. As the simulation framework introduced in [31] is based on the same metamodel used by AutoFOCUS3 for the specification of E/E architectures (cf. [15]), the model transformation for generating the simulation's configuration files is straightforward. Overall, (1) a JSON file encoding the allocation of tasks and partitions is generated, (2) a JSON file for each transmission unit containing the source and target of each signal routed through that network, and, finally, (3) a JSON file breaking down the reconfiguration graph to sets of tasks to be either activated or terminated after each fault.
- A ROS *launch file*, starting a ROS node for each FMU and each transmission unit as described in [31]. To ensure the deterministic co-simulation of FMUs across ROS nodes and thus retain the execution semantics defined by the FMI standard, the FMI adapter first introduced in [32] has been modified to use ROS services instead of ROS topics. In addition, a custom ROS master node is launched, coordinating the co-simulation.

The ROS launch file can be used to launch the co-simulation in isolation or, ideally, in conjunction with a simulation of actuator dynamics, sensor perception, and the physical environment. This allows the observation of the system's closed loop behavior.

The plugin also provides a command-line interface to support the user with the fault injection itself (c.f. 18). Given the generated configuration files, the program is aware of the system's software-hardware integration and, thereby, the system's fault containment regions. Hence, the immediate effects of a fault can be replicated accurately. The loss of an execution unit is, for instance, translated to a loss of the allocated partitions and, in turn, to the loss of the tasks running within the affected partitions. Finally, the FMUs associated with the identified tasks are terminated. A command-line

interface was implemented over a graphical interface to allow for automated test case execution.

## Results

To evaluate the safety architecture candidate selected in Sects. "Safety Pattern Synthesis" and "Feature Degradation Synthesis", the here introduced tool support for co-simulation-based fault injection testing has been applied to the highway pilot use case using the fortissimo Simulation.<sup>3</sup> The ROS-powered environment provided by the fortiss Mobility Lab<sup>4</sup> simulates the ego vehicle's sensors, actuators, dynamics as well as its environment.

For the purpose of illustrating the application of the here presented tool support for the fault injection framework introduced in [31], we perform two test runs. We simulate the loss of a core to which parts of the highway pilot's ACC are deployed, first with the baseline architecture and then using the synthesized safety architecture candidate:

As shown in Fig. 19, the user is informed about all tasks stopped as a consequence of losing the processing element they are allocated to. As the baseline architecture does not provide any safety patterns, any functionality associated with the affected tasks is lost. The visualization of the fortissimo Simulation running the baseline architecture—a screenshot of which is seen in Fig. 20—confirms the loss of the ACC functionality: The rover stops and its control panel indicates that the functionality is no longer available.

In case of the synthesized architecture, the consumer's fault detector switches to the second instance (i.e., redundancy) of the ACC controller. As the fault detector was introduced by the architecture pattern synthesis, its behavior has not yet been specified at the time of simulation. Thus, the switching is automatically triggered by the fault injection tool (see Fig. 21). Due to the consumer (in this case the Driver Assistance Controller) now reading the output of the second instance of the ACC task, the

<sup>3</sup> <https://git.fortiss.org/ff1/simulation>.

<sup>4</sup> <https://www.fortiss.org/en/research/fortiss-labs/detail/mobility-lab>.



```

root@for1856:~/simulation# ./inject.py -v -s rover_2 -t e -n Core_1
Stopped: Task_Adaptive_Cruise_Control
Stopped: Core_1

```

**Fig. 19** Output of the command-line fault injection tool after simulating the loss of a core: In addition to informing the user about the loss of the selected execution unit, the script also reports the tasks which have been lost as a consequence

functionality is not disrupted: The vehicle continues driving and the control panel confirms the availability of the functionality.

These results indicate the effectiveness of the selected safety architecture in improving the availability of software functions in the face of hardware losses. Nevertheless, a single test run does not yet result in any factual evidence for the adequacy of the safety architecture. The confidence in results obtained by means of simulation can, however, be increased by (1) proving the accuracy of the simulation (e.g., as suggested in [33]), (2) indicating the completeness of functional scenarios to be tested (cf., [34]), and (3) optimizing the criticality of test cases, i.e. to which extent the worst cases are covered (e.g., by means of search-based approaches as suggested in [35]).

In terms of performance, a system with an Intel Core i7-9850 H CPU and a NVIDIA Quadro T2000 GPU is capable of executing the aforementioned experiments involving the highway pilot with a *real-time factor* (the ratio of simulation time to real time) of 3, i.e., the scenarios can be simulated three times faster than real time. Here, the physics engine (Open Dynamics Engine<sup>5</sup>) is set to a step size of 3 ms (with respect to the simulation time) while the FMI-based co-simulation is stepped at 33 Hz (with respect to the simulation time). Two alternative task architectures of the highway pilot use case involving 30 (47) software tasks exchanging 145 (197) signals can still be executed with a real-time factor of 2 (1.65). The observed reduction of the real-time factor is mainly caused by the latency of the TCP/IP-based communication among ROS nodes.

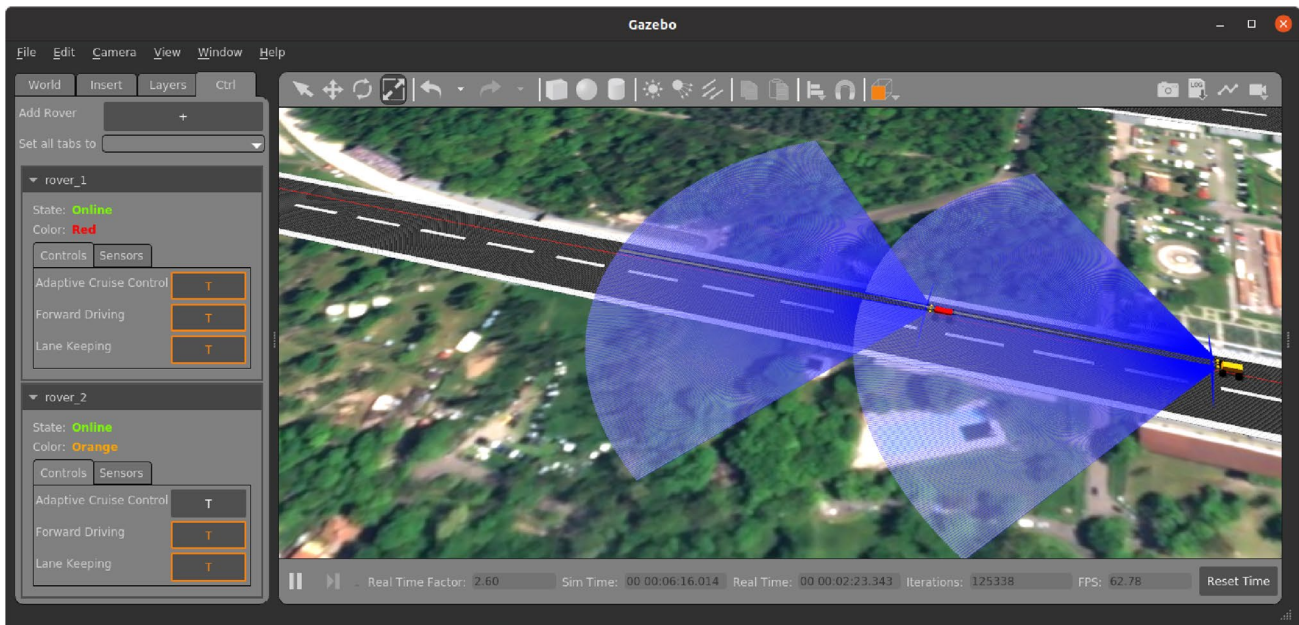
While increasing the step size of the physics engine improves performance, it compromises the simulation's accuracy. Hence, the maximum step size depends on the complexity of the dynamics to simulate and the desired level of accuracy, and is, thus, use case-specific. In contrast, the step size of the co-simulation must match the step size required by the FMUs (with respect to the simulation time) to ensure that the behavior of the simulated system reflects the real-time properties of the system under test. For large test suites, the execution of test cases can be parallelized and distributed by means of containerization.<sup>5</sup>

<sup>5</sup> <https://ode.org/>.

## Related Work

In the automotive domain, and for automated and autonomous driving in particular, fail-operational fault-tolerance is essential [36–38]. A number of research projects of investigated fault-tolerant vehicle architectures, such as RACE [39, 40], SafeAdapt [41, 42] and SAFER [43]. The SafeAdapt project provided a meta-modeling approach to describe architectural patterns for fail-operational, gracefully degrading systems, providing a library of different redundancy and graceful degradation patterns, such as a fail-operational graceful degradation (FOGD) that considers different (possibly degraded) states of system features [42]. While the approach foresees redundancy, e.g., based on hot/cold standbys, it does not consider the deployment of software components to hardware units. SAFER introduced a system architecture for dependable autonomous vehicles, including support for graceful degradation in failure scenarios (e.g., in cases of ECU failures) [43, 44]. While the approach foresees environment dependent (or context aware) degradation strategies, it focuses on the provision of the system architecture rather than a methodology to deploy functions to it. The RACE project developed a fault-tolerant software and system automotive architecture for vehicles. The hardware platform is based on the Central Platform Computer (CPC) that consists of computing units in a DMR configuration. The RACE runtime environment [45] is based on a data-centric paradigm that enables that application software components have access to data and the system's peripherals independent of their (current) deployment, resulting in a flexible solution in which software components may be dynamically relocated. In the context of RACE, a model-based approach relying on formal methods (SMT) to analyze the degradation of functional features in failure scenarios based have been investigated [30, 46].

Approaches combining MBSE with safety analysis methods have been proposed by, e.g., [47, 48]. For example, the HiP-HOPS tool [47] has been proposed to semi-automate the safety analysis process (using FTA and FMEA methods) on system architectures. This tool has three main steps: The first step is the system modeling and the failure annotation. This step is performed manually by the user, in particular the user annotates failure data to individual components of the system such as components that provide control actions (e.g., acceleration, steering or braking). The second step is the fault tree synthesis process to automatically determine which components caused the failures defined the the first step. The third step is the generation of an FMEA to obtain the minimal cut sets, i.e., the smallest possible combinations of failures capable of causing the failures defined the the first step. Our work complements [47] by providing means to automate the recommendation of safety mechanisms to



**Fig. 20** Screenshot of the fortissimo Simulation's visualization: After the loss of the core used to execute the ACC, the rover stops and the control panel indicates that the function is no longer active

address faults/failures. We are interested in augmenting our tool chain to enable the automatic identification of system faults/failures by, e.g., using tools like HiP-HOPS.

**Safety Pattern Synthesis** provides requirements for each recommended safety architecture pattern, in particular allocation (a.k.a. deployment) requirements. The allocation requirements are currently performed manually in the toolchain proposed by this article. We believe that some of the requirements provided by **Safety Pattern Synthesis** can be automated. Eder et al. [9] proposed a design space exploration approach to enable the allocation of software components into hardware units in an semi-automated fashion. This approach takes into account the structure of system architectures (incl., software components and hardware units), and a DSL to formalize requirements (e.g., timing, memory consumption) with respect to the design space exploration problem. This DSL is specified as a first-order logic language that can be automated by solving techniques such as Satisfiability Modulo Theories (SMT) [49]. The combination of the proposed DSL with the designed system architectures enabled the applicability of the semi-automated design space exploration for allocating software components into hardware units. This approach has been extended to enable a synthesis of the topology of technical platforms together with a deployment [12]. The approach has been implemented as a feature of AF3.

As noted in our previous work, the problem of validating the real-time properties of safety architectures of Cyber-Physical Systems (CPS) has been addressed by several

simulation-based Fault Injection (FI) approaches [31]. These, however, require the disclosure of tool-specific behavior specifications—thereby violating the requirements for interoperability and protection of Intellectual Property (IP) mentioned in Sect. "Requirements". [50–52], and [53] all present system-level FI approaches relying, however, on the availability of white box behavior specifications: The MODIFI tool presented in [52], for instance, provides extensive FI capabilities in both software and hardware components—however, only when defined as MATLAB Simulink models. Co-simulation-based system integration frameworks provide a different line of action: Often based on the FMI standard enabling the exchange of tool-agnostic black-box behavior specifications, these approaches meet the interoperability and IP protection requirements—lacking, however, either indications on how to represent E/E architectures and the deployment of software to hardware, or adequate support for FI testing. The FMI-based integration platforms for CPS introduced in [54] and [55], for example, demonstrate the feasibility

```

root@for1856:~/simulation# ./inject.py -v -s rover_2 -t e -n Core_1 -r
Stopped: Task_Adaptive_Cruise_Control
Stopped: Core_1
Switched: HTD_Fault_Detector_2_Task_Driver_Assistance_Coordinator_ID_145674
Stopped: Task_Adaptive_Cruise_Control
Stopped: Task_Adaptive_Cruise_Control
Stopped: Task_Emergency_Brake_V2
Stopped: Task_Emergency_Brake_V2
Configuration: 011

```

**Fig. 21** Output of the command-line fault injection tool after simulating the loss of a core and the subsequent reconfiguration: All fault detectors supervising affected functions are notified of the loss, triggering a switch to the respective fallback tasks

of system-level FMI-based co-simulation of hardware and software components, yet lack FI capabilities. In contrast, the FI approach for FMI-based co-simulation proposed in [56] does not provide any guidelines on how to represent software and hardware components and their interactions (e.g., through deployment relations). Aiming at filling this gap, we previously introduced an FMI-based FI framework [31]. It lacks, however, adequate tool support and does allow for pattern support components with undefined behavior. By automating the simulation setup and introducing an emulation mechanism for generated pattern support components, the solution presented in Sect. "Simulation" addresses both shortcomings of our previous work.

## Conclusion

This article presented a toolchain for synthesizing and validating architectures with safety architecture patterns. Specifically, our toolchain provides means to synthesize both the structure of the architecture to include safety architecture patterns, and the switching logic of pattern components (i.e., the switch from the primary to redundant components in the presence of faults). We validate our toolchain by means of simulation-based fault-injection using an industrial use-case for autonomous driving systems, namely, a Highway Pilot system.

**Data availability** The results have been achieved together with our industrial partners, and not publicly available. The interested reader may contact us for the data.

## Declarations

**Conflict of interest** The authors declare that they have no conflict of interest.

## References

1. Armoush, A. Design patterns for safety-critical embedded systems. PhD thesis, RWTH Aachen University (2010)
2. Preschern C, Kajtazovic N, Kreiner C. Building a safety architecture pattern system. In: van Heesch U, Kohls C (eds) Proceedings of the 18th European Conference on Pattern Languages of Program, EuroPLoP 2013, Irsee, Germany, July 10–14, 2013, pp. 17–11755. ACM, New York (2013). <https://doi.org/10.1145/2739011.2739028>.
3. Dantas YG Munaro T, Cărlan C, Nigam V, Barner S, Fan S, Pretschner A, Schöpp U, Tverdyshev S. A Model-based System Engineering Plugin for Safety Architecture Pattern Synthesis. In: Pires LF, Hammoudi S, Seidewitz E (eds) Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2022, Online Streaming, February 6–8, 2022, pp. 36–47. SCITEPRESS, Portugal (2022). <https://doi.org/10.5220/0010831700003119>.
4. fortiss GmbH: AutoFOCUS 2.21. Available at <https://af3.fortiss.org/>. <https://af3.fortiss.org/>. Accessed 10 June 2022.
5. Eclipse Foundation: Eclipse Modeling Framework (EMF). Available at <https://www.eclipse.org/modeling/emf/>. <https://www.eclipse.org/modeling/emf/>. Accessed 10 June 2022.
6. Pohl K, Hönniger R, Harald Achatz Broy M (eds) (2012) The SPES 2020 Engineering-methodology for software-intensive embedded systems, p. 301. Springer, New York
7. Aravatinos V, Voss S, Teuffl S, Hölzl F, Schätz B. AutoFOCUS 3: Tooling concepts for seamless, model-based development of embedded systems. In: Proc. 8<sup>th</sup> Int. Workshop Model-based Architecting of Cyber-Physical and Embedded Systems (ACES-MB), pp. 19–26 (2015)
8. Barner S, Chauvel F, Diewald A, Eizaguirre F, Haugen Ø, Migge J, Vasilevskiy A. In: Ahmadian, H., Obermaisser, R., Perez, J. (eds.) Modeling and Development Process, pp. 87–161. CRC Press, Boca Raton (2018). <https://doi.org/10.1201/9781351117821-4>
9. Eder J, Zverlov S, Voss S, Khalil M, Ipatiov A, Bringing DSE to life: Exploring the design space of an industrial automotive use case. In: 20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17–22, 2017, pp. 270–280. IEEE Computer Society, Washington, D.C. (2017). <https://doi.org/10.1109/MODELS.2017.36>.
10. Zverlov S, Voss S, Böhm T, Herpel H.-J, Kerep M, Model-based methodology for space vehicles. In: Proceedings of the Eurospace Annual Conference on Data Systems in Aerospace (DASIA) (2019)
11. Diewald A, Barner S, Saidi S, Combined data transfer response time and mapping exploration in mpsoes. In: 10<sup>th</sup> International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS) Co-located with ECRTS (2019). <https://archives.ecrts.org/fileadmin/WebsitesArchiv/ecrts2019/waters/waters-program/>
12. Eder J, Bayha A, Voss S, Ipatiov A, Khalil M, From deployment to platform exploration: Automatic synthesis of distributed automotive hardware architectures. In: Wasowski, A., Paige, R.F., Haugen, Ø. (eds.) Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, Copenhagen, Denmark, October 14–19, 2018, pp. 438–446. ACM, New York (2018). <https://doi.org/10.1145/3239372.3239385>.
13. Eder J, Voss S, Bayha A, Ipatiov A, Khalil M. Hardware architecture exploration: automatic exploration of distributed automotive hardware architectures. Software and Systems Modeling. 2020. <https://doi.org/10.1007/s10270-020-00786-6>.
14. Migge J, Balbastre P, Barner S, Chauvel F, Craciunas S.S, Diewald A, Durrieu G, Haugen Ø, Seyed A.A.J, Pagetti C, Oliver R.S, Vasilevskiy A In: Ahmadian, H., Obermaisser, R., Perez, J. (eds.) Algorithms and Tools, pp. 163–259. CRC Press, Boca Raton, 2018. <https://doi.org/10.1201/9781351117821-5>
15. Barner S, Diewald A, Migge J, Syed A, Fohler G, Faugère M, Gracia Pérez D. DREAMS toolchain: Model-driven engineering of mixed-criticality systems. In: Proceedings of the ACM/IEEE 20<sup>th</sup> International Conference on Model Driven Engineering Languages and Systems (MODELS '17), pp. 259–269. IEEE, Austin, TX, USA 2017. <https://doi.org/10.1109/MODELS.2017.28>
16. Barner S, Diewald A, Eizaguirre F, Vasilevskiy A, Chauvel F. Building product-lines of mixed-criticality systems. In: Proceedings of the Forum on Specification and Design Languages (FDL 2016). IEEE, Bremen, Germany 2016. <https://doi.org/10.1109/FDL.2016.7880378>

17. Eder J, Voss S. Usable design space exploration in AutoFOCUS3. In: Joint Proceedings of the 12th Educators Symposium (EduSymp 2016) and 3rd International Workshop on Open Source Software for Model Driven Engineering (OSS4MDE 2016) Collocated with MODELS 2016, pp. 51–58. CEUR-WS, 2016. <http://ceur-ws.org/Vol-1835/paper08.pdf>
18. Voss S, Eder J, Hölzl F. Design space exploration and its visualization in autofocus3. In: Software Engineering (Workshops), pp. 57–66 2014. <http://ceur-ws.org/Vol-1129/paper33.pdf>
19. ISO26262: ISO 26262, road vehicles - functional safety - part 6: Product development: software level (2018). Available at <https://www.iso.org/standard/43464.html>
20. Avizienis A, Laprie J-C, Randell B, Landwehr CE. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans Dependable Secur Comput.* 2004;1(1):11–33.
21. Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems, (2012)
22. Sljivo I, Uriagereka GJ, Puri S, Gallina B. Guiding assurance of architectural design patterns for critical applications. *J Syst Archit.* 2020;110: 101765. <https://doi.org/10.1016/j.sysarc.2020.101765>.
23. Preschern C, Kajtažovic N, Kreiner C. Security analysis of safety patterns. In: Proceedings of the 20th Conference on Pattern Languages of Programs (PLoP '13). 2013. pp. 1–38.
24. Biondi A, Nesti F, Cicero G, Casini D, Buttazzo GC. A safe, secure, and predictable software architecture for deep learning in safety-critical systems. *IEEE Embed Syst Lett.* 2020;12(3):78–82. <https://doi.org/10.1109/LES.2019.2953253>.
25. Bak S, Chivukula D.K, Adekunle O, Sun M, Caccamo M, Sha L. The system-level simplex architecture for improved real-time embedded system safety. In: 15<sup>th</sup> IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2009, San Francisco, CA, USA, 13-16 April 2009, pp. 99–107. IEEE Computer Society, Washington, D.C. 2009. <https://doi.org/10.1109/RTAS.2009.20>
26. Dantas YG, Kondeva A, Nigam V. Less manual work for safety engineers: Towards an automated safety reasoning with safety patterns. In: International Conference on Logic Programming (ICLP) 2020
27. Leone N, Pfeifer G, Faber W, Eiter T, Gottlob G, Perri S, Scarcello F. The DLV system for knowledge representation and reasoning. *ACM Trans Comput Logic.* 2006;7(3):499–562.
28. Wood M, Robbel P, Maass M, Tebbens RD, Meijjs M, Harb M, Reach J, Robinson K, Wittmann D, Srivastava T, Bouzouraa M.E, Liu S, Wang Y, Knobel C, Boymanns D, Löhning M, Dehlink B, Kaule D, Krüger R, Frtunikj J, Raisch F, Gruber M, Steck J, Mejia-Hernandez J, Syguda S, Blüher P, Klonecki K, Schnarz P, Wiltshko T, Pukallus S, Sedlaczek K, Garbacik N, Smerza D, Li D, Timmons A, Bellotti M, O'Brien, M., Schöllhorn, M., Dannebaum, U., Weast, J., Tatourian, A., Dornieden, B., Schnetter, P., Themann, P., Weidner, T., Schlicht, P.: Safety first for automated driving. Technical report, Aptiv; Audi; Baidu; BMW; Continental; Daimler; Fiat Chrysler Automobiles; HERE; Infineon; Intel; Volkswagen; (2019). <https://www.daimler.com/documents/innovation/other/safety-first-for-automated-driving.pdf>. Accessed 10 June 2022.
29. EmbASP. Available at <https://www.mat.unical.it/calimeri/projects/embasp/>. Accessed 10 June 2022.
30. Becker K, Voss S, Schätz B. Formal analysis of feature degradation in fault-tolerant automotive systems. *Science of Computer Programming.* 2018;154:89–133. <https://doi.org/10.1016/j.scico.2017.10.007>. Formal Techniques for Safety-Critical Systems 2015.
31. Munaro T, Muntean I. Early assessment of system-level safety mechanisms through co-simulation-based fault injection. In: 2022 IEEE Intelligent Vehicles Symposium (IV), pp. 1703–1708 2022. <https://doi.org/10.1109/IV51971.2022.9827327>
32. Schröder N, Lenord O, Lange R. Enhanced motion control of a self-driving vehicle using modelica, fmi and ros. Proceedings of the 13th International Modelica Conference, Regensburg, Germany, March 4-6, 2019 157, 441–450 (2019). <https://doi.org/10.3384/ecp19157441>
33. Sargent R.G. Verification and validation of simulation models. In: Proceedings of the 2010 Winter Simulation Conference, pp. 166–183. IEEE, 2010. <https://doi.org/10.1109/WSC.2010.5679166>.
34. Hauer F, Schmidt T, Holzmüller B, Pretschner A. Did we test all scenarios for automated and autonomous driving systems?. pp. 2950–2955. IEEE, (2019). <https://doi.org/10.1109/ITSC.2019.8917326>.
35. Matinnejad R, Nejati S, Briand L, Bruckmann T, Poull C. Search-based automated testing of continuous controllers: Framework, tool support, and case studies. *Information and Software Technology.* 2015;57:705–22. <https://doi.org/10.1016/j.infsof.2014.05.007>.
36. Sinha P. Architectural design and reliability analysis of a fail-operational brake-by-wire system from iso 26262 perspectives. *Reliability Engineering & System Safety.* 2011;96(10):1349–59. <https://doi.org/10.1016/j.res.2011.03.013>.
37. Kohn A, Käßmeyer M, Schneider R, Roger A, Stellwag C, Herkersdorf A. Fail-operational in safety-related automotive multi-core systems. In: 10th IEEE International Symposium on Industrial Embedded Systems (SIES) 2015. <https://doi.org/10.1109/SIES.2015.7185051>
38. Wei J, Snider J.M, Kim J, Dolan J.M, Rajkumar R, Litkouhi B. Towards a viable autonomous driving research platform. In: 2013 IEEE Intelligent Vehicles Symposium (IV), pp. 763–770 (2013). <https://doi.org/10.1109/IVS.2013.6629559>
39. Sommer S, Camek A, Buckl C, Becker K, Zirkler A, Fiege L, Armbruster M, Knoll A. Race: A centralized platform computer based architecture for automotive applications. In: Vehicular Electronics Conference (VEC) and the International Electric Vehicle Conference (IEVC) (VEC/IEVC 2013). IEEE 2013
40. Knoll A, Buckl C, Kuhn K.-J, Spiegelberg G. In: Dajsuren, Y., van den Brand, M. (eds.) The RACE Project: An Informatics-Driven Greenfield Approach to Future E/E Architectures for Cars, pp. 171–195. Springer. [https://doi.org/10.1007/978-3-030-12157-0\\_8](https://doi.org/10.1007/978-3-030-12157-0_8)
41. Ruiz A, Juez G, Schleiss P, Weiss G. A safe generic adaptation mechanism for smart cars. In: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), pp. 161–171 2015. <https://doi.org/10.1109/ISSRE.2015.7381810>
42. Penha D, Weiss G, Stante A. Pattern-based approach for designing fail-operational safety-critical embedded systems. In: 2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing, pp. 52–59 (2015). <https://doi.org/10.1109/EUC.2015.14>
43. Kim J, Bhatia G, Rajkumar R, Jochim M. Safer: System-level architecture for failure evasion in real-time applications. In: 2012 IEEE 33rd Real-Time Systems Symposium, pp. 227–236 2012. <https://doi.org/10.1109/RTSS.2012.74>
44. Kim J, Rajkumar RR, Jochim M. Towards dependable autonomous driving vehicles: A system-level approach. *SIGBED Rev.* 2013;10(1):29–32. <https://doi.org/10.1145/2492385.2492390>.
45. Becker K, Frtunikj J, Felser M, Fiege L, Buckl C, Rothbauer S, Zhang L, Klein C. RACE RTE: A Runtime Environment for Robust Fault-Tolerant Vehicle Functions. In: CARS 2015 - Critical Automotive Applications: Robustness & Safety, Paris, France 2015. <https://hal.archives-ouvertes.fr/hal-01192987>. Accessed 10 June 2022.
46. Becker K. Software deployment analysis for mixed reliability automotive systems. Dissertation, Technische Universität München, München (2017). <http://nbn-resolving.de/urn/resolver>.

- [pl?urn:nbn:de:bvb:91-diss-20170726-1345914-1-1](https://doi.org/10.1016/j.engfailanal.2010.09.025). Accessed 10 June 2022.
47. Papadopoulos Y, Walker M, Parker D, Ruede E, Hamann R, Uhlig A, Graetz U, Lien R. Engineering failure analysis and design optimisation with HiP-HOPS. *Journal of Engineering Failure Analysis*. 2011;18(2):590–608. <https://doi.org/10.1016/j.engfailanal.2010.09.025>.
  48. Belmonte F, Soubiran E. A model based approach for safety analysis. In: Ortmeier, F., Daniel, P. (eds.) *Computer Safety, Reliability, and Security - SAFECOMP 2012 Workshops: Sassur, ASCoMS, DESEC4LCCI, ERCIM/EWICS, IWDE, Magdeburg, Germany, September 25-28, 2012*. Proceedings. *Lecture Notes in Computer Science*, vol. 7613, pp. 50–63. Springer, New York (2012). [https://doi.org/10.1007/978-3-642-33675-1\\_5](https://doi.org/10.1007/978-3-642-33675-1_5).
  49. de Moura L.M, Bjørner N. Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008*. Proceedings. *Lecture Notes in Computer Science*, vol. 4963, pp. 337–340. Springer, New York, (2008). [https://doi.org/10.1007/978-3-540-78800-3\\_24](https://doi.org/10.1007/978-3-540-78800-3_24).
  50. Uriagereka G.J, Lattarulo R, Rastelli J.P, Calonge E.A, Lopez A.R, Ortiz H.E. Fault injection method for safety and controllability evaluation of automated driving. In: *2017 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1867–1872. IEEE, (2017). <https://doi.org/10.1109/IVS.2017.7995977>
  51. Sini J, Violante M. An Automatic Approach to Perform FMEDA Safety Assessment on Hardware Designs. In: *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*, pp. 49–52. IEEE, (2018). <https://doi.org/10.1109/IOLTS.2018.8474217>
  52. Svenningsson R, Vinter J, Eriksson H, Törngren M. MODIFI: A MODEL-Implemented Fault Injection Tool. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6351 LNCS, 210–222 2010
  53. Saraoglu M, Morozov A, Janschek K. MOBATSIm: MODEL-Based Autonomous Traffic Simulation Framework for Fault-Error-Failure Chain Analysis. *IFAC-PapersOnLine*. 2019;52:239–44.
  54. Neema H, Gohl J, Lattmann Z, Sztipanovits J, Karsai G, Neema S, Bapty T, Batteh J, Tummescheit H, Sureshkumar C. Model-Based Integration Platform for FMI Co-Simulation and Heterogeneous Simulations of Cyber-Physical Systems. In: *Proceedings of the 10th International Modelica Conference, March 10-12, 2014, Lund, Sweden*, vol. 96, pp. 235–245 2014
  55. dSPACE GmbH: Always the Right Model. *dSPACE Magazin*, 12–17 2015
  56. Frasheri M, Thule C, Macedo H.D, Lausdahl K, Larsen P.G, Esterle L. Fault injecting co-simulations for safety. In: *2021 5th International Conference on System Reliability and Safety (ICRSRS)*, pp. 6–13. IEEE, ??? (2021). <https://doi.org/10.1109/ICRSRS53853.2021.9660728>

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

## Chapter 6

# An Intruder Model for Automotive Service-Oriented Architectures

Chapter 6 proposes the formalization of an intruder model tailored for automated driving systems, particularly those that employ Service-Oriented Architectures (SOA). This chapter also presents a logic programming tool that implements the proposed intruder model, facilitating the automated enumeration of attack paths. The logic programming tool has been validated using an example taken from the automotive domain.

**Contributing article:** Yuri Gil Dantas, Simon Barner, Pei Ke, Vivek Nigam, and Ulrich Schöpp: Automating Vehicle SOA Threat Analysis Using a Model-Based Methodology. ICISSP 2023: 180-191

**Copyright information:** Science and Technology Publications Lda (SCITEPRESS), 2024. <https://doi.org/10.5220/0011786400003405>.

**Author contributions:** The concept for the publication was jointly developed by Yuri Gil Dantas, Vivek Nigam, and Ulrich Schöpp. Yuri Gil Dantas performed a literature review to identify relevant attacks against SOA architectures implemented by automated driving systems. Based on the results of the literature review, Yuri Gil Dantas formalized an intruder model for SOA architectures, which is presented in the article. After several research discussions with Vivek Nigam, Yuri Gil Dantas implemented the tool and carried out the experiments presented in the article. Yuri Gil Dantas took the lead in writing the initial draft of the article. The article includes the description of a SOA architecture modeled by Ulrich Schöpp. Vivek Nigam assisted in improving the article. Yuri Gil Dantas handled subsequent revisions and corrections.

# Automating Vehicle SOA Threat Analysis Using a Model-Based Methodology

Yuri Gil Dantas<sup>1</sup>, Simon Barner<sup>1</sup>, Pei Ke<sup>2</sup>, Vivek Nigam<sup>2</sup> and Ulrich Schöpp<sup>1</sup>

<sup>1</sup>fortiss GmbH, Munich, Germany

<sup>2</sup>Huawei Technologies Düsseldorf GmbH, Düsseldorf, Germany

**Keywords:** Automotive, Threat Analysis, Service-Oriented Architectures, Automation, Safe and Secure-by-Design.

**Abstract:** This article proposes automated methods for threat analysis using a model-based engineering methodology that provides precise guarantees with respect to safety goals. This is accomplished by proposing an intruder model for automotive SOA which together with the system architecture and the loss scenarios identified by safety analysis are used as input for computing assets, impact rating, damage/threat scenarios, and attack paths. To validate the proposed methodology, we developed a faithful model of the autonomous driving functions of the Apollo framework, a widely used open source autonomous driving stack. The proposed machinery automatically enumerates several attack paths on Apollo, including attack paths not reported in the literature.

## 1 INTRODUCTION

The automotive industry is under great transformation to meet challenges of implementing features such as Autonomous Driving and Over-the-Air Updates. Instead of using distributed architectures with domain-specific hardware, vehicles are using software-intensive *Service-Oriented Architectures* (SOA) with powerful centralized computer units. The open-source Apollo framework (Apollo, 2021) is an example of this transformation providing autonomous vehicle features that have been used in the development of real-world autonomous vehicle applications, such as autonomous taxis and buses.

This transformation has also increased concerns on how attackers can affect road-user safety. While security threats to safety have been known for more than a decade ago (WIRED, 2015), the upcoming/recent standards ISO 21434 (ISO/SAE 21434, 2020) and the UNECE (UN, 2021) have pushed industry to change its development process to enable safe and secure-by-design vehicles. For example, the ISO 21434 puts great emphasis on the development process and on the threat analysis, e.g., Damage/Threat Scenario/Attack Path enumeration, that shall be performed and addressed before putting the vehicle on the road. At the end, Original Equipment Manufacturers (OEMs) shall provide compelling arguments and evidence, i.e., an assurance case, that their vehicles are safe also from a security perspective.

OEMs may pay a costly price if they develop autonomous vehicle features without previously producing analysis, argument, and evidence supporting vehicle safety and security. Without these artifacts, it is hard to expect that these vehicles will be accepted by certification agencies and be allowed to be used in several countries, once standards are more heavily enforced. Even more troublesome is that several attacks have been reported that can cause serious hazards to road-users, such as vehicle collisions. *As we claim here, many of these attacks could have been identified during the design of the system architecture by using a safe and secure-by-design approach with suitable threat analysis supported by automation.*

A key challenge for the development of safe and secure-by-design vehicles is handling the enormous complexity involved. For example, without adequate countermeasures, SOA allows any software component to publish any data including data that may be consumed by safety-critical functions. This has been a source of, e.g., overprivilege attacks (Hong et al., 2020) causing hazardous situation whenever a safety-critical function consumes data erroneously published by a malicious component (or even by a faulty component). For another example, malicious components may exploit SOA communication vulnerabilities to cause man-in-the-middle attacks (Zelle et al., 2021). Moreover, sensors, such as cameras and GPS radios,

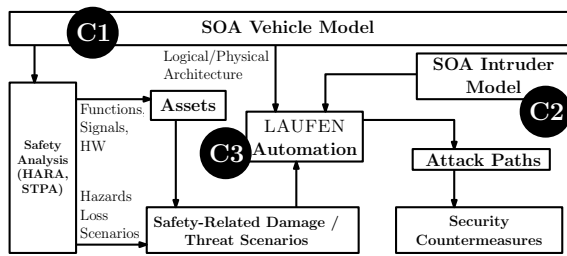


Figure 1: Illustration of the proposed Safe and Secure-by-Design methodology, tool-chain and key contributions (C1, C2 and C3).

are attack surfaces that may be exploited by attackers to cause hazards (Jha et al., 2020; Shen et al., 2020).

### 1.1 Safe and Secure-by-Design Methodology and Contributions

The proposed safety and security methodology and three key contributions are depicted in Figure 1. The methodology is built upon the following key ideas from the automotive safety and security co-engineering literature:

- **Analysis Techniques for Software-Intensive Systems:** System Theoretic Process Analysis (STPA) (Leveson and Thomas, 2018) has been recommended for safety analysis of autonomous functions by standards such as the ISO 21448 (Safety Of The Intended Functionality – SOTIF) (SOTIF, 2021) for assuring the safety of features such as autonomous driving. This is because STPA does not assume linear causal dependency and rather puts a greater emphasis on the faulty/malicious component interactions.
- **Safety to Security:** The approach recommended by Bosch engineers (Förster et al., 2019) uses safety artifacts, e.g., safety goals and hazards, as inputs to security analysis. There are two key motivations for this: 1) A safety analysis is typically carried out before a security analysis. 2) By using safety as input to security, one can claim, through appropriate traceability, completeness of security analysis w.r.t. to the results of the safety analysis. This is done, for example, by checking whether all causes of hazards (called loss scenarios in STPA terminology) have traces to appropriate security analysis.
- **Model-Based Tool-Chains:** Model-based engineering approaches are based on formal abstractions of the system under design and therefore help mitigate the complexity of nowadays software and hardware architectures and to boost development speed and quality

when compared to traditional document-based approaches by means of automated analysis, design and validation tools.

While these methods have been proposed, this article is the first to apply them together into an overarching model-based methodology for SOA vehicle architectures. As depicted in Figure 1, we start from a (SOA) Vehicle Model, specifying the key functions, logical components, and platform (a.k.a. physical) architecture. These model elements ensure the soundness of the approach, as the safety and security analysis that follow are traced to the model. From the Hazard Analysis and Risk Assessment (HARA) and STPA analysis, key safety functions, channels and physical elements are identified, which are then traced as assets from the security perspective that need to be protected. Loss scenarios obtained from STPA, i.e., the situations that may lead to hazards, are traced to damage and threat scenarios specifying how intruders can cause safety hazards. From this point onward, we carry out a security analysis, e.g., using the logical and platform architectures to identify attack paths that can cause threat scenarios. Ultimately, we discuss potential countermeasures to address threats.

The key benefits of the approach are three-fold: The first benefit is a full traceability between safety and security analysis and the vehicle model. This means that the analysis is reflected in the actual implementation that will be deployed in the vehicle. The second benefit is that the methodology provides guarantees that all loss scenarios for all hazards are considered by the security analysis, e.g., all loss scenarios are traced to damage/threat scenarios. This means that all identified safety issues shall be considered from the security perspective. The third benefit is that our model-based methodology enables the use of automated methods, e.g., the automated enumeration of attack paths based on intruder models.

The main contributions of this article are:

- **Apollo-Based Vehicle Model (C1):** By examining the relevant pieces of code in the Apollo code-base related to autonomous driving functions, we designed a faithful vehicle model. The model reflects the SOA publish and subscribe pattern, and the information (namely the topics) between the Apollo components. To the best of our knowledge, it is the first model based on the Apollo v7.0.0 code base.
- **Intruder Model for Vehicle SOA (C2):** By examining vehicle SOA security literature, we formalized an intruder model for vehicle SOA. The intruder is capable of carrying out Man-in-the-Middle (MITM) attacks, and carrying out spoofing attacks by infiltrating the system



from public interfaces to, e.g., exploit perception sensors, such as LiDAR and Camera.

- **Attack Path Automation (C3):** We developed a machinery (LAUFEN) to automate the enumeration of attack paths on the vehicle system architecture. LAUFEN takes as input the model, assets, damage/threat scenarios, and the implementation of the intruder model, and outputs all attack paths.

We demonstrate and validate our approach and automation on the developed Apollo Vehicle Model. Our focus is on safety assets as it is the main concern for autonomous driving. The developed machinery identified **246** attack paths. The attack paths include attacks that have been reported in the literature. Given the traceability to safety analysis, our machinery identifies a much greater number of attack paths that would need to be mitigated (or for which some security rationale shall be provided) by security countermeasures. Indeed, based on the generated attack paths, we identified potential attacks that have not yet been reported.

**Structure of This Article:** Section 2 describes the Apollo vehicle model. Section 3 describes how to trace assets, damage/threat scenarios to artifacts produced by safety analysis. Sections 4 and 5 describe the intruder model and the attack paths generated, including its validation using the Apollo model. After discussing related and potential future work in Section 6, we conclude in Section 7.

## 2 APOLLO MODELING

Apollo (Apollo, 2021) is an open-source autonomous driving stack enabling highly autonomous vehicle features (more precisely, at Level 4 in the SAE ranking (sae, 2018)), such as Highway and Traffic Jam Pilots, where a vehicle can drive with limited human supervision. Apollo v7.0.0 (Apollo, 2021) consists of more than 500k lines of C++ code.

A central part of the Apollo implementation is the Cyber RT middleware (cyb, 2019). Cyber RT provides a publish/subscribe pattern to enable the communication between software components running over it. Components can communicate via tagged channels, a.k.a. topics. Components may publish data to topics by writing messages to a named topic and may subscribe to any topic of interest by referring to the topic name. Whenever a publisher writes data to a topic, this data is received by all subscribers. Cyber RT allows more than one component to publish data on a topic, and more than one component to subscribe to it. The announcement of (new) topics

and the subscription of components to topic names are performed by a mechanism called service discovery.

This section describes the designed Apollo model used to demonstrate our methodology. The model focuses on the parts of the code-base that are related to autonomous vehicle features, namely, sensors (Camera, LiDAR), localization, perception, prediction, planning, control, and HMI.

The Apollo system architecture has been modeled in the model-based system engineering tool AutoFOCUS3 (fortiss GmbH, 2022). The model comprises of **9** functions, **61** logical components, **341** ports, transmitting **73** data structures with **361** members, **16** execution units, **12** transmission units, and **6** sensors. We developed an experimental metamodel (Aravantinos et al., 2015) extension in AutoFOCUS3 to describe publish/subscribe communication by means of dedicated *topic* port data types. Due to the lack of space, the remainder of this section describes only selected parts of the logical and platform architecture, since the security results presented in this article mainly focus on the logical architecture and platform architecture.

**Logical Architecture.** The designed logical architecture is complex and consists of four hierarchical levels with multiple components. Figure 2 depicts the second highest level of our Apollo model containing the main autonomous driving components.

The **localization** component receives sensor data from GNSS and computes the vehicle's position. The vehicle's position is received by the following components. The **perception** component receives sensor data from cameras, radars and LiDAR, and the vehicle's position. Perception identifies obstacles, such as other vehicles on the road, as well as the state of traffic lights. The **prediction** component takes the list of obstacles from perception and the vehicle's position, and tries to predict the intention of obstacles, which may be other vehicles or pedestrians. The prediction includes aspects such as whether a vehicle intends to change lanes. The **relative map** component aggregates the list of obstacles and combines it with map data, which contains information about the road, such as lanes and traffic lights. The **planning** component takes as input all the data computed by localization, perception, prediction and relative map. Planning uses this data to plan a safe and comfortable trajectory for the vehicle. The **control** component receives the planned trajectory and produces control commands (steering, acceleration, etc.) for the vehicle to follow the trajectory.

A key challenge was to ensure the faithfulness of the model to the Apollo code. To accomplish this, we

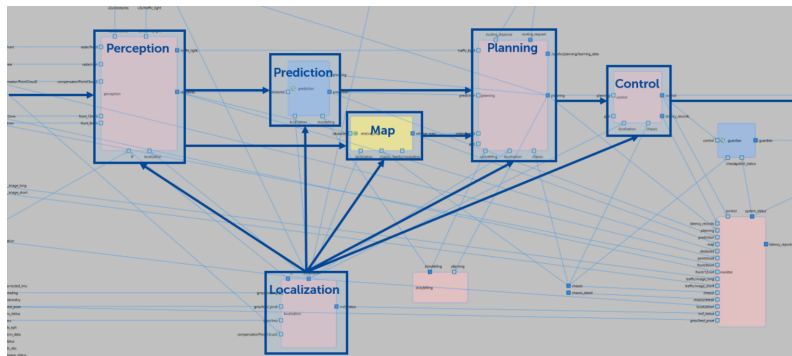


Figure 2: Logical architecture: Main autonomous driving components.

extracted the model elements by manually inspecting the Apollo code. For example, to find all Cyber RT components implemented in Apollo, we inspect the code to find all implementations of the `class cyber::Component`. The next step was to identify the topics and which components publish to them and subscribe to them. The Apollo implementation specifies the topic communication using the following mechanisms: DAG configuration files, C++ code implementing readers for topics and producers of topics, and library code. We inspected each of these mechanisms to map the topics that are subscribed and published to components.

**Platform Architecture.** Figure 3 illustrates our platform architecture that follows the trend for modern smart car architectures consisting of a few, but powerful ECUs and using network interfaces (i.e., switches) between ECUs.

The main ECUs in the platform architecture are: (1) **MDC:** Mobile Data Center: This hardware is responsible for the autonomous function related components, such as inferring objects from camera input, predicting the movement of objects in the environment, planning trajectories. The MDC is further sub-divided into sub-systems with different types of processing units with different levels of safety assurance levels, such as an ASIL-D MCU. (2) **CDC:** Intelligent Cockpit: This hardware is responsible for all the cockpit related functions, such as driver monitoring systems and entertainment functions. (3) **VDC:** Vehicle Controller: This hardware is responsible for the basic vehicle control functions, such as Electric Power Steering, Battery Management, and Anti-lock Braking functions. (4) **VIU 1-4:** Vehicle Integration Units: These hardware are powerful gateways that interface the MDC, CDC, and VDC, connected through network interfaces, to the domain specific hardware connected through CAN buses. The yellow shade in the model represents the system boundary (a.k.a. item boundary). We consider as part

of the system all components that are implemented in the Logical Architecture. For example, Sensors (e.g., LiDAR and GPS radio) are not part of the system itself. They are third-party devices that are connected to the system and provide inputs from the environment. We consider them as public interfaces that are outside and may be accessed by external users.

### 3 SAFETY-INFORMED SECURITY ANALYSIS

Our main focus is to identify assets, damage and threat scenarios related to safety as it is the main concern to autonomous driving. We describe how key safety artifacts are consumed by security analysts to identify key security artifacts, establishing a traceability between security and safety concerns. One can argue from such traces the (relative) completeness with respect to safety of the security analysis in the sense that threats that can cause any one of the safety loss scenarios are identified. As there is existing literature that advocates similar traceability between safety and security (Förster et al., 2019; Dantas and Nigam, 2022a), albeit not using loss scenarios and artifacts mentioned in the ISO 21434 (ISO/SAE 21434, 2020), we simply exemplify the method on examples using the Apollo system architecture. Our mapping from safety artifacts to security artifacts is inspired by the following work (Sabaliauskaite et al., )(Macher et al., 2015)(Förster et al., 2019).

**Safety Analysis.** We carried out a safety analysis for the Apollo system architecture. In compliance to ISO 26262-3 (ISO26262, 2018), we have identified hazards by using Hazard Analysis and Risk Assessment (HARA). Furthermore, we use System Theoretic Process Analysis (STPA) (Leveson and Thomas, 2018) to identify how such hazards may occur.

Relevant for this article are hazards and loss

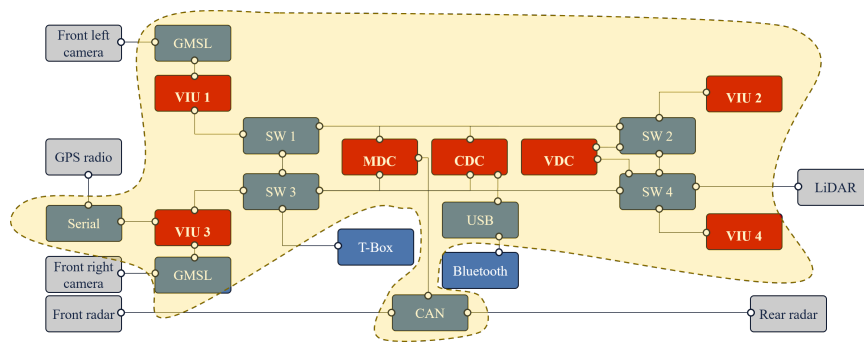


Figure 3: Platform architecture based on modern smart car architecture (yellow shading represents the system boundary).

Table 1: Example of safety analysis results.

|   |  |                                  |                               |
|---|--|----------------------------------|-------------------------------|
| <b>Hazard HZ1</b>                               | Unintended distance between the ego vehicle and other objects. |                                  |                               |
| <b>Severity</b>                                 | Life-Threatening (S3)  | <b>Safety Risk Level: ASIL D</b> |                               |
| <b>Exposure</b>                                 | High Probability (E4)  |                                  |                               |
| <b>Controllability</b>                          | Difficult to Control (C3)                                      |                                  |                               |
| <b>Loss Scenario LS1 that causes Hazard HZ1</b> |  |                                  |                               |
| <b>Source planning</b>                          | <b>Target control</b>  | <b>Message trajectory</b>        | <b>Failure Mode erroneous</b> |

scenarios provided, respectively, by HARA and STPA. A *hazard* is a potential source of loss (e.g., loss of life) caused by malfunctioning behavior of the item (i.e., Apollo system architecture). A *loss scenario* describes the causal factors that may lead to a hazard.

We have identified 4 hazards and 21 loss scenarios. We will use the hazard (HZ1) and loss scenario (LS1) described in Table 1 to demonstrate the model-based methodology for threat analysis described in Section 1.1. HZ1 is a high risk level (ASIL D) related to the autonomous driving functions. LS1 is a possible cause for HZ1. LS1 is traced to two components in the model, planning and control, and to the topic containing the trajectory produced by planning. LS1 specifies that if the computed trajectory is erroneous, e.g., instead of recommending a low acceleration, it recommends a right acceleration, HZ1 may occur, i.e., the vehicle may collide with obstacles.

**Assets and Damage/Threat Scenarios from Safety Analysis.** Following the ISO 21434 (ISO/SAE 21434, 2020), *assets* are objects (e.g., software components, hardware units) for which the compromise of its cybersecurity property can lead to the damage of the item. A *damage scenario* denotes the adverse consequence due to the compromise of a cybersecurity property of an asset. A *threat scenario* denotes the potential actions (or simply attack) on assets that can lead to damage scenarios. The hazards and loss scenarios obtained from the safety analysis

can be directly used to identify such security artifacts related to safety-related damages.

**Damage scenarios** are traced to hazards. The damage scenario traced to HZ1 specifies that unintended distance shall be avoided also from a security perspective. There are three main **assets** that can be traced to the loss scenarios: *Safety Functions*: The safety related functions (typically implemented as pieces of software) shall be protected. For LS1, the functions planning and control are such assets. *Topic/Message*: The safety-related signals/messages mentioned in the loss scenarios shall be protected. For LS1, the topic carrying the trajectory information shall be protected. *Hardware/Physical*: The hardware in which safety functions are deployed shall be protected. The functions associated to LS1 are deployed at the MCU (inside of MDC) hardware unit. Moreover, the failure mode of loss scenarios indicates which cyber-security properties (CIA properties) are associated to the these assets. The failure mode erroneous and loss indicate, respectively, that the integrity and availability of the corresponding assets shall be ensured. Notice that the confidentiality property cannot be extracted from safety analysis as lack of confidentiality does not lead to safety-related damages. From the loss scenario and its derived assets, one can elaborate **threat scenarios** by using, e.g., the STRIDE methodology (Shostack, 2014). For example, the integrity of safety functions and of physical assets can be violated by tampering attacks,

while the integrity of topic/messages can be violated by spoofing and elevation of privilege.

These artifacts are used to enumerate attack paths that shall be considered, namely those that can lead to threat scenarios. The enumeration of attack paths depends on the technology that is being used. For example, if a software may be updated using Over-the-Air mechanisms, then attack paths shall consider how these mechanisms can be exploited to tamper the software with malicious updates. For the Apollo system architecture considered in this article, one needs to consider the use of SOA machinery, e.g., protocols for service discovery, publish-subscribe communication patterns, sensors, and other public interfaces, e.g., Bluetooth and WiFi. These are considered in the next section.

#### 4 INTRUDER MODEL FOR VEHICLE SOA

We formalize an SOA intruder model defined by the rules in Figure 4. The intruder model is based on the main attacks against vehicle SOA with centralized architecture, described in Section 6. Intuitively, SOA contain two main attack surfaces that may be exploited if no suitable countermeasures are deployed.

- *Outsider Attackers* can exploit public interfaces, such as sensors and communication interfaces, to infiltrate the system and attack vehicle assets, such as safety functions. For example, attackers can spoof GPS coordinates thus violating the integrity of published position information by localization.
- *Insider Attackers* can exploit vulnerabilities in the underlying SOA protocols and carry out MITM attacks thus violating the integrity of topics. For example, attackers can carry out MITM attacks between localization and perception to violate the integrity of position information.

Figure 4 introduces the rules of the intruder model reflecting these type of attacks. These inference rules derive three judgments described below.  $\Gamma$  contains system specifications which are extracted from the vehicle model. These specifications are formalized as atomic formulas using the predicate symbols described in Table 2.

$\Gamma \vdash \text{wrt}(X, Y)$  and  $\Gamma \vdash \text{rd}(X, Y)$  denote that the port  $X$  of model element may write, respectively, read on  $Y$ . Rule  $\text{write}_1$  specifies that an output  $eo$  of an ECU may write on an input port  $ni$  of a network element if an output port  $co$  of a component is allocated to  $eo$  (specified by  $\text{cpo}(c, co)$ ,  $\text{alloc}(co, eo)$ ), and there is a channel from  $eo$  to  $ni$  (specified by  $\text{ch}(eo, ni)$ ). Rule

Table 2: Description of the predicates used to define the intruder’s capabilities.

| Predicate                           | Denotation   |
|-------------------------------------|--|
| $\text{ecui}(ecu, ei)$              | ECU $ecu$ and its input port $ei$ .  |
| $\text{ecuo}(ecu, eo)$              | ECU $ecu$ and its output port $eo$ .   |
| $\text{neti}(\text{net}, ni)$       | net. interface $net$ and its input port $ni$ .                               |
| $\text{neto}(\text{net}, no)$       | net. interface $net$ and its input port $no$ .                               |
| $\text{ch}(\text{out}, \text{inp})$ | channel from output port $\text{out}$ to input port $\text{inp}$ .           |
| $\text{wrt}(el1, el2)$              | element $el1$ writes data to $el2$ .   |
| $\text{rd}(el1, el2)$               | element $el1$ reads data from $el2$ .  |
| $\text{cpi}(c, ci)$                 | component $c$ and its input port $ci$ .                                      |
| $\text{cpo}(c, co)$                 | component $c$ and its output port $co$ .                                     |
| $\text{alloc}(el, ecu)$             | element $el$ is allocated to $ecu$ .   |
| $\text{pub}(c, co, tp)$             | component $c$ publishes the topic $tp$ through output port $co$ .            |
| $\text{sub}(c, ci, tp)$             | component $c$ subscribers to the topic $tp$ through input port $ci$ .        |
| $\text{if}(ecu, ci, tp)$            | topic $tp$ is published within $ecu$ through an information flow from $ci$ . |
| $\text{pro}(tp)$                    | topic $tp$ is protected by a cryptographic primitive.                        |
| $\text{publico}(el, po)$            | public $el$ and its output port $po$ .                                       |
| $\text{i\_reach}(el)$               | element $el$ is reachable by the intruder.                                   |
| $\text{i\_attack}(el)$              | $el$ may be attacked by the intruder.  |

$\text{write}_4$  is similar, but for public elements. Rule  $\text{write}_2$  specifies that an input port of an ECU may write to its own output port – we assume that there exists an internal transmission within the ECU ( $\text{ch}(ei, eo)$ ), e.g., components exchanging messages within the ECU. Rule  $\text{write}_3$  is similar, but for network interfaces. Rule  $\text{read}_2$  specifies when an ECU reads from a network interface (similar to  $\text{write}_1$ ). Rule  $\text{read}_1$  specifies that subscriber ports may read from publisher ports.

$\Gamma \vdash \text{i\_reach}(X)$  denotes when a port  $X$  of a model element is reachable by an intruder. Rule  $\text{basic\_out}$  specifies that any port of a public element in the architecture can be reached by the (outsider) intruder. Rule  $\text{reach\_wrt}$  specifies that a port  $p2$  of a model element can be reached by the (outsider) intruder if a port  $p1$  writes on  $p2$ . Respectively,  $\text{reach\_rd}$  specifies

**Write and Read Rules**

$$\frac{\text{cpc}(c, co), \text{alloc}(co, eo), \text{ecuo}(ecu, eo), \text{neti}(\text{net}, ni), \text{ch}(eo, ni) \in \Gamma}{\Gamma \vdash \text{wrt}(eo, ni)} \text{write}_1$$

$$\frac{\text{ecui}(ecu, ei), \text{ecuo}(ecu, eo), \text{ch}(ei, eo) \in \Gamma}{\Gamma \vdash \text{wrt}(ei, eo)} \text{write}_2 \quad \frac{\text{neti}(\text{net}, ni), \text{neto}(\text{net}, no), \text{ch}(ni, no) \in \Gamma}{\Gamma \vdash \text{wrt}(ni, no)} \text{write}_3$$

$$\frac{\text{publico}(el, po), \text{neti}(\text{net}, ni), \text{ch}(po, ni) \in \Gamma}{\Gamma \vdash \text{wrt}(po, ni)} \text{write}_4 \quad \frac{\text{sub}(c1, ci, tp), \text{pub}(c2, co, tp) \in \Gamma}{\Gamma \vdash \text{rd}(ci, co)} \text{read}_1$$

$$\frac{\text{cpi}(c, ci), \text{alloc}(ci, ei), \text{ecui}(ecu, ei), \text{neto}(\text{net}, no), \text{ch}(no, ei) \in \Gamma}{\Gamma \vdash \text{rd}(ei, no)} \text{read}_2$$

**Intruder Reachability Rules**

$$\frac{\text{publico}(el, po) \in \Gamma}{\Gamma \vdash i\_reach(po)} \text{basic\_out} \quad \frac{\text{pub}(c, co, tp) \in \Gamma}{\Gamma \vdash i\_reach(co)} \text{basic\_ins}$$

$$\frac{\Gamma \vdash \text{wrt}(p1, p2) \quad \Gamma \vdash i\_reach(p1)}{\Gamma \vdash i\_reach(p2)} \text{reach\_wrt} \quad \frac{\Gamma \vdash \text{rd}(p2, p1) \quad \Gamma \vdash i\_reach(p1)}{\Gamma \vdash i\_reach(p2)} \text{reach\_rd}$$

$$\frac{\text{pub}(c, co, tp), \text{sub}(c, ci, tp1), \text{pub}(c1, co1, tp1) \in \Gamma \quad \Gamma \vdash \text{rd}(ci, co1) \quad \Gamma \vdash i\_reach(co)}{\Gamma \vdash i\_reach(ci)} \text{reach\_ins\_rd}$$

**Intruder Attack Rules**

$$\frac{\text{if}(ecu, p, tp) \in \Gamma \quad \Gamma \vdash i\_reach(p)}{\Gamma \vdash i\_attack(tp)} \text{at\_out}$$

$$\frac{\text{sub}(c1, ci, tp), \text{pub}(c, co, tp), \neg \text{pro}(tp) \in \Gamma \quad \Gamma \vdash i\_reach(ci) \quad \Gamma \vdash i\_reach(co)}{\Gamma \vdash i\_attack(tp)} \text{at\_ins}$$

Figure 4: Intruder model for SOA.

that a port  $p2$  of a model element can be reached by the (outsider) intruder if  $p2$  reads on a port  $p1$ . Rule  $\text{basic\_ins}$  specifies that any publisher port in the architecture can be reached by the (insider) intruder. Rule  $\text{reach\_ins\_rd}$  specifies that the (insider) intruder can reach a subscriber port  $ci$  if  $ci$  reads on a reached publisher port  $co$ .

$\Gamma \vdash i\_attack(X)$  denotes when a topic  $X$  can be attacked. Rule  $\text{at\_out}$  specifies that any topic published within an information flow ( $\text{if}(ecu, p, tp)$ ) from a reached ECU's input port may be attacked. Rule  $\text{at\_ins}$  specifies that any topic between publisher and subscriber ports reached by the (insider) intruder may be attacked if the topic is not protected.

**Outsider Intruder (Example).** Consider the platform architecture depicted in Figure 5. The black and white circles connected to hardware units are, respectively, output and input ports. We assume that  $\text{Sensor}$  is a public interface. The output port  $o1$  of

$\text{Sensor}$  can be reached by the intruder based on the rule  $\text{basic\_out}$ . The output port  $o1$  writes on the input port  $i1$  of the network interface  $\text{Network1}$ , then based on  $\text{reach\_wrt}$  the intruder can reach  $i1$  and  $o2$ . We assume that the subscriber port (light blue square) of component  $\text{CP1}$  is allocated to the input port  $i2$  of  $\text{ECU1}$ , and that  $i2$  reads from  $o2$ . The intruder can then reach  $i2$  and  $o3$  based on  $\text{reach\_rd}$ . Neither  $i3$  nor  $o4$  can be reached by the intruder. The intruder can reach  $i4$  as  $o3$  writes to  $i4$ . The intruder cannot reach  $i5$  and  $o5$ . Finally, an intruder may carry out, e.g., a spoofing attack from  $\text{Sensor}$  to violate the integrity of the topics published by either  $\text{CP1}$  or  $\text{CP2}$  since there is an information flow from  $i2$  ( $\text{at\_out}$ ).

**Insider Intruder (Example).** Consider the logical architecture depicted in Figure 6. The dark and light blue squares connected to components are, respectively, publisher and subscriber ports. The intruder can reach all publisher ports  $o1 \dots o6$  based

on `basic_ins`. Based on `reach_ins_rd`, the intruder can reach the subscriber ports `i1...i7`, as these ports read from publishers, e.g., `i7` reads from `localization` via port `o5`. The intruder cannot reach the subscriber port `i8`, as `infotainment` is not a publisher. We assume the topics published by ports `o4` and `o5` are protected. Assume the topic published by `planning` through port `o2` is the intruder's target. As a result, the intruder has the following options to carry out MITM attacks. An attack may be carried out between `routing` and `planning` or even between `perception` and `prediction` given that the topic published by `perception` may affect the topic published by `planning`. The intruder can neither carry out attacks between `localization` and `planning` (same for `perception` and `prediction`), nor between `prediction` and `planning` since the topics are protected (`at_ins`). The intruder cannot carry out attacks from `infotainment`.

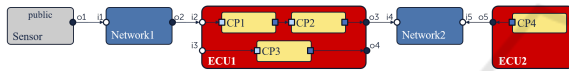


Figure 5: Illustration of the outsider intruder.

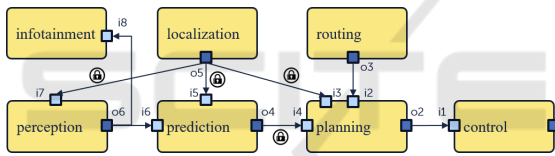


Figure 6: Illustration of the insider intruder.

Table 3: Number of identified attack paths and the execution time taken by LAUFEN to computed the attack paths.

| Intruder | #Attack Paths | Execution time (s) |
|----------|---------------|--------------------|
| Outsider | 152           | 1.11               |
| Insider  | 94            | 0.06               |

## 5 AUTOMATING ATTACK PATH ANALYSIS

LAUFEN (vehicLe threAt analysis aUtomation For sErvice-orieNted architectures) is an SOA machinery that enables the automated computation of several activities of the Threat Assessment and Remediation Analysis (TARA) analysis. This section focuses on the automated computation of attack paths that can cause threat scenarios to vehicle SOA, i.e., the paths that violate cybersecurity properties of assets (Section 3). To this end, LAUFEN implements the proposed intruder model in the logic programming

tool DLV (Leone et al., 2006). LAUFEN encodes the system specification as facts using the predicates described in Table 2, and the intruder model described in Section 4. Then the DLV solver is used to enumerate the attack paths. We validate LAUFEN on the modeled Apollo system architecture. The implementation and the experimental results are available at (Dantas and Nigam, 2022b).

Given the high complexity of the Apollo model, naively computing the attack paths based on reachability does not scale, in particular for the outsider intruder. To address this problem, the computation is divided into two steps. The first step, *Intruder reachability*, computes all the model elements that are reachable by the intruder as specified by the write and read, and reachability rules. Since no paths are computed, the DLV engine computes the reachable elements in the range of milliseconds. We then use the reachable elements as input to the second step, *Path computation*, where we make use of the attack rules. Instead of enumerating all paths, we proceed using a goal-oriented search to enumerate only the attack paths on assets (a.k.a. asset-centric approach). This means that DLV does not require to compute all paths.

We run the experiments on a 1.90GHz Intel Core i7-8665U with 16GB of RAM running Ubuntu 18.04 LTS with kernel 5.4.0-113-generic and DLV 2.1.1. Table 3 shows the number of identified attack paths, and the execution time of LAUFEN. The execution time in enumerating the attack paths is rather low, i.e., 1.11 and 0.06 seconds for the outsider and insider intruder, respectively. The number of identified attack paths is high due the complexity of the system, e.g., the great number of public elements and the great number of information flows in the architecture. We do not rule out any attack path to guarantee a complete coverage of possible steps exploited by the intruder. Section 5.1 elaborates on countermeasures that may mitigate several of the identified attack paths.

We analyzed the generated attack paths w.r.t. potential attacks against safety-critical topics. Table 4 organizes selected attack paths into attacks carried out by an outsider attacker and insider attacker.

Firstly, our analysis was able to identify several attacks that have been reported in the literature, namely those attacks associated with a citation. The set of attack paths computed includes attacks carried out by outsider attackers exploiting Bluetooth, LiDAR, Camera, GPS and Radar that may target safety-critical topics, cause loss scenarios and harm to road-users, as well as by insider attackers exploiting SOA communication vulnerabilities to target topics.

Secondly, we also identified potential attacks that up to the best of our knowledge have not been reported

Table 4: Potential attacks derived from selected attack paths. *From* and *To* denote, respectively, the model element where the attack starts and ends. *Affected Topic* denotes the actual target of the attacker. The upper and lower part of the table describe, respectively, selected attacks carried out by the outsider and insider intruder, incl. attacks reported in the literature. NA denotes attacks that up to the best of our knowledge have not been reported in the literature. *#Attack Paths* denotes that number of computed attack paths *From* and *To*.

| From                     | To                 | Affected Topic    | Article                     | #Attack Paths |
|--------------------------|--------------------|-------------------|-----------------------------|---------------|
| <b>Outsider Intruder</b> |                    |                   |                             |               |
| Bluetooth                | VDC                | signal            | (Chowdhury et al., 2020)    | 3             |
| LiDAR                    | MCU                | obstacles         | (Hau et al., 2021)          | 24            |
| Front Left Camera        | MCU                | obstacles         | (Jha et al., 2020)          | 18            |
| GPS                      | MCU                | localization pose | (Shen et al., 2020)         | 18            |
| Front Radar              | MCU                | obstacles         | (Komissarov and Wool, 2021) | 6             |
| T-Box                    | MCU                | traffic light     | NA                          | 18            |
| <b>Insider Intruder</b>  |                    |                   |                             |               |
| gnss driver              | velodyne detection | tf                | (Hong et al., 2020)         | 1             |
| gnss driver              | msf localization   | gnss best pose    | (Hong et al., 2020)         | 1             |
| compensator              | velodyne detection | pointcloud2       | (Hong et al., 2020)         | 1             |
| control                  | chassis            | signal            | (Hong et al., 2020)         | 1             |
| chassis                  | gnss driver        | chassis           | (Hong et al., 2020)         | 1             |
| v2x proxy                | traffic light      | traffic light     | NA                          | 1             |
| routing                  | planning           | routing response  | NA                          | 1             |
| relative map             | planning           | map               | NA                          | 1             |

in the literature (marked with NA). We analyzed in further detail some of the attacks by using the model and its connection to the Apollo code to find out how such attacks can lead to safety problems.

```
void TrafficLightsPerceptionComponent::OnReceiveImage(
    const std::shared_ptr<apollo::drivers::Image> msg,
    const std::string& camera_name) { ...
    /** Set traffic light status based on camera data */
    traffic_light_pipeline->Perception(
        camera_perception_options_, frame_.get()); ...
    /** Overwrites traffic light status if valid v2x data */
    SyncV2XTrafficLights(frame_.get()); ... }
```

Figure 7: Overwriting traffic light status with V2X data.

**V2X Traffic Light Overwrite Attack.** LAUFEN has identified attack paths targeting `v2v proxy` from both outside and inside. Figure 8 illustrates the attack from the outside. The `v2v proxy` component publishes data on the traffic light status obtained from the road infrastructure. This data is subscribed by the `traffic light` component which also subscribes data from the cameras to identify and publish the traffic light status. Since there is a traceability from the Apollo model and the Apollo source code, it is straightforward to find the relevant classes for vulnerabilities. Indeed, we found out that the function `TrafficLightsPerceptionComponent` gives priority to the data received by `v2v proxy` over the data received by the cameras. Figure 7 shows a code snippet from the `TrafficLightsPerceptionComponent` function. As

a result, an attacker may manipulate the traffic light status from either outside (i.e., spoofing attack) or inside (i.e., MITM attack). As illustrated by Figure 8, a spoofing attack from T-Box manipulating the traffic light status can cause serious harm to passengers and pedestrians as the vehicle can cross a red-light.

**Route/Mapping Injection Attack.** The seriousness of some of the identified attack paths may not be too obvious from a safety perspective. For example, LAUFEN has identified the following attack path: {routing, planning} targeting the routing response topic. An insider attacker may carry out a MITM attack between routing and planning to provide a malicious route for the ego vehicle. From a safety perspective, a loss scenario of type erroneous from routing to planning may be easy to control, and hence it would lead to a low criticality hazard (e.g., ASIL A or B). From a security perspective, however, there are several serious consequences, including hijacking of passengers. The planning component may also be affected by the road map published by relative map, as planning takes the map into account while computing the vehicle trajectory.

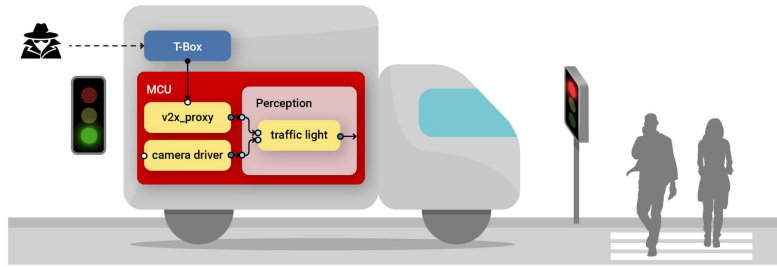


Figure 8: Illustration of a spoofing attack from T-Box to manipulate traffic light status received by v2v proxy.

## 5.1 Potential Countermeasures

We have performed an attack path analysis to deduce potential locations for instantiating security countermeasures. Our analysis focused on the computed attack paths using the outsider intruder. This choice was made because we noticed that many of such attack paths have the same prefix, which may be a hint for instantiating security countermeasures.

Table 5 presents the main results of our attack path analysis. Specifically, Table 5 shows the public element that can be reached the outsider intruder, the number of attack paths computed by LAUFEN from the public element, and the common prefix for all attack paths from the same public element.

Table 5: Attack paths analysis (outsider intruder).

| Public element     | #Attack Paths | Prefix                            |
|--------------------|---------------|-----------------------------------|
| Front Left Camera  | 21            | Front Left Camera → GMSL → VIU 1  |
| Front Right Camera | 21            | Front Right Camera → GMSL → VIU 3 |
| GPS                | 21            | GPS → Serial → VIU 3              |
| Front Radar        | 10            | Front Radar → CAN → MDC           |
| Rear Radar         | 10            | Rear Radar → CAN → MDC            |
| LiDAR              | 27            | LiDAR → SW4                       |
| Bluetooth          | 21            | Bluetooth → USB → CDC             |
| T-Box              | 21            | T-Box → SW3                       |

The last architecture element described in the Prefix column may be a suitable location to instantiate a countermeasure and consequently address the attack

paths. From the Prefix column, we can also notice that the gateway VIU 3 is a common location in the attack paths from both Front Right Camera and GPS. Similarly, the connection between CAN and MDC is a common location in the attack paths from Front Radar and Rear Radar. This gives us a hint that security countermeasures could be placed in front of VIU 3, and between CAN and MDC to address such attack paths (specifically, 62 attack paths).

Firewalls are, e.g., recommended (Cheng et al., 2019) as means to protect vehicle architectures against such attacks. They may be deployed in front of the last architecture elements in the Prefix column to filter network traffic and prevent malicious intrusion. For the network interfaces (i.e., T-Box and Bluetooth), one could also implement a mutual authentication mechanism (e.g., mTLS) to ensure that only authenticated messages are accepted.

Safety architecture patterns, such as Heterogeneous Duplex pattern (Armoush, 2010), may also be deployed as a second-layer of defense. Consider, e.g., the V2X traffic light overwrite attack carried by an outsider attacker. This attack violates the integrity of traffic light topic through T-Box. A possible countermeasure is to include a checker in the traffic light component to consider inputs from both v2x proxy and cameras (i.e., heterogeneous inputs) – the traffic light component emits an alert to the driver or transition the system to a safe state if the inputs do not match.

The MITM attacks (e.g., the route injection attack) carried by an insider attacker exploit SOA communication vulnerabilities to violate the integrity of topics. Digital signatures are a well-known countermeasure for ensuring authenticity and integrity between servers (e.g., publishers) and clients (e.g., subscribers). To address MITM attacks, one can implement digital signatures in the Apollo system, where each publisher originator signs its message, and each subscriber of the message verifies the signature of the message. Fast-DDS provides a cryptographic plugin for message authentication codes computation and verification. The use of digital



certificates to address MITM attacks in Apollo was inspired by (Hong et al., 2020) that proposed a countermeasure using digital signatures for mitigating publisher-subscriber overprivilege issues in Apollo.

All 94 attack paths (insider intruder) are, in principle, addressed upon implementing digital signatures. The decision of using digital signatures causes, however, a performance penalty at the execution time of software components. The performance penalty can then be analyzed according to several points of view, including security, safety and financial. However, since all of the identified attack paths are safety critical, countermeasures shall be implemented to ensure vehicle safety.

## 6 RELATED WORK

Due to the lack of space, we moved some parts of the related work to (Dantas et al., 2022).

**Attacks Against Vehicle SOA.** The following work has inspired us to formalize the intruder model for vehicle SOA. A recent systematization of knowledge article (Shen et al., 2022) gives an overview of the state-of-the-art of the literature. In “Drift with the devil” (Shen et al., 2020) it is shown that an intruder may manipulate location information by spoofing GPS radio signals. LiDAR sensor signals may be spoofed to remove obstacles on the road (Hau et al., 2021). Camera signals may also be spoofed to manipulate video frames given that the camera traffic is transmitted in plain text (Jha et al., 2020). An attack may exploit vulnerabilities in a Bluetooth stack weakness to lock the brakes of the vehicle (Chowdhury et al., 2020). In the work by (Zelle et al., 2021), the authors investigate possible security issues in the service discovery mechanism of vehicle SOA, in particular SOA using the SOME/IP protocol, enabling the attacker to carry out MITM attacks between publishers and subscribers. In the work on AVGuardian (Hong et al., 2020), the authors investigated possible publisher/subscriber overprivilege instances in Apollo.

Our intruder model specifies the main attacker’s capabilities needed to carry out the above attacks at the architecture level, including the capabilities of attackers to carry out (a) spoofing attacks from outside (e.g., from sensors), and (b) MITM attacks from inside (e.g., between components), thus violating the integrity of safety-critical topics. The attacks exploiting overprivileged instances can be seen as a specific case of the MITM attack.

**Automate Threat Analysis.** A survey on threat modeling (Xiong and Lagerström, 2019) has shown that most threat modeling work remains to be done manually. We briefly describe some of the security/threat analysis tools that provide computed-aided support in the automotive domain.

AVGuardian (Hong et al., 2020) is a static analysis tool to detect overprivilege instances in source code implementing service-oriented architectures for automotive systems. AVGuardian examines each module’s source code and automatically detects publisher and subscriber overprivilege instances in the fields of topics defined by the module. AVGuardian requires the behavior specification of the system to detect overprivilege instances. LAUFEN has been implemented to identify threats and attack paths during the design of the system architecture without the behavior specification of the system. LAUFEN was able to automatically compute attack paths that may lead to the attacks detected by the AVGuardian tool.

ProVerif (Blanchet et al., 2022) and Tamarin Prover (Basin et al., 2022) are well-known automated reasoning tools to verify the security properties of systems (in particular, security protocols) with the Dolev-Yao intruder model (Dolev and Yao, 1983). These reasoning tools require the formal specification of the behavior of the system to verify its properties. A promising future work direction is to include the behavior specification in our Apollo model and use such reasoning tools to verify security properties of SOA protocols such as SOME/IP or DDS.

An attack propagation method that targets automotive safety-critical functions has been proposed by (Fockel et al., 2022). The commercial tool ThreatGet (thr, 2022) enables the identification of attack paths following ISO 21434. Microsoft SDL Threat Modeling tool (sdl, 2022) is another well-known commercial tool to compute threats. The threats are computed using STRIDE. The attack path associated to each compute threat is represented using data flow diagrams. To the best of our knowledge, these tools do not support intruder model capabilities for vehicle SOA. This article advances the state-of-the-art by proposing a machinery built upon realistic formalized intruder models for vehicle SOA.

## 7 CONCLUSION

This article proposed automated methods for threat analysis using a model-based engineering methodology. To this end, we have (a) modeled a faithful vehicle model of the autonomous driving functions of the Apollo framework, (b) formalized

an intruder model for vehicle SOA based on a literature review, and (c) developed LAUFEN, an SOA machinery for computing several activities of a threat analysis, including attack paths.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for valuable comments. We thank Anait Boyajyan for the assistance provided with the figures used in this article.

## REFERENCES

- (2018). SAE International Releases Updated Visual Chart for Its “Levels of Driving Automation” Standard for Self-Driving Vehicles.
- (2019). Apollo Cyber RT. Available at <https://cyber-rt.readthedocs.io/>.
- (2022). Microsoft SDL Threat Modeling Tool. Available at <https://www.microsoft.com/en-us/securityengineering/sdl/threatmodeling>.
- (2022). ThreatGet - Threat Analysis and Risk Management. Available at <https://www.threatget.com/>.
- Apollo (2021). An Open Autonomous Driving Platform. <https://github.com/ApolloAuto/apollo>.
- Aravantinos, V., Voss, S., Teuffl, S., Hölzl, F., and Schätz, B. (2015). AutoFOCUS 3: Tooling Concepts for Seamless, Model-based Development of Embedded Systems. In *ACES-MB'15*.
- Armoush, A. (2010). *Design Patterns for Safety-Critical Embedded Systems*. PhD thesis, RWTH Aachen University.
- Basin, D., Cremers, C., Dreier, J., Meier, S., Sasse, R., and Schmidte, B. (2022). Tamarin Prover <https://tamarin-prover.github.io/>.
- Blanchet, B., Cheval, V., Allamigeon, X., Smyth, B., and Sylvestre, M. (2022). ProVerif <https://bblanche.gitlabpages.inria.fr/proverif/>.
- Cheng, B. H. C., Doherty, B., Polanco, N., and Pasco, M. (2019). Security Patterns for Automotive Systems. In *MODELS'19*.
- Chowdhury, A., Karmakar, G. C., Kamruzzaman, J., Jolfaei, A., and Das, R. (2020). Attacks on Self-Driving Cars and Their Countermeasures: A Survey. *IEEE Access*.
- Dantas, Y. G., Barner, S., Ke, P., Nigam, V., and Schoepp, U. (2022). Technical Report: Automating Vehicle SOA Threat Analysis using a Model-Based Methodology. Technical report.
- Dantas, Y. G. and Nigam, V. (2022a). Automating Safety and Security Co-Design through Semantically-Rich Architectural Patterns. *ACM Trans. Cyber Phys. Syst.*
- Dantas, Y. G. and Nigam, V. (2022b). <https://github.com/ygdantas/LAUFEN>.
- Dolev, D. and Yao, A. C. (1983). On the security of public key protocols. *IEEE Trans. Inf. Theory*, 29(2):198–207.
- Fockel, M., Schubert, D., Trentinaglia, R., Schulz, H., and Kirmair, W. (2022). Semi-automatic integrated safety and security analysis for automotive systems. In *MODELSWARD'22*.
- Förster, D., Loderhose, C., Bruckschlägl, T., and Wiemer, F. (2019). Safety goals in vehicle security analyses: a method to assess malicious attacks with safety impact. In *the 17th escar Europe - Embedded Security in Cars*.
- fortiss GmbH (2022). AutoFOCUS 2.21.
- Hau, Z., Co, K. T., Demetriou, S., and Lupu, E. C. (2021). Object removal attacks on lidar-based 3d object detectors. *CoRR*, abs/2102.03722.
- Hong, D. K., Kloosterman, J., Jin, Y., Cao, Y., Chen, Q. A., Mahlke, S. A., and Mao, Z. M. (2020). AVGuardian: Detecting and Mitigating Publish-Subscribe Overprivilege for Autonomous Vehicle Systems. In *EuroS&P'20*.
- ISO26262 (2018). ISO 26262, road vehicles — functional safety — part 6: Product development: software level.
- ISO/SAE 21434 (2020). Road vehicles - cybersecurity engineering.
- Jha, S., Cui, S., Banerjee, S. S., Cyriac, J., Tsai, T., Kalbarczyk, Z., and Iyer, R. K. (2020). ML-Driven Malware that Targets AV Safety. In *DSN 2020*.
- Komissarov, R. and Wool, A. (2021). Spoofing attacks against vehicular FMCW radar. In *ASHES@CCS'21*.
- Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. (2006). The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7.
- Leveson, N. G. and Thomas, J. P. (2018). *STPA Handbook*.
- Macher, G., Sporer, H., Berlach, R., Armengaud, E., and Kreiner, C. (2015). SAHARA: A Security-aware Hazard and Risk Analysis Method. In *DATE'15*.
- Sabaliauskaite, G., Liew, L. S., and Cui, J. C.
- Shen, J., Wang, N., Wan, Z., Luo, Y., Sato, T., Hu, Z., Zhang, X., Guo, S., Zhong, Z., Li, K., Zhao, Z., Qiao, C., and Chen, Q. A. (2022). Sok: On the semantic AI security in autonomous driving. *CoRR*, abs/2203.05314.
- Shen, J., Won, J. Y., Chen, Z., and Chen, Q. A. (2020). Drift with Devil: Security of Multi-Sensor Fusion based Localization in High-Level Autonomous Driving under GPS Spoofing. In *USENIX'20*.
- Shostack, A. (2014). *Threat Modeling: Designing for Security*. Wiley.
- SOTIF, I. . (2021). Safety of the Intended Functionality.
- UN (2021). UN Regulation No. 155 - Cyber security and cyber security management system.
- WIRED (2015). Hackers remotely kill a jeep on the highway-with me in it. Available at <https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/>.
- Xiong, W. and Lagerström, R. (2019). Threat modeling - A systematic literature review. *Comput. Secur.*, 84:53–69.
- Zelle, D., Lauser, T., Kern, D., and Krauß, C. (2021). Analyzing and Securing SOME/IP Automotive Services with Formal and Practical Methods. In *ARES'21*.

# Chapter 7

## Knowledge Representation and Reasoning for Security System Architectures

Chapter 7 proposes a Domain-Specific Language (DSL) for specifying system architecture artifacts, security artifacts, and security architecture patterns. Security reasoning rules have been specified to enable the automation of security activities, including the recommendation of security architecture patterns. The DSL and security reasoning rules have been implemented as a dedicated logic programming tool, which has been validated using an example taken from the unmanned air vehicle domain.

**Contributing article:** Yuri Gil Dantas and Ulrich Schöpp: SeCloud: Computer-Aided Support for Selecting Security Measures for Cloud Architectures. ICISSP 2023: 264-275

**Copyright information:** Science and Technology Publications Lda (SCITEPRESS), 2024. <https://doi.org/10.5220/0011901900003405>.

**Author contributions:** The concept for the publication was jointly developed by Yuri Gil Dantas and Ulrich Schöpp. They had several research discussions about specifying semantically-rich architecture patterns for cloud architectures used in the unmanned air vehicle domain. Yuri Gil Dantas implemented the tool and carried out the experiments presented in the article. Yuri Gil Dantas took the lead in writing the initial draft of the article, excluding Section 2 that was written by Ulrich Schöpp. Ulrich Schöpp assisted in improving the article. Yuri Gil Dantas handled subsequent revisions and corrections.

# SeCloud: Computer-Aided Support for Selecting Security Measures for Cloud Architectures

Yuri Gil Dantas and Ulrich Schöpp

Fortiss GmbH, Munich, Germany

**Keywords:** Securing Cloud Architectures, Security Architecture Patterns, Automation.

**Abstract:** The adoption of cloud infrastructures requires the deployment of security measures to protect assets against threats (e.g., tampering). Several security measures/technologies are available for securing cloud infrastructures, such as Service Mesh Istio and OpenID Connect. In the current state of practice, the selection of security measures is manually done by an expert (e.g., a security engineer). It becomes challenging for experts to make these selections due to the complexity of cloud infrastructures and the vast number of available security measures and technologies. This article proposes a tool for automating the recommendation of security measures for cloud architectures. Our tool expects as input information both the cloud architecture and assets identified during a threat analysis, and recommends security measures for protecting such assets against threats. We validate our tool in a case study that provides cloud services for unmanned air vehicles (UAVs).

## 1 INTRODUCTION

Cloud infrastructures offer runtime environments with sophisticated mechanisms for reliability, observability, manageability and security. These infrastructures provide several benefits for business and IT, including lower implementation and maintenance costs.

Security is one of the biggest concerns about cloud infrastructures, especially because the data is no longer controlled by the client who purchased the cloud service. Indeed, a literature review conducted by (Carroll et al., 2011) confirms that security is the main risk for businesses using cloud infrastructures.

Two attack surfaces against cloud systems have brought the attention of security researchers and engineers: 1) External attackers may carry out attacks against cloud services (Eliseev et al., 2021). For example, without suitable security measures, an external attacker may carry out spoofing attacks to impersonate a legitimate client in accessing critical data. 2) An internal service may also be the source of attacks due to, e.g., misconfiguration, compromise, or being intentionally malicious (Oleshchuk and Kjøien, 2011). To implement an effective defense-in-depth strategy, it is important to consider also threats that originate from internal services. Internal services may, e.g., carry out elevation-of-privilege attacks to access data from critical services without authorization. Potential attacks through these attack surfaces shall be mitigated by se-

lecting and deploying suitable security measures for cloud infrastructures.

Before selecting security measures, security engineers perform a threat analysis to identify assets, threat scenarios, and attack paths leading to threat scenarios. Threat analysis is recommended for the early stages of the system development, i.e., during the design of the cloud architecture to avoid expensive changes later in the cloud system lifecycle.

The identification of potential security measures to mitigate the identified threat scenarios is a main challenge for cloud architectures. Examples of technologies offering security measures are Service Mesh Istio, the authentication protocol OpenID Connect, and the Kubernetes network plugin Cilium. In the current state of practice, the selection of security measures is manually done by an expert (e.g., a security engineer).

There are many cloud technologies and platforms to choose from, and it is hardly feasible to evaluate the implications of many possible choices in detail, e.g., due to time constraints. Even when the technologies are understood in principle, it is not easy to keep track of the consequences of selecting a combination of them. One would also like to understand the trade-offs between different choices regarding system development and operation, e.g., in the form of additional requirements for certificate management, or regarding resource overhead.

*The cloud native landscape is vast, and it's*

easy to become overwhelmed by its growing number of competing and overlapping platforms and technologies.<sup>1</sup>

Figure 1 may give an impression of the vastness.

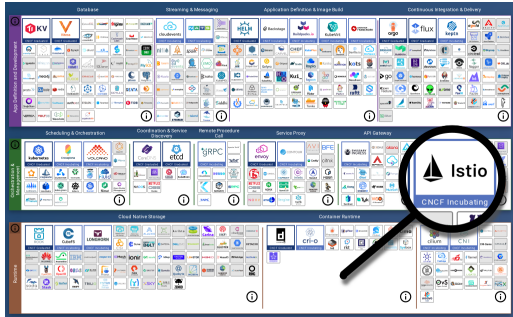


Figure 1: Cloud Native Technology Landscape (excerpt).

**Contribution:** This article proposes SECLLOUD, a tool for automating the recommendation of security measures for cloud architectures. (i) SECLLOUD computes threat scenarios for assets provided as input information by the user. (ii) SECLLOUD computes attack paths based on an intruder model, and the cloud architecture received as input information. (iii) SECLLOUD recommends security measures for mitigating threat scenarios (i.e., addressing all attack paths leading to threat scenarios). The target user for SECLLOUD is security engineers responsible for assessing the security of cloud architectures and hardening such architectures with security measures. SECLLOUD has been implemented in the logic programming engine clingo (Potassco, 2022).

SECLLOUD is built on the work by (Dantas and Nigam, 2022), who proposed a tool for automating the recommendation of security architecture patterns for autonomous vehicle architectures. The original aspects of our work are: (i) extension of the domain-specific language for cloud architectures, (ii) specification of security measures suitable for cloud architectures, (iii) more specific reasoning principles for recommending security measures. The work by (Dantas and Nigam, 2022) only considers the cybersecurity property satisfied by the pattern as the main condition to recommend patterns, and (iv) specification of constraints to reduce the number of recommended solutions with security measures to deal with scalability and usability issues.

SECLLOUD is available online at (SeCloud, 2022).

**Structure of this Article:** The remainder of this article is structured as follows. Section 2 describes a

<sup>1</sup><https://www.cncf.io/blog/2020/09/15/top-7-challenges-to-becoming-cloud-native>

running example to help us introduce the contributions of the article. Section 3 describes an intruder model for cloud infrastructures. Section 4 describes the workflow of SECLLOUD, including its inputs and output artifacts. Section 5 describes the domain-specific language of SECLLOUD, including how SECLLOUD specifies security measures to enable their automation through the reasoning rules described in Section 6. Section 7 illustrates the benefits of using constraints and the increased automation enabled by SECLLOUD. We conclude the article by discussing related and future work in Sections 8 and 9.

## 2 RUNNING EXAMPLE

SECLLOUD is intended to assist security engineers with security architecture decisions for cloud infrastructures. We present it using a real application developed in a research project as a running example.

The example application provides services by unmanned air vehicles (UAVs), such as transportation services or search and rescue services. It optimizes the usage of UAV and implements planning, optimization and prognostic health management functions. While the particular details of the use-case are not important to describe the application of SECLLOUD, the application provides a realistic use-case.

The role of SECLLOUD is to support the selection of technologies to implement security functions at an early stage in the system development, where the main components and their interfaces have been identified, but where the security architecture of the system is still under consideration.

Figure 2 gives an overview of the logical architecture of the system. The main system components are Service Broker (sb), Multi Resource Manager (mrm), Cognitive Assistant (ca), Operations Manager (om), Fleet Manager (fm). These components are intended to be deployed on a cloud platform. They provide public interfaces where clients may access the services.

The applications communicate with each other through two mechanisms. First, the components offer a Rest/HTTP API for access to resources. Second, they use Kafka<sup>2</sup> for event-based communication. Kafka implements topic-based pub/sub communication. It provides named topics where the application components may publish messages about certain aspects, e.g. the topic `av-updates` is intended for messages pertaining to the status of air vehicles. These messages are delivered to all components that subscribe to them. For example, the Operations Manager receives telemetry

<sup>2</sup><https://kafka.apache.org/>

data from the ground control station and publishes this to `av-updates`. The Fleet Manager subscribes to this topic and thus receives telemetry updates. Pub/sub communication is convenient for decoupling components, but also introduces challenges for security, e.g. for limiting the impact of the compromise of one internal component.

The main application components are each realized by a set of Docker containers. Figure 3 shows a container architecture of the system. It contains the application containers and support containers, such as for ingress. While, in realistic deployments, Kafka will likely be deployed redundantly, it suffices to consider it as a single container for our purposes.

Figure 3 shows assumptions about the deployment and the security environment of the system. We consider client a public, external component, to which attackers have full access. Most of the components are intended to be hosted on a cloud platform, where ingress serves as a gateway component. While ground control station is part of the system, it needs to be hosted on-site at an airport rather than in the cloud.

The container architecture of Figure 3 represents an early stage of system development. It does not contain any security features. It could be deployed directly using Docker, but the components would communicate over plain http without authentication or authorization. The purpose of SECLLOUD is to support the design of a security architecture for the application.

### 3 INTRUDER MODEL

This section describes an intruder model for cloud infrastructures. The intruder model is taken into account by SECLLOUD when computing attack paths. We consider both external and internal intruders based on the Dolev-Yao intruder (Dolev and Yao, 1983). Both intruder models are inspired by related work, e.g., (Oleshchuk and Kjøien, 2011; Eliseev et al., 2021).

**External Intruder.** The external intruder assumes that public interfaces may be exploited by attackers, as in (Eliseev et al., 2021). As a result, the external intruder may inject malicious data into the cloud infrastructure through public interfaces (e.g., cloud consumers) to violate the cybersecurity property of assets. Consider, e.g., the architecture described in Section 2, where the interface from client to `sb` represents a public interface of the system, and the container `sb` is an asset. A malicious attacker may carry out spoofing attacks impersonating a legitimate client to write unauthorized data to `sb`, thus violating the authenticity properties of data arriving at `sb`. We assume that an attack from

a public interface to an asset may be possible if there exists an information flow from the public interface to the asset. This is the case for the example above, as shown in the cloud architecture illustrated in Figure 3.

**Internal Intruder.** The internal intruder assumes that internal containers may not be trustworthy, as in (Oleshchuk and Kjøien, 2011). As a result, an internal container may inject malicious data into other containers (possibly assets) in the cloud system. For example, `sb` may not be trustworthy and carry out elevation of privilege attacks to access data from other assets like `mrm` without authorization. We assume that an attack from an internal container to another asset container may be possible if there exists a communication channel from the internal container and to the asset. This is the case for `sb` and `mrm`, as illustrated in Figure 3.

## 4 SeCloud: OVERVIEW

Our goal is to provide automated methods for the selection of security measures for cloud architectures. To this end, we build on SecPat proposed by (Dantas and Nigam, 2022). We propose the use of Knowledge Representation and Reasoning (KRR) (Baral, 2010) for representing cloud architectures, security artifacts, and security measures as knowledge bases. The security measures are recommended in an automated fashion through the specification of reasoning rules. The representation of the knowledge bases are realizable through a domain-specific language (DSL). We specify rules to reason about the security of the cloud architecture, including rules to enable the automated recommendation of security measures. Our tool – SECLLOUD – implements both the DSL and reasoning rules for securing cloud architectures. SECLLOUD has been developed in the logic programming engine `clingo` (Potassco, 2022).

Figure 4 illustrates the workflow of SECLLOUD. The gray boxes represent either artifacts received as input or generated for output. SECLLOUD receives two main artifacts as input, namely cloud architecture and security artifacts.

**Architecture Model.** This input artifact consists of a Logical Architecture, a Container Architecture, and an Allocation Table. An allocation table denotes the mapping of logical components to containers. Selected containers may be annotated as public (i.e., they are external components that are not under the control of the system) or gateways. Annotating containers as public is relevant for security, especially for identifying potential attack paths.

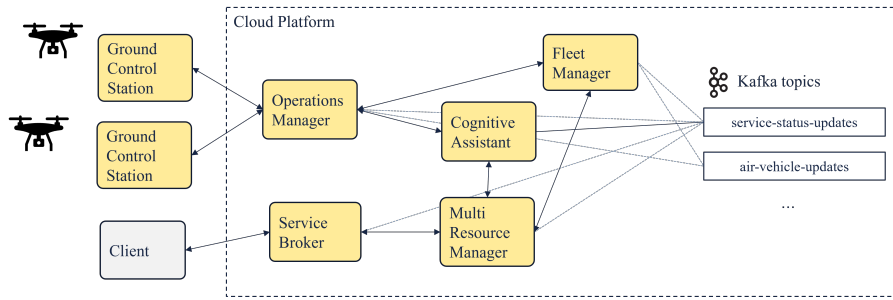


Figure 2: High-level architecture of running example.

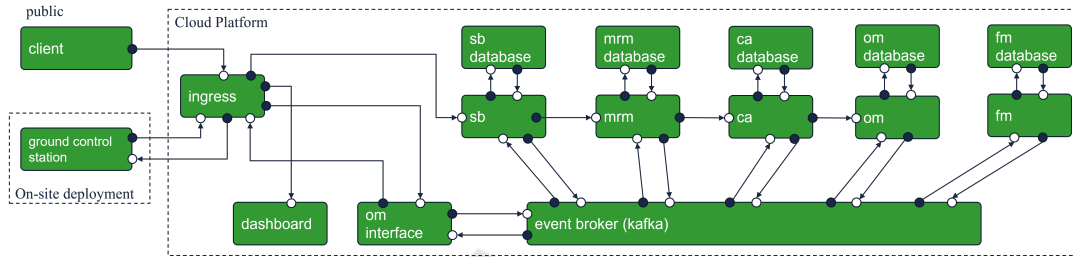


Figure 3: Container architecture of running example.

**Security Artifacts.** This input artifact consists of artifacts that may be identified in a Threat Analysis and Risk Assessment (TARA) analysis. Examples of security artifacts identified in a TARA analysis are assets and threat condition. SECLLOUD expects that at least assets are provided as input. Assets are objects (e.g., software elements) that need protection. SECLLOUD expects that an asset is associated with a container.

Both the cloud architecture and security artifacts are specified in the DSL of SECLLOUD. This specification enables SECLLOUD to reason about the security of the cloud architecture through reasoning principles. The reasoning principles enables the automated computation of threat scenarios and attack paths. SECLLOUD attempts to deploy security measures wherever they are applicable to mitigate threat scenarios. SECLLOUD outputs assumptions that need to be valid to ensure that the recommended security measure works as intended. The assumptions are turned into requirements that are output together with the recommended measures. These requirements shall be implemented and validated during the development of the system. In summary, SECLLOUD provides as output artifacts **threat scenarios** and **attack paths**, as well as **security measures alongside assumptions/requirements**.

### 4.1 Clingo

Clingo is an engine to implement logic programs based on Answer-Set Programming (ASP) semantics (Gelfond and Lifschitz, 1990).

A logic program is a set of rules. Each rule is of

the form  $a_0 \leftarrow a_1, \dots, a_n$ , where the literal  $a_0$  is the head of the rule, and the literals  $a_1, \dots, a_n$  are the body of the rule. A literal is an atom ( $a_m$ ) or a negated atom ( $\neg a_m$ ). A rule with an empty head is a constraint. A model (often referred by this article as solution) is an interpretation satisfying a set of rules.

This article uses the clingo notation, where  $:-$  denotes  $\leftarrow$ , and  $\neg$  denotes  $\neg$ . Identifiers beginning with capital letters (e.g., A, B) denote variables that during clingo’s execution are instantiated by appropriate terms. Identifiers beginning with a lower-case letter (e.g., a, b) are constants. The  $_$  (underscore) character specifies that the argument can be ignored in the current rule.

## 5 DOMAIN-SPECIFIC LANGUAGE

SECLLOUD provides a domain-specific language (DSL) for specifying (a) cloud architectures, (b) security artifacts, and (c) security measures. Table 1 provides the predicates/facts for specifying selected architectural elements and security artifacts. With the exception of threat conditions, threat scenarios and attack paths, all these facts shall be provided as input. The threat conditions denote the adverse consequences if the cybersecurity property of an asset is violated. Following the STRIDE methodology (Shostack, 2014), this article considers the authenticity, integrity, and authorization properties. These properties may be violated by, respectively, spoofing, tampering, and elevation of privilege attacks.

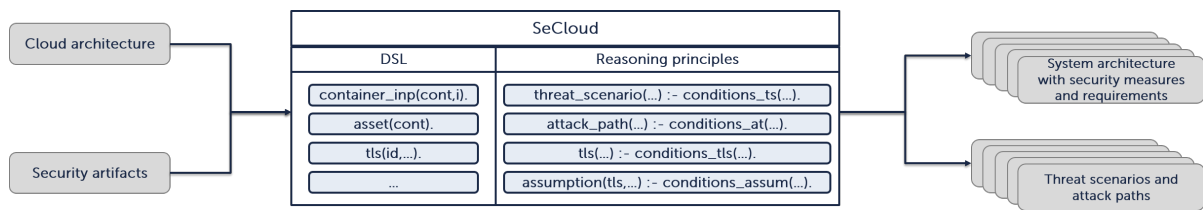


Figure 4: SECLoud’s workflow. Gray boxes are artifacts either received as input or generated for output. The light blue boxes under SECLoud are for illustrative purposes only.

**Example.** Consider the container architecture illustrated in Figure 3 and the predicates described in Table 1. Assume that both ground control station and sb have been identified as assets in the initial threat analysis. The user of SECLoud specifies the container architecture and the identified assets as follows.

```

% Containers (excerpt)
container_out(client,o1).
container_out(gcs,o2).
container_inp(gcs,i1).
container_inp(ingress,i2).
container_inp(ingress,i3).
container_onp(ingress,o3).
container_onp(ingress,o4).
container_inp(sb,i5).
container_out(sb,o6).

% Connections (excerpt)
conn(sg1,o1,client,i2,ingress).
conn(sg2,o2,gcs,i3,ingress).
conn(sg3,o3,ingress,i1,gcs).
conn(sg3,o4,ingress,i5,sb).
conn(sg3,o6,sb,i6,sb_database).

% Assets (excerpt)
asset(gcs).
asset(sb).
    
```

Listing 1: Specification of container architecture and assets.

### 5.1 Specifying the Security Content of Cloud Technologies

We refer to security measures at the architectural level as security architecture patterns (Cheng et al., 2019). Security architecture patterns are architectural solutions for mitigating threat scenarios. This section describes by example how SECLoud provides semantically-rich description of patterns that will enable the automated reasoning described in Section 6.

Before introducing the security architecture patterns, we describe the DSL of SECLoud for specifying security architecture patterns. The bottom of Table 1 describes the predicates/facts for specifying (a) the pattern instantiation, (b) the pattern attributes.

The former denotes all relevant architecture elements for instantiating the pattern. The relevant architecture elements are the pattern components, and the pattern channels. The pattern attributes denote the intent and problem addressed by the pattern. That is, the pattern attributes consist of the desired cybersecurity property achieved by the pattern, the threat addressed by the pattern, and the attack surface suitable to instantiate the pattern. You may read this as the pattern mitigates the particular threats (e.g., tampering) at the attack surface against the cybersecurity property (e.g., integrity).

We consider two kinds of attack surfaces, namely external and internal interface. The former denotes any attacks carried out by with external entities, such as entities that are public or entities that send data to the system through a gateway. The latter denotes any attacks carried out within the system. Considering the attack surface is relevant because some patterns may only be applied inside the system (internal interface) or outside the system (external interface). The traditional firewall is an example of such patterns that may only be applied at the external (a.k.a. network) interface.

At present, SECLoud formulates five security architecture patterns for the recommendation of different kinds of technologies for cloud applications: *Cilium*<sup>3</sup> is a Kubernetes network plugin with advanced functionality for providing, securing, and observing network connectivity between containers. Cilium uses IPsec to transparently encrypt data in transit between applications to avoid data tampering attacks. Cilium is also able to enforce policies for satisfying both authenticity and authorization properties. *TLS* (Transport Layer Security) is a protocol to provide secure communication over a network communication channel. In the context of cloud computing, TLS can be used to enforce secure communication between containers to prevent attackers to tamper with the data exchanged between applications. Mutual Authentication is described in Section 5.1.1. *OpenID Connect*<sup>4</sup> is an authentication protocol built on top of OAuth 2.0. OpenID Connect allows a client to authenticate itself when accessing services. An OpenID Connect

<sup>3</sup><https://cilium.io>

<sup>4</sup><https://openid.net/connect>



Table 1: DSL for (selected) architecture elements, security artifacts, and security architecture patterns.

| Fact   | Description   |
|--|---|
| <b>Architectural elements</b>  |   |
| container_inp(id,id <sub>i</sub> )   | id is a container and it has an input port id <sub>i</sub> .  |
| container_out(id,id <sub>o</sub> )   | id is a container and it has an output port id <sub>o</sub> .   |
| conn<br>(id,id <sub>1o</sub> ,id <sub>1</sub> ,id <sub>2i</sub> ,id <sub>2</sub> ) | id is a signal connecting an output port id <sub>1o</sub> of container id <sub>1</sub> to an input port id <sub>2i</sub> of container id <sub>2</sub> .   |
| gateway(id)  | id is a gateway container.  |
| public(id)   | id is a public container that may be accessible by external users.  |
| <b>Security artifacts</b>  |   |
| asset(id)  | denotes that container id is an asset.  |
| threat_condition<br>(id,secp,id <sub>ast</sub> )                                   | id is the adverse consequence if the cybersecurity property secp of asset id <sub>ast</sub> is violated, where secp $\in$ {authenticity, integrity, authorization}.   |
| threat_scenario<br>(id,ts <sub>des</sub> ,as,ts,id <sub>ast</sub> )                | id is a threat scenario, with description ts <sub>des</sub> , originating from attack surface as, to violate the cybersecurity property of asset id <sub>ast</sub> with threat ts, where ts $\in$ {spoofing, tampering, elevation_of_privilege}.  |
| attack_path<br>(id,el,id <sub>ast</sub> )  | id is an attack path denoting that malicious data may be injected from element el to target asset id <sub>ast</sub> .   |
| <b>Security architecture patterns</b>  |   |
| security_pattern<br>(id,pat,cp,inp,int,out)  | id is the unique identification of security architecture pattern pat. This pattern consists of a list of components cp. The last three parameters inp, int and out denote, respectively, the input, the internal, and the output channels related to the pattern.   |
| security_attributes<br>(pat,as,secp,ts)  | pat is a security architecture pattern suitable for satisfying the cybersecurity property secp by mitigating threat ts at the attack surface as.<br>as $\in$ {external_interface, internal_interface},<br>secp $\in$ {authenticity, integrity, authorization},<br>ts $\in$ {spoofing, tampering, elevation_of_privilege}. |

server verifies user credentials (e.g., username and password) and issues identity tokens. Client may use such identity token to authenticate themselves when accessing application services. *Service Mesh Istio*<sup>5</sup> is an application-level infrastructure for managing services in a cloud environment. Security-wise, Istio can enforce secure communication between applications by encrypting traffic and providing mutual authentication (i.e., to ensure integrity and authenticity). It provides support functions, such as key distribution and rotation. Istio provides the functionality to enforce policies to authorize resource access in the mesh.

We describe how SE-CLOUD captures cloud technologies in the form of semantically-rich description of security architecture patterns by example using Mutual Authentication. Table 2 describes the pattern attributes for all the patterns supported by SE-CLOUD.

<sup>5</sup><https://istio.io>

### 5.1.1 Mutual Authentication

Mutual Authentication (mTLS) (Vasudev et al., 2020) is a security measure for verifying the authenticity of two entities that wish to exchange data over a communication channel. Assume a client and a server connected through a communication channel. This pattern ensures that the server authenticates with the client, and the client authenticates with the server before the actual communication occurs. To ensure the authenticity of these entities over a communication channel, both the client and the server shall provide their digital certificates to prove their identities to each other. This is in contrast to TLS, where only the server presents a certificate. The Mutual Authentication pattern additionally guarantees the integrity of data exchanged between the client and server given that the identities of the entities have been correctly verified. Finally, for satisfying the authorization property, this pattern assumes that both client and server implement policies

Table 2: Security architecture patterns currently supported by SECLLOUD. This table describes that a pattern may be used to mitigate threat (e.g., spoofing) at the attack surface (e.g., external interface) to satisfy the security property (e.g., authenticity).

| Pattern            | Threat → Security Property   | Attack Surface              |
|--------------------|--|-----------------------------|
| Cilium             | spoofing → authenticity<br>tampering → integrity<br>elevation of privilege → authorization | internal interface          |
| OpenID Connect     | spoofing → authenticity  | internal/external interface |
| TLS                | tampering → integrity  | external interface          |
| mTLS               | spoofing → authenticity<br>tampering → integrity<br>elevation of privilege → authorization | internal/external interface |
| Service Mesh Istio | spoofing → authenticity<br>tampering → integrity<br>elevation of privilege → authorization | internal/external interface |

for data access control. Mutual Authentication may be applied for components communicating over an internal or external interface. The instantiation of the Mutual Authentication pattern is shown in Table 3.

Based on the structure of mTLS shown in Table 3, SECLLOUD instantiates this pattern as shown below.

```
security_pattern(id, mTLS, (cp1, cp2),
                (inp1, inp2), (int1, int2), none) .
```

Listing 2: Pattern instantiation (mTLS).

```
security_attributes(mTLS,
                  attack_surface(external_interface,
                                internal_interface),
                  property(authenticity, integrity,
                           elevation_of_privilege),
                  threat(spoofing, tampering,
                        elevation_of_privilege)) .
```

Listing 3: Security attributes (mTLS).

## 6 SECURITY REASONING PRINCIPLES

SECLLOUD specifies reasoning principles to reason about the security of the cloud architecture, including reasoning principles to (a) compute threat scenarios, (b) enumerate attack paths for the identified threat scenarios, (c) recommend security architecture patterns for mitigating threat scenarios. While the main focus of this section is about the pattern recommendation, we also provide a brief explanation of (a) and (b).

**Computing Threat Scenarios.** SECLLOUD computes threat scenarios that may violate the cybersecurity property of assets. We consider the authentic-

ity, integrity, authorization of assets received as input. Based on STRIDE (Shostack, 2014), we consider that these three properties may be violated, respectively, by spoofing, tampering, and elevation of privilege attacks. For example, for each asset received as input, we consider that the integrity of the asset may be violated by tampering attacks.

While this is a fairly coarse way of computing threat scenarios, SECLLOUD provides the possibility to define more fine-grained criteria for threat scenarios that may use all available information. This can be done by defining inference rules of the following form.

```
threat_scenario(ID, THREAT, SOURCE,
               TYPE, ASSET) :- ...
```

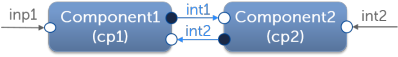
Listing 4: Threat scenario inference rule (snippet).

**Computing Attack Paths.** SECLLOUD computes attack paths for each threat scenario. To this end, SECLLOUD implements the intruder model described in Section 3 with capabilities to carry out both external and internal attacks.

**Example.** Figure 5 illustrates a potential attack path (based on the architecture from Section 2) for three threat scenarios. The upper part of Figure 5 illustrates an attack path from client (public interface) targeting the asset sb (short for service broker). The lower part Figure 5 describes three threat scenarios for asset sb. The attack surface is obtained from the attack path originated at the external interface (i.e., by client).

We now introduce three predicates for specifying whether a threat scenario is mitigated or not. First, mitigated\_by(IDTS, IDPAT) specifies when a threat scenario IDTS is mitigated by a suitable security architecture

Table 3: Instantiation of the Mutual Authentication pattern. The assumptions represent only a selection.

|                                       | Description   | SE-CLOUD Specification  |
|---------------------------------------|---|---|
| Pattern name                          | Mutual Authentication   | NAME=mTLS;  |
| Structure                             |    | COMPONENTS=[cp1,cp2];<br>INPUT_CH=[inp1,inp2];<br>INTERNAL_CH=[int1,int2];  |
| Intent                                | This pattern is used when two entities over a communication channel verify each other’s identity (authentication). Given the correct verification of the entities, this pattern also satisfies the integrity of messages exchanged between the sender and the receiver. Both entities implement authorization policies. | TYPE_SEC_PROPERTY=[authenticity,integrity,authorization];<br>TYPE_THREAT=[spoofing,tampering,elevation_of_privilege];<br>TYPE_ATTACK_SURFACE=[external_interface,internal_interface]; |
| Problem addressed                     | This pattern prevents that data exchanged between two entities in communication channel are spoofed or tampered. This pattern prevents such entities to gain privileges in accessing resources that shall not be authorized.  |   |
| Assumptions/ Requirements (selection) | CP1 and CP2 have digital certificate signed by a trusted CA.  | TYPE_ASSUMPTION=entity_has_received_certificate_signed_by_trusted_ca;<br>COMPONENTS=[cp1,cp2];  |

Example of an attack path



Threat scenarios

| Attack Surface     | Threat                 | Asset |
|--------------------|------------------------|-------|
| external interface | spoofing               | sb    |
| external interface | integrity              | sb    |
| external interface | elevation of privilege | sb    |

Figure 5: Illustration of one attack path and three threat scenarios computed by SE-CLOUD.

pattern IDPAT. We assume that a threat scenario is mitigated if the attack path leading to the threat scenario is addressed. Second, mitigated(IDTS) expresses that the threat scenario IDTS by some pattern. Finally, nmitigated(IDTS) specifies that IDTS is not mitigated. The reasoning rules for three predicates are as follows:

```
mitigated(IDTS) :- mitigated_by(IDTS, IDPAT) .
nmitigated(IDTS) :-
    threat_scenario(IDTS,_,_,_,_),
    not mitigated(IDTS) .
```

Listing 5: Mitigation rules.

### 6.1 Pattern Instantiation

SE-CLOUD specifies reasoning rules for automating the recommendation of security architecture patterns. These rules specify the conditions for when a particular

security architecture pattern can be recommended to mitigate threat scenarios targeting assets, which were identified by the user. Whenever a security architecture pattern is recommended, the rule mitigated\_by applies to infer which threat scenarios have been mitigated. SE-CLOUD only outputs architecture solutions when all threat scenarios have been mitigated. This is ensured through the specification of the following constraint, which expresses that any non-mitigated threat scenario is not allowed (implies empty/false).

```
:- nmitigated(IDTS) .
```

Listing 6: Constraint: No non-mitigated threat scenarios.

A security architecture pattern is recommended if there is a match between security artifacts and the pattern attributes. That is, SE-CLOUD considers information related to a threat scenario, i.e., attack surface, threat, and property violated by the threat, and the pattern attributes. A security architecture pattern is recommended if the following conditions hold:

- attack\_surface ∈ pattern.attribute
- threat ∈ pattern.attribute
- security\_property ∈ pattern.attribute

We illustrate a reasoning rule specified in SE-CLOUD to recommend security architecture patterns. The following reasoning rule specifies the conditions for recommending security architecture patterns. The rule checks whether there is one instance of security\_attributes for the security artifacts specified on the right side of the rule.

```

{ recommended_pattern(PAT, IDTS, IDAP, EL1, EL2)
  : security_attributes(PAT, AS, SECP, TS) } = 1
:- threat_scenario(IDTS, _, AS, TS, AST),
   threat_condition(_, SECP, AST),
   get_ap_from_ts(IDTS, IDAP),
   attack_path(IDAP, EL1, EL2).

```

Listing 7: Pattern recommendation rule.

The above rule derives facts of the form `recommended_pattern(PAT, IDTS, IDAP, EL1, EL2)`, which express that pattern `PAT` has been recommended to address attack path `IDAP` of threat scenario `IDTS`. `EL1` and `EL2` are architectural elements in the attack path `IDAP`. These architecture elements may become pattern components. For example, assume that the mTLS pattern has been recommended. The instantiation denotes that mTLS used for the communication from `EL1` (i.e., client) to `EL2` (i.e., server). SECLLOUD may derive the input, internal and output channels from the (baseline) system architecture received as input. Omitted here, we have a rule for mapping `recommended_pattern` to `security_pattern` (described in Table 1).

## 6.2 Combinations and Constraints

Pattern instantiation will produce a large number of possible solutions. The number of solutions depends on the number of (a) assets, (b) attack paths leading to threat scenarios, and (c) security architecture patterns. SECLLOUD computes all possible solutions with security architecture patterns as long as there exists a match between security artifacts and pattern attributes.

The purpose of SECLLOUD is to assist the selection of architecture options, so it is essential to select reasonable options from the set of possible instantiations. By building on an ASP solver like `clingo`, we can define sophisticated constraints on the possible combinations of patterns, which can be checked efficiently by the solver. Indeed, the intended usage of ASP solvers is to generate a (possibly very large) set of potential solutions up front and to select suitable ones using constraints (Lifschitz, 2019).

SECLLOUD uses the following constraints.

1. **All threats have been mitigated.** SECLLOUD specifies a constraint to only compute architecture solutions where all threat scenarios have been mitigated. That is, SECLLOUD discards solutions if at least one threat scenario has not been mitigated. This constraint is shown in Listing 6.
2. **Only one pattern for addressing each threat scenario.** SECLLOUD may output two security architecture patterns for addressing the same threat scenario. For example, possible solutions for mitigating tampering threats violating the integrity

of assets are Mutual Authentication and Service Mesh patterns. SECLLOUD specifies a constraint to only recommend one pattern for each threat scenario avoiding that two patterns are recommended for the same threat scenario.

3. **No TLS and mTLS for the same element.** SECLLOUD may compute solutions where TLS and mTLS are recommended for the same element. These solutions may be redundant in the sense that the element should have two certificates, one for TLS and one for mTLS. SECLLOUD specifies a constraint to avoid TLS and mTLS from being recommended for the same element.
4. **No mTLS for public elements.** Mutual Authentication may be impractical for public elements, as certificates would need to be distributed to all external clients. SECLLOUD therefore specifies a constraint to avoid the mTLS pattern from being recommended for public elements.
5. **Only one pattern for addressing equivalent security artifacts.** SECLLOUD may compute several combinations of patterns for addressing threat scenarios. This might lead to expensive solutions to be implemented during the development of the system. For example, SECLLOUD may output a solution where Service Mesh is recommended for two containers, and Cilium is recommended for the remaining containers in the system. It might be expensive and unnecessary to deploy a Service Mesh for only two containers of the system, especially when Cilium may be deployed for all containers. SECLLOUD specifies constraints to recommend the same pattern to address security artifacts with equivalent attributes (i.e., attack surface, threat, and cybersecurity property).

## 7 CASE STUDY

This section illustrates the use of SECLLOUD for the cloud architecture described in Section 2. We evaluate the use of the constraints defined in Section 6.2, and discuss the recommended patterns by SECLLOUD.

For the sake of illustration, we assume two assets: `ground control station` and `sb`. As described in Section 2, we assume that `client` is a public component, `ingress` is a gateway, and `ground control station` sends and receives data through `ingress`. This means that all attack paths involving `client` and `ground control station` denote potential attacks through an external interface. The remaining attack paths denote potential attacks through an internal interface.

Table 4: Experiments with constraints. Scenario with 2 assets, 31 attack paths, and 5 security architecture patterns. The entry ‘-’ means that SECLLOUD has not returned all solutions after 60 minutes. The constraint’s ID (e.g., 1, 2,...) refers to the constraints described in Section 6.2.

| Constraint    | #Solutions | Execution time (s) |
|---------------|------------|--------------------|
| none          | -          | 3600               |
| 1             | -          | 3600               |
| 1, 2          | -          | 3600               |
| 1, 2, 3       | 2916       | 0.27               |
| 1, 2, 3, 4    | 1458       | 0.13               |
| 1, 2, 3, 4, 5 | 6          | 0.06               |

SECLLOUD has computed 93 threat scenarios, and 31 attack paths (28 and 3 attack paths based on, respectively, the external and internal intruder). For each attack path, SECLLOUD considers three threat scenarios that may violate the property of the asset through spoofing, tampering, and elevation of privilege. Thus, the number of threat scenario is  $31 \times 3 = 93$ .

There are many ways of applying the security architecture patterns to mitigate these threat scenarios. With the patterns and constraints defined in Sections 5 and 6, SECLLOUD recommends six architecture options almost instantly. These options are shown in Table 5.

To understand how SECLLOUD may deal with a larger number of possible solutions, e.g., when extended with more patterns, it may be instructive to consider also its performance when some of the constraints from Section 6.2 are lifted. Table 4 describes the results of our experiments. We ran the experiments on a 1.9GHz Intel Core i7-8665U with 16GB RAM wuth Ubuntu 18.04 LTS and clingo 5.2.2.

SECLLOUD could not enumerate all solutions without any constraints and with constraints 1, and 1 & 2. With constraints 1 & 2 & 3, and 1 & 2 & 3 & 4, SECLLOUD computed all solutions within 0.27 and 0.13 seconds, respectively. These results illustrate that while the number of pattern instantiations being considered is the same in all cases, imposing constraints to filter solutions is sufficient to improve performance. However, the number of solutions were still high, impacting the usability of SECLLOUD (i.e., it is impractical for the user to choose a solution). The breakthrough was the specification of constraint 5 for only recommending one type of pattern to address equivalent security artifacts. Together with the other constraints, SECLLOUD computed six solutions within 0.06 seconds.

The six solutions are shown in Table 5. In general, they represent a reasonable selection of architecture options to address the 93 threat scenarios identified by

SECLLOUD. Solution 6 has previously been selected manually in the development of the system that serves as our running example. One aspect that is perhaps not reasonable is that all solutions use the authorization policies of the Istio service mesh to verify OpenID Connect tokens, even when no components are placed in the mesh. This is an artifact of the current small selection of security architecture patterns. However, the reasons for selecting the patterns are documented by SECLLOUD, so users may replace individual patterns in each solution.

Indeed, SECLLOUD provides documentation for the identified attack paths, threat scenarios, and new requirements for each solution. The requirements are traced to threat scenarios, which themselves are associated to attack paths. For each threat scenario, SECLLOUD documents which security architecture pattern in the selected solution has mitigated the threat.

To aid the user in selecting a solution, SECLLOUD computes simple metrics of the solution. We consider the number of new requirements on application components and on the infrastructure deployment as one selection criterion. We distinguish such requirements in the following sense:

- *Application requirements* need to be considered in the development of the applications themselves. For example, Solution 1 adds the new application requirement that sb uses mTLS to communicate with sb database. This needs to be considered by the developers, e.g., by using a suitable library.
- *Infrastructure requirements* need to be implemented when building the infrastructure and deploying the components. For example, Solution 6 adds a requirement that an authorization policy for the communication from sb to mrm has been defined. The configuration of the Istio service mesh is also defined in the form of requirements. For example, Solution 6 has a requirement that sb must be part of the service mesh.

If the aim is to provide security measures in a transparent manner for the application developers, then Solution 6 will likely be preferable over Solution 1. This choice is indeed the case as Solution 6 makes encryption transparent, while Solution 1 requires application developers to use mTLS explicitly.

## 8 RELATED WORK

SECLLOUD is built on SecPat proposed by (Dantas and Nigam, 2022). SecPat enables the recommendation of security architecture patterns for autonomous vehicle architectures, while SECLLOUD supports the

Table 5: Solutions recommended by SECLLOUD.

| Solution | Recommended Security Patterns   | # Additional Application Requirements | # Additional Infrastructure Requirements |
|----------|---|---------------------------------------|--|
| 1        | <i>Mutual Authentication</i> for mitigating the threat scenarios with attack paths based on the internal intruder, including attack paths targeting ground control station. TLS for ingress acting as a server for client to mitigate tampering attacks. <i>OpenID Connect</i> for client for authentication purposes to address spoofing attacks. The identity tokens of client obtained by OpenID Connect can be checked by an instance of Service Mesh Istio for all containers receiving data from client.  | 19                                    | 34                                       |
| 2        | <i>Cilium</i> for mitigating selected threat scenarios with attack paths based on the internal intruder, and <i>Mutual Authentication</i> for addressing the attack paths targeting ground control station. In this solution, <i>Mutual Authentication</i> has been recommended for addressing attacks from outside the system (external interface). TLS for ingress acting as a server for client to mitigate tampering attacks. <i>OpenID Connect</i> for client for authentication purposes to address spoofing attacks. The identity tokens of client obtained by OpenID Connect can be checked by an instance of Service Mesh Istio for all containers receiving data from client. | 11                                    | 22                                       |
| 3        | <i>Cilium</i> for mitigating selected threat scenarios with attack paths based on the internal intruder. TLS for ingress acting as a server for client and ground control station to mitigate tampering attacks. <i>OpenID Connect</i> for both client and ground control station for authentication purposes to address spoofing attacks. The identity tokens of client and ground control station obtained by OpenID Connect can be checked by an instance of Service Mesh Istio for all containers receiving data from client and ground control station.  | 11                                    | 25                                       |
| 4        | <i>Mutual Authentication</i> for mitigating selected threat scenarios with attack paths based on the internal intruder. TLS for ingress acting as a server for client and ground control station to mitigate tampering attacks. <i>OpenID Connect</i> for both client and ground control station for authentication purposes to address spoofing attacks. The identity tokens of client and ground control station obtained by OpenID Connect can be checked by an instance of Service Mesh Istio for all containers receiving data from client and ground control station.   | 19                                    | 37                                       |
| 5        | <i>Service Mesh Istio</i> for mitigating selected threat scenarios with attack paths based on the internal intruder. TLS for ingress acting as a server for client and ground control station to mitigate tampering attacks. <i>OpenID Connect</i> for both client and ground control station for authentication purposes to address spoofing attacks. The identity tokens of client and ground control station obtained by OpenID Connect can be checked by the instance of Service Mesh Istio.  | 11                                    | 40                                       |
| 6        | <i>Service Mesh Istio</i> for mitigating selected threat scenarios with attack paths based on the internal intruder, and <i>Mutual Authentication</i> for addressing the attack paths targeting ground control station. TLS for ingress acting as a server for client to mitigate tampering attacks. <i>OpenID Connect</i> for client for authentication purposes to address spoofing attacks. The identity tokens of client obtained by OpenID Connect can be checked by the instance of Service Mesh Istio.   | 11                                    | 37                                       |

recommendation of patterns suitable for cloud architectures. SecPat provides a DSL and reasoning principles to automate the recommendation of patterns. SecPat only considers one condition to recommend patterns, namely the cybersecurity property satisfied by the pattern. SECLLOUD extends SecPat's DSL to include further pattern's attributes, namely the threat mitigated by the pattern, and the attack surface that the pattern may be deployed. SECLLOUD follows the STRIDE methodology to map threats to cybersecurity properties. This extension enables a more precise recommendation of patterns through reasoning principle rules. Another extension is the specification and evaluation of constraints to reduce the number of solutions with patterns. This extension deals with scalability issues and improves the usability of SECLLOUD.

ThreatGet<sup>6</sup> is a commercial tool for threat analysis. ThreatGet identifies threat scenarios and attack paths in an automated fashion. To deal with such security artifacts, ThreatGet provides a list of potential security measures to be selected by the user. ThreatGet does not instantiate the selected security measures in the system architecture. As a result, it might be unclear for the user to identify which components are relevant to the selected security measures. SECLLOUD instantiates the recommended security measures by making explicit which components are part of the security measure (e.g., mTLS for components A and B).

Another commercial tool for STRIDE analysis of cloud architectures is Microsoft's Threat Analysis tool<sup>7</sup>. While SECLLOUD is also based on STRIDE, it is able to automatically compute architecture options. Its flexible definition using ASP allows the extension to more fine-grained threat scenario models.

SECLLOUD outputs requirements alongside the recommended security measures. Other tools, such as Ansible (Spanaki and Sklavos, 2018), may be used to harden cloud infrastructures by implementing such requirements. Ansible provides the means of, e.g., installing SSL certificates, installing and configuring monitoring tools, and configuring user accounts.

## 9 CONCLUSION

This article proposed SECLLOUD, a tool to assist security engineers with the selection of security measures for cloud architectures. We validated SECLLOUD in a case study that provides cloud services for unmanned air vehicles (UAVs). We are currently investigating several future directions, including (i) specification

of security measures to address threat scenarios that violate the availability of assets, and (ii) integration of SECLLOUD in a model-based system engineering tool that will serve as a frontend to improve its usability.

## ACKNOWLEDGMENTS

We thank the German Ministry for Economic Affairs and Climate Action of Germany for funding this work through the LuFo V-3 project RTAPHM.

## REFERENCES

- Baral, C. (2010). *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.
- Carroll, M., Kotzé, P., and van der Merwe, A. (2011). Secure cloud computing: Benefits, risks and controls. In Venter, H. S., Coetzee, M., and Looock, M., editors, *Information Security South Africa Conference 2011, ISSA 2011*. ISSA, Pretoria, South Africa.
- Cheng, B. H. C., Doherty, B., Polanco, N., and Pasco, M. (2019). Security Patterns for Automotive Systems. In *MODELS'19*.
- Dantas, Y. G. and Nigam, V. (2022). Automating Safety and Security Co-Design through Semantically-Rich Architectural Patterns. *ACM Trans. Cyber Phys. Syst.*
- Dolev, D. and Yao, A. C. (1983). On the security of public key protocols. *IEEE Trans. Inf. Theory*, 29(2):198–207.
- Eliseev, V., Miliukova, E., and Kolpinskiy, S. (2021). Neural Network Cryptographic Obfuscation for Trusted Cloud Computing. In *Integrated Models and Soft Computing in Artificial Intelligence*, pages 201–207.
- Gelfond, M. and Lifschitz, V. (1990). Logic programs with classical negation. In *ICLP*.
- Lifschitz, V. (2019). *Answer Set Programming*. Springer.
- Oleshchuk, V. A. and Kōien, G. M. (2011). Security and privacy in the cloud a long-term view. In *2011 2nd International Conference on Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology (Wireless VITAE)*.
- Potassco (2022). Clingo: A grounder and solver for logic programs <https://github.com/potassco/clingo>.
- SeCloud (2022). <https://github.com/ygdantas/SeCloud>.
- Shostack, A. (2014). *Threat Modeling: Designing for Security*. Wiley.
- Spanaki, P. and Sklavos, N. (2018). Cloud Computing: Security Issues and Establishing Virtual Cloud Environment via Vagrant to Secure Cloud Hosts. In *Computer and Network Security Essentials*, pages 539–553. Springer.
- Vasudev, H., Deshpande, V., Das, D., and Das, S. K. (2020). A Lightweight Mutual Authentication Protocol for V2V Communication in Internet of Vehicles. *IEEE Trans. Veh. Technol.*, 69(6):6709–6717.

<sup>6</sup><https://www.threatget.com>

<sup>7</sup><https://www.microsoft.com/en-us/securityengineering/sdl/threatmodeling>

# Chapter 8

## Integrating Security Knowledge in MBSE: A Plugin for Automated Synthesis

Chapter 8 focuses on the integration of the developed logic programming tool [43], designed for reasoning about security activities, into a model-based systems engineering (MBSE) framework. Within this chapter, a plugin aimed at automating security synthesis within the MBSE context is proposed. We exemplify the use of the developed plugin using an example taken from the unmanned air vehicle domain.

### Contributing article:

- Yuri Gil Dantas, Vivek Nigam, and Ulrich Schöpp: A Model-Based Systems Engineering Plugin for Cloud Security Architecture Design. *SN Computer Science* 2024: Volume 5, Article number: 553.

**Copyright information:** The article [44] is reproduced with permission from Springer Nature. License number: 5798741399960. Print rights of the Version of Record are provided for; electronic rights for use only on institutional repository as defined by the Sherpa guideline ([www.sherpa.ac.uk/romeo/](http://www.sherpa.ac.uk/romeo/)) and only up to what is required by the awarding institution.

**Author contributions:** The idea of developing a Model-Based Systems Engineering (MBSE) plugin for securing cloud architectures was brought up by Yuri Gil Dantas, Vivek Nigam and Ulrich Schöpp. They had several discussions on how the interfaces with the backend (previously developed by the same authors [43]) should be designed. With the help of Ulrich Schöpp, Yuri Gil Dantas implemented the MBSE plugin. Yuri Gil Dantas also had several discussions with MBSE researchers at fortiss. Yuri Gil Dantas and Ulrich Schöpp took the lead in writing the initial draft of the article, in particular Yuri Gil Dantas wrote the core sections (i.e., Section 3 and Section 4) of the article.





# A Model-Based Systems Engineering Plugin for Cloud Security Architecture Design

Yuri Gil Dantas<sup>1</sup> · Vivek Nigam<sup>2</sup> · Ulrich Schöpp<sup>1</sup>

Received: 23 July 2023 / Accepted: 26 February 2024  
© The Author(s), under exclusive licence to Springer Nature Singapore Pte Ltd. 2024

## Abstract

Security is one of the biggest concerns for cloud infrastructures. Cloud infrastructures are susceptible to a wide range of threats, including external and internal threats. Without proper security mechanisms, these threats may compromise the security properties of services hosted in the cloud. To secure cloud infrastructures against threats, it is crucial to perform a threat analysis in the early stages of the system development (i.e., during the design of the system architecture). Threat Analysis and Risk Assessment (TARA) is a well-known approach used by researchers and practitioners. TARA consists of several activities, including asset identification, threat scenarios, attack paths, and risk treatment decision. The risk treatment decision activity involves selecting appropriate security measures to mitigate the identified threat scenarios. In the current state of practice, TARA activities are performed manually by engineers, leading to time-consuming processes and potential errors. In our previous article, we proposed a logic programming tool to enable the automation of TARA activities, including the recommendation of cloud-based security measures. This article proposes Security Pattern Synthesis, a Model-Based Systems Engineering (MBSE) plugin for securing cloud architectures. Security Pattern Synthesis is implemented in Java while using the previously proposed logic-programming tool as a backend to reason about the security of the cloud architecture.

**Keywords** Securing cloud architectures · Model-based systems engineering · Security architecture patterns · Automation

## Introduction

Providing connected applications presents many challenges. It is not enough to implement just the application functions themselves. For productive operation, there are additional requirements for reliability, observability, manageability and security. Cloud technologies offer building blocks to address such requirements. They provide runtime environments with mechanisms to address reliability, scalability and resilience,

e.g., by load balancing, automatic scaling and redundant, and distributed deployment of components. They provide interfaces and tools for monitoring performance and operational details. They support the implementation of modern security architectures, such as zero trust architectures [1], e.g., by providing the interfaces and components to implement authentication and authorization architectures.

While cloud infrastructures provide many benefits to the application developer, their use presents significant challenges. Security is one of the biggest concerns. A literature review conducted by [2] confirms that security is the main risk for businesses using cloud infrastructures. One risk is that when using cloud infrastructures operated by third-party providers, the application data is no longer controlled by the client who purchased the cloud service.

Two attack surfaces need to be considered for cloud applications: First, external attackers may carry out attacks via external interfaces [3]. For instance, external attacker may carry out spoofing attacks to impersonate legitimate clients. Second, because application components are hosted on a cloud infrastructure that may be controlled by third parties, it is also essential to consider internal interfaces as

---

This article is part of the topical collection “Recent Trends on Information Systems Security and Privacy” guest edited by Steven Furnell and Paolo Mori.

---

✉ Yuri Gil Dantas  
dantasyg@gmail.com  
Vivek Nigam  
vivek.nigam@gmail.com  
Ulrich Schöpp  
schoepp@fortiss.org

<sup>1</sup> Fortiss GmbH, Munich, Germany

<sup>2</sup> Federal University of Paraíba, João Pessoa, Brazil

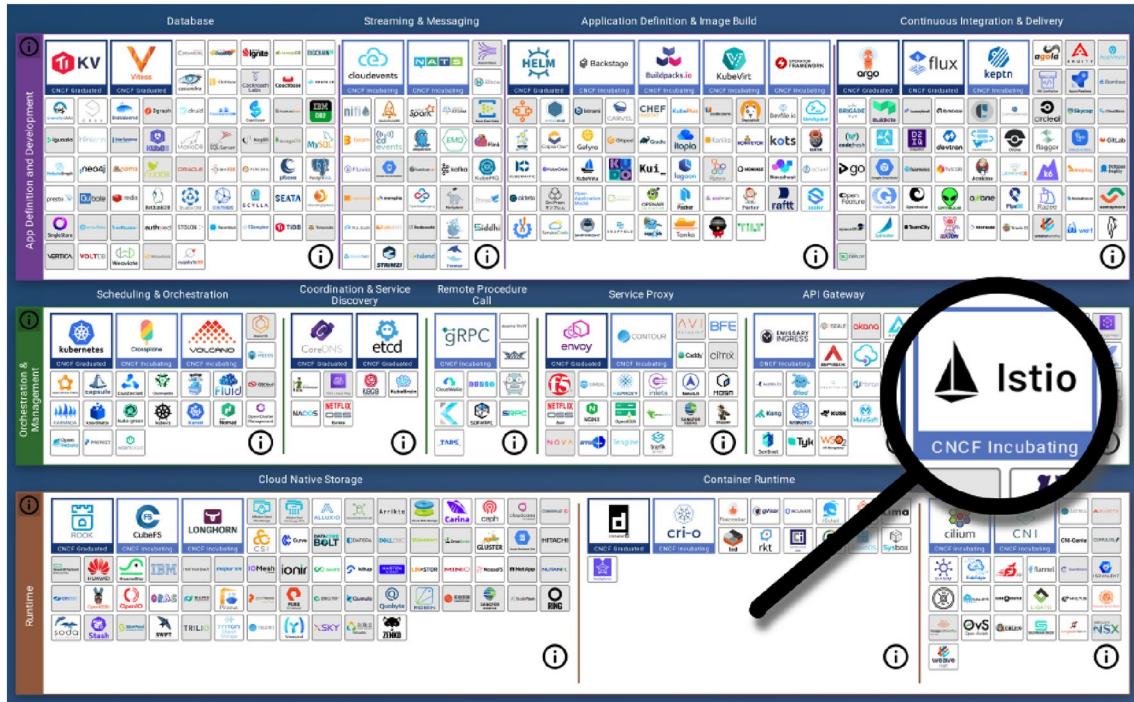


Fig. 1 Cloud native technology landscape (excerpt) [6]

possible attack surfaces. An internal interface may be the source of attacks, e.g., because of a misconfiguration of the network connection between data centers, allowing unauthorized access, or even intentionally malicious insiders [4]. To implement an effective defense-in-depth strategy, it is important to also consider threats originating from internal services. Potential attacks through these attack surfaces must be mitigated by selecting and deploying suitable security measures for cloud infrastructures.

To secure cloud applications against threats at these two attack surfaces, it is important to perform a threat analysis in the early stages of the system development, so that the application architecture can be designed to mitigate potential threats and to avoid expensive changes later in the cloud system life-cycle. A threat analysis includes of identifying assets, threat scenarios, and possible attack paths. On the basis of a threat analysis, one can then select security measures to mitigate the identified threat scenarios and reduce the risk of threats to an acceptable level. Threat Analysis and Risk Assessment (TARA) [5] is a well-know threat analysis approach used in several domains, such as automotive.

Identifying threat scenarios and selecting a suitable security architecture to mitigate them is a difficult challenge for cloud applications. Performing a comprehensive threat analysis and keeping it up-to-date during development is requires a lot of effort. In addition to having to consider both internal and external attack surfaces, the complexity

of cloud technologies and the number of possible design choices makes the evaluation and comparison of architecture options a difficult task.

There are a vast number cloud architectures, technologies, components, frameworks and platforms to choose from, and it is hardly feasible to evaluate the implications of all possible choices in detail, e.g., due to resource and time constraints. Even when the technologies are understood in principle, it is not easy to keep track of the consequences of selecting a combination of them. One would also like to understand the trade-offs between different choices regarding system development and operation, e.g., in the form of additional requirements for certificate management, or regarding resource overhead.

In an effort to give a systematic overview of the available cloud technologies, the Cloud Native Foundation has produced the cloud native landscape.<sup>1</sup> An excerpt of the landscape is shown in Fig. 1. Each of the tiles in this figure represents a component or framework that may be used in an application. It may require weeks to understand in detail the impacts of using each of these components for an application. The challenge of selecting a suitable combination of components may be summarized by the following quote:

The cloud native landscape is vast, and it's easy to become overwhelmed by its growing number of

<sup>1</sup> <https://landscape.cncf.io/>.

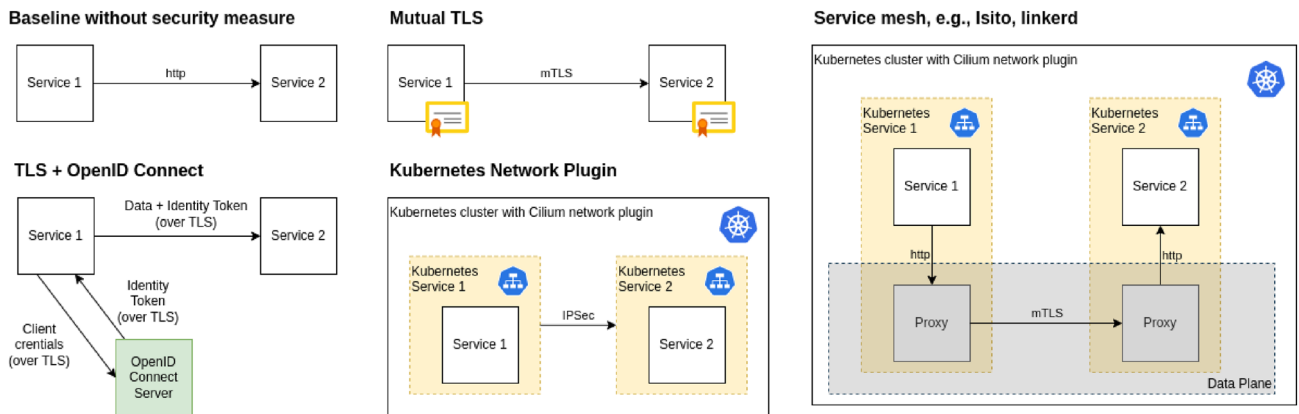
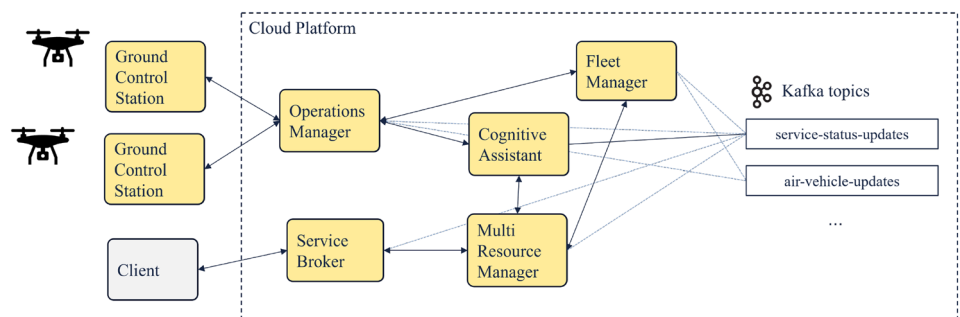


Fig. 2 Some options for securing service-to-service communication

Fig. 3 High-level architecture of running example



competing and overlapping platforms and technologies.<sup>2</sup>

To give a concrete example of possible choices to address a potential threat, consider the problem of securing internal service to service communication. Suppose our cloud application consists of several internal services that communicate with each other internally. Depending on the configuration of the cloud infrastructure, these services may be deployed in different data centers, so it is important to secure the communication between them. Services need to be authenticate themselves between each other to avoid spoofing attacks. The transmitted data must be protected against tampering. Figure 3 shows a few options for securing service-to-service communication against tampering and spoofing, and to achieve mutual authentication (Fig. 2).

- **Baseline:** The baseline is unsecured communication using http. While this communication is susceptible to spoofing and tampering attacks, this baseline option may still be suitable if a security perimeter can be established.

This may be the case if both services are hosted on a trusted private network or private infrastructure.

- **Mutual TLS:** A second option is to use the mTLS protocol for encrypting communication and to achieve mutual authentication. It is the responsibility of the application developers to secure the communication using mTLS, i.e., the use of mTLS will become an application requirement. It is important to take into account that this requires the application developers to have experience with secure coding practices. The incorrect use of cryptographic libraries has been the source of many security issues, see, e.g., [7]. Moreover, the use of mTLS requires both services to be issued with TLS certificates, which need to be renewed in operation when they expire or are revoked.
- **TLS with OpenID Connect:** A third option is to use TLS just for encryption and to use an authentication protocol like OpenID Connect [8]. This option can be implemented by integrating off-the-shelf components of this protocol, such as Keycloak, where the authentication servers need to be operated and be available for reliable operation. Application developers need to make sure that the services perform authentication.
- **Kubernetes network plugin:** A more transparent option is to build on the infrastructure to perform encryption and

<sup>2</sup> <https://www.cncf.io/blog/2020/09/15/top-7-challenges-to-becoming-cloud-native>.

mutual authentication between services. For example, the Kubernetes network plugin Cilium is able to encrypt communication and to authenticate the communication between services on the network level using the IPsec protocol. In this case, there are no additional application requirements for the development of the services. However, the application deployment to Kubernetes must be configured to use the Cilium plugin correctly.

- **Service mesh:** Another option for transparent securing of service-to-service communication is the use of a service mesh like Istio or linkerd. Here, the communication is secured on the application level rather than on the network level as in the previous option. The idea if a service mesh is to deploy additional proxy components, which implement the communication along a software defined service mesh. The application services communicate only locally with proxy components. The proxies are managed by the service mesh. It is their responsibility to securely relay the communication to the target service. They take care of tasks like encryption, authentication, and authorization. The service mesh is responsible for administrative tasks like certificate rotation.

This outlines the several options that require consideration when selecting a security architecture for cloud-based systems. As outlined, each of these options has a advantages, such as being transparent to the application, and disadvantages, such as importing a dependency to a complex technology component like Cilium or Istio. Moreover, the discussion addresses only the threats of tampering and spoofing attacks. To protect against elevation of privilege attacks, one would additionally like to authorize the actions of services. As one considers multiple threat scenarios and possible mitigations, it becomes a complex task to consider all possible options and to understand how they may be combined and what the trade-offs between different choices are.

## Goal and Contribution

In our previous article [6], we proposed **SeCloud**: a logic-based automated tool for the recommendation of security measures for cloud architectures. Despite its effectiveness in automating the recommendation of security measures, **SeCloud** operates solely through a command-line interface. Recognizing that the industry frequently employs a model-based systems engineering approach across various disciplines, including security, we aim to align with this approach. Model-based development is based on the idea of providing integrated tool support for the whole design and development process from requirement analysis and architecture modeling up to the final configuration of the completed system. This integrated approach is essential for

the design of the security architecture of cloud systems. The security architecture needs to be maintained, revisited and updated throughout the life-cycle of an application. As shown in this article, a model-based approach allows us to automate security activities in the application design phases.

*The goal of this article is to leverage a model-based approach for seamlessly integrating **SeCloud** into a model-based systems engineering tool.* This integration enhances the tool's usability and makes it more accessible to a broader user base. We propose **Security Pattern Synthesis**, a plugin for the model-based systems engineering tool **AutoFOCUS 3** [9]. The developed plugin supports the security analysis and selection of cloud architectures using a model-based approach. The core functionality of the proposed plugin is **SeCloud** [6]. **SeCloud** is implemented in the logic programming engine **clingo** [10], while **AutoFOCUS 3** is implemented in Java. We have integrated **SeCloud** into **AutoFOCUS 3** in the form of the **Security Pattern Synthesis** plugin. The main target user for **Security Pattern Synthesis** is security engineers responsible for assessing the security of cloud architectures and hardening such architectures with security measures.

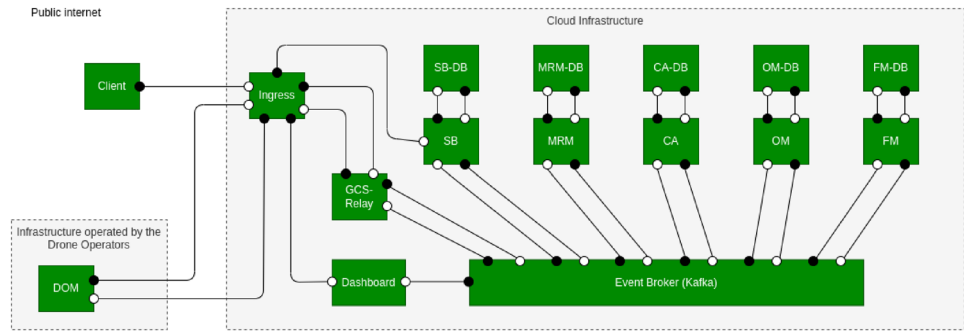
In summary, this article is an extended version of our article published at **ICISSP 2023** [6]. To ensure the self-contained nature of this article, we provide a detailed description of the reasoning core of **SeCloud**, which was omitted from the **ICISSP** article due to space constraints. Additionally, as the main contribution, we present the integration of **SeCloud** into a model-based systems engineering tool.

## Roadmap

The remainder of this article is structured as follows:

- Section “**Running Example**” provides the description of a real cloud application serving as a running example for explaining **Security Pattern Synthesis**.
- Section “**SeCloud**” provides a verbose description of **SeCloud**, including its workflow, domain-specific language, and reasoning core. This verbose description represents a minor contribution to this article, which had been omitted from our prior publication due to space constraints.
- Section “**Security Pattern Synthesis**” provides the description **Security Pattern Synthesis**, specifically focusing on how **SeCloud** integrates into a model-based systems engineering tool. This section represents the main contribution of the article.
- Sections “**Related Work**” and “**Conclusion**” conclude the article by discussing related and future work.

**Fig. 4** Task architecture of running example: The gray area specifies the item boundary



SeCloud is available online at [11], and interested readers have the option to request the binary of Security Pattern Synthesis when needed.

### Running Example

The Security Pattern Synthesis plugin assists with the design of a technical security architecture for cloud applications. In this section, we introduce a running example for the explanation and evaluation of both SeCloud and Security Pattern Synthesis.

The running example is a real cloud application developed in a research project on the digitalization and automation in avionics systems. The application coordinates unmanned air vehicles (UAVs) and provides services, such as transportation services or search-and-rescue services. It optimizes the usage of UAV and implements planning, optimization and prognostic health management functions. While the particular details of the use-case are not important for the presentation of our tools, the application provides a realistic use-case. Security is particularly important in this use-case, as attacker may gain control over UAVs, which may have severe implications for safety.

The role of Security Pattern Synthesis is to automate parts of the security analysis and to support the selection of cloud technologies at an early stage in the system development, where the main components and their interfaces have been identified, but where the security architecture of the system is still under consideration.

Following, a model-based approach, we model different aspects of the application at different abstraction levels.

### Logical Architecture

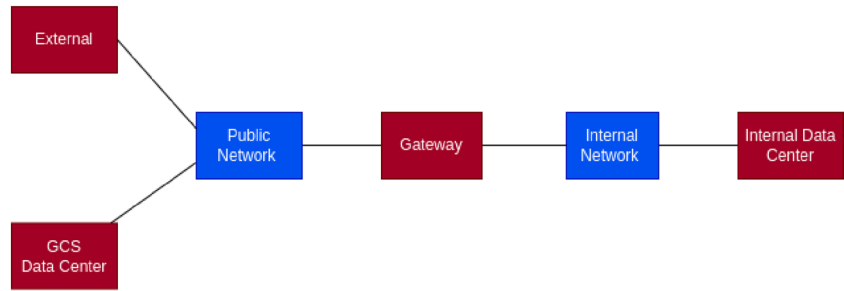
The logical architecture captures the hierarchical structure of the application into logical blocks of systems and their sub-systems. The logical architecture captures communication interfaces between these components.

Figure 3 gives an overview of the logical architecture of the components of the example application. They are the Service Broker (SB), Multi Resource Manager (MRM), Cognitive Assistant (CA), Operations Manager (OM), Fleet Manager (FM). Together, these components perform the provided services like goods transportation or search-and-rescue operations. The SB manages the requests by the user, MRM and CA plan and optimize the selection of assets, OM is responsible for orchestrating operative tasks, such as pre-flight preparation of UAVs and post-flight maintenance, and FM records and analyzes fleet data to enable predictive maintenance. These components are intended to be deployed on a cloud platform. They implement the overall service and provide public interfaces where clients can access the services. They communicate with with ground control stations for executing missions. Ground control stations are operated by Drone Operators with their own infrastructure.

The components communicate with each other through two mechanisms. First, components communicate directly with each other over bidirectional channels. On a technical level, these direct communication interfaces will be realized by Rest/HTTP APIs that allow direct access to resources. Second, the components use a service-based architecture for pub/sub communication. On a technical level, this will be implemented by Kafka.<sup>3</sup> Kafka provides named topics where the application components may publish messages about certain aspects, e.g. the topic *av-updates* is intended for messages pertaining to the status of air vehicles. These messages are delivered to all components that subscribe to them. For example, the Operations Manager receives telemetry data from the ground control station and publishes this to *av-updates*. The Fleet Manager subscribes to this topic and thus receives telemetry updates. Pub/sub communication is convenient for decoupling components, but also introduces challenges for security, e.g. for limiting the impact of the compromise of one internal component.

<sup>3</sup> <https://kafka.apache.org/>.

**Fig. 5** Platform architecture of running example



## Task Architecture

The task (a.k.a. software) architecture models the software components that implement the components from the logical architecture. In our case, the components of the task architecture are given by Docker components. The components of the logical architecture are implemented by containers.

Figure 4 shows the task architecture of the our example application system. It contains containers that implement the application components, as well as additional support containers. The logical components from Fig. 3 are subdivided into sub-component that, in this example, each map to exactly one container. For example, the Service Broker has two sub-components SB and SB-DB, the first of which implements the component's nominal functions, and the second of which provides a database to support the main component. On a technical level, both components are realized by containers, as shown in Fig. 4. In general, it is also possible to map multiple logical components on the same container.

In addition to containers implementing the components of the logical architecture, there are additional support containers, such an Ingress component that acts as a gateway for external communication from the cloud infrastructure that hosts the main components of the system. In particular, the topic-based communication of the service-oriented architecture from Fig. 3 is implemented by Kafka. The deployment of Kafka is now made explicit in the task architecture. While, in realistic deployments, Kafka will likely be deployed redundantly, it suffices to consider it as a single container for our purposes. A GCS-Relay container that provides an interface for ground control stations to and that relays messages from and to Kafka topics.

Figure 4 shows assumptions about the deployment and the security environment of the system. We consider client a public, external component, to which attackers have full access. Most of the components are intended to be hosted on a cloud platform, where ingress serves as a gateway component. While ground control station is part of the overall system, it needs to be hosted on-site at an airport rather than in the cloud. These assumptions are formalizes by a mapping

of the task architecture to a platform architecture, which we outline next.

## Platform Architecture

The final layer of the architecture considered in this work is the platform architecture. It captures assumptions about the hardware that the containers are deployed.

Figure 5 shows the coarse design of a platform architecture at an early design stage. The red boxes represent hardware units, such as data centers or individual machines. The blue boxes model network connections and assumptions about the network.

The components from the task architecture are allocated to hardware units in the platform architecture. In our example, the containers of the drone operator are allocated to the GCS Drone Center, the Client is allocated to a catch-all node for external systems, Ingress is allocated to the Gateway, and the remaining components of the cloud infrastructure are allocated to the Internal Data Center. These allocations formalize the item boundaries from Fig. 4.

We provided an informal description of the relationships between logical, task, and platform architecture using the example. In AutoFOCUS3 [12], the allocations from logical architecture to task architecture and from task architecture to platform architecture are defined in the form of allocation tables.

## SeCloud

SeCloud is a tool for automating the computation of TARA activities. The current implementation of SeCloud automates four TARA activities, namely threat conditions, threat scenarios, attack paths, and security architecture patterns for mitigating threat scenarios. These patterns are specifically designed for cloud-based systems, aiming to mitigate threat scenarios that may arise within cloud infrastructures. SeCloud has been implemented in the logic programming tool clingo [10]. SeCloud has been introduced in our previous article [6]. This article provides a more comprehensive

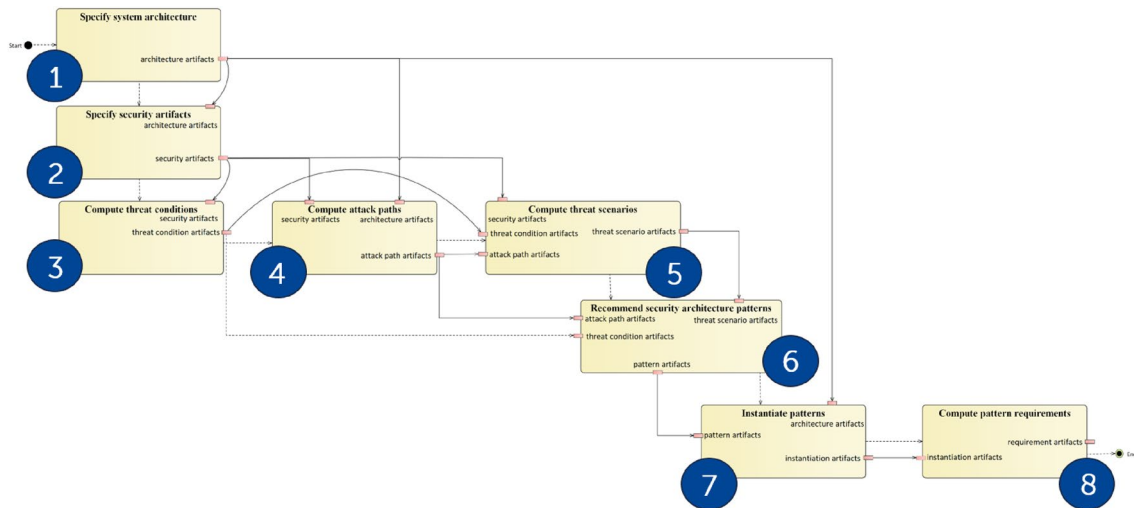


Fig. 6 SeCloud: activity diagram

description of SeCloud, including further implementation descriptions. Although the main focus of this article is on the cloud, we present SeCloud in a more generic manner.

SeCloud provides lightweight semantics for key artifacts computed by activities performed throughout the design of the system architecture (i.e., during the concept phase). By lightweight semantics, we mean a defined vocabulary for which the meaning is understood in the corresponding domain. We consider three domains, namely systems engineering, security and cloud. Key artifacts from such domains for automating TARA activities are system architecture artifacts, security artifacts computed by TARA activities, and cloud-based patterns. By incorporating lightweight semantics for key artifacts into a (domain specific) language, machines may understand their meaning, interpret, and make informed decisions based on the meaning of such artifacts.

Before delving into technical details (i.e., how SeCloud represents lightweight semantics to key artifacts and reason to automate TARA activities), we provide a high-level description of SeCloud’s behavior. This is illustrated through the activity diagram depicted in Fig. 6. Actions in the activity diagram are represented as yellow rectangles, with the action title written at the top of each rectangle. Each action may contain one or more parameters represented as light red rectangles. Input parameters are positioned on the left side or at top of the action and output parameters are positioned on the right side or at the bottom of the action. Control flows from one action to another are represented as dashed lines, while object flows from output to input parameters are represented as normal line. The blue circles represents the sequence of the actions.

The first two actions denote the specification of system architecture and security artifacts. Examples of system architecture artifacts are logical components and communication channels between logical components, which may

be modeled using a model-based system engineering tool. Regarding the security artifacts, SeCloud expects that at least assets are specified, such as which logical components shall be protected from a security perspective. The action Compute threat conditions receives the specified assets as input and compute one or more threat conditions, i.e., which security properties are relevant for the asset. Next, the action Compute attack paths receives the specified assets and system architecture artifacts as input and computes attack paths. The action Compute threat scenarios receives the specified assets and attack paths as input and computes one or more threat scenarios that may violate the security properties of the corresponding asset. The action Recommend security architecture patterns recommends a set of suitable cloud-based patterns based on the outputs from actions Compute threat conditions, Compute threat scenario and Compute attack path. The next action Instantiate patterns provides an instantiation of each recommended pattern into the system architecture. Finally, the last action Compute pattern requirements provides requirements for the recommended patterns, i.e., requirements that shall be implemented and tested to ensure that the pattern works as intended.

The rest of this section describes SeCloud, including the implementation details that have been implemented in clingo. To help the reader in comprehending these implementation details, we present a brief description of clingo.

**Clingo** [10]. Clingo is an open-source software tool used for solving logic programs implemented based on Answer-Set Programming (ASP) semantics [13]. A logic program consist of rules that define relationships and constraints. Each rule is of the form  $a_0 \leftarrow a_1, \dots, a_n$ , where the literal  $a_0$  is the head of the rule, and the literals  $a_1, \dots, a_n$  are the body of the rule. A literal is an atom ( $a_m$ ) or a negated atom ( $\neg$

**Table 1** SeCloud: DSL for selected architecture elements and security artifacts

| Fact   | Description  |
|--|--|
| <b>Architectural elements</b>  |  |
| <code>component_inp(id,id<sub>i</sub>)</code>                            | id is a logical component and it has an input port id <sub>i</sub> .   |
| <code>component_out(id,id<sub>o</sub>)</code>                            | id is a logical component and it has an output port id <sub>o</sub> .  |
| <code>channel(id,id<sub>o</sub>,id<sub>i</sub>)</code>                   | id is a logical channel connecting an output port id <sub>o</sub> to an input port id <sub>i</sub> , where the ports belong to different logical components.   |
| <code>pub(id,tp,id<sub>o</sub>)</code>                                   | id is a logical component that publishes topic tp through output port id <sub>o</sub> .  |
| <code>sub(id,tp,id<sub>i</sub>)</code>                                   | id is a logical component that subscribers to topic tp through input port id <sub>i</sub> .  |
| <code>task_inp(id,id<sub>i</sub>)</code>                                 | id is a task container and it has an input port id <sub>i</sub> .  |
| <code>task_out(id,id<sub>o</sub>)</code>                                 | id is a task container and it has an output port id <sub>o</sub> .   |
| <code>signal(id,id<sub>o</sub>,id<sub>i</sub>)</code>                    | id is a signal connecting an output port id <sub>o</sub> to an input port id <sub>i</sub> , where the ports belong to different task containers.   |
| <code>hardware_inp(id,id<sub>i</sub>)</code>                             | id is a hardware unit and it has an input port id <sub>i</sub> .   |
| <code>hardware_out(id,id<sub>o</sub>)</code>                             | id is a hardware unit and it has an output port id <sub>o</sub> .  |
| <code>transmission(id,id<sub>o</sub>,id<sub>i</sub>)</code>              | id is a transmission channel connecting an output port id <sub>o</sub> to an input port id <sub>i</sub> , where the ports belong to different hardware units.  |
| <code>gateway(id)</code>   | id is a gateway element.   |
| <code>public(id)</code>  | id is a public element that may be accessible by external users.   |
| <code>allocation(id<sub>1</sub>,id<sub>2</sub>)</code>                   | denotes the allocation of element id <sub>1</sub> to id <sub>2</sub> , such the allocation of task container id <sub>1</sub> to hardware unit id <sub>2</sub> .  |
| <b>Security artifacts</b>  |  |
| <code>asset(id)</code>   | denotes that element id is an asset.   |
| <code>threat_condition(id,secp,id<sub>ast</sub>)</code>                  | id is the adverse consequence if the security property secp of asset id <sub>ast</sub> is violated, where secp ∈ {authenticity, integrity, authorization}.   |
| <code>threat_scenario(id,ts<sub>des</sub>,as,ts,id<sub>ast</sub>)</code> | id is a threat scenario, with description ts <sub>des</sub> , originating from element as, to violate the security property of asset id <sub>ast</sub> with threat ts, where ts ∈ {spoofing, tampering, elevation_of_privilege}. |
| <code>attack_path(id,el,id<sub>ast</sub>)</code>                         | id is an attack path denoting that malicious data may be injected from element el to target asset id <sub>ast</sub> .  |

$a_m$ ). A rule is considered valid if the body of the rule is valid. A rule with an empty head is a constraint. An answer set is an interpretation satisfying a set of rules and constraints. When an answer set is found by clingo, it provides a possible solution to the given problem. If there are multiple answer sets, each one represents a distinct solution. In the context of this article, we refer to these clingo solutions as architecture solutions. This article uses the clingo notation,

where :- denotes  $\leftarrow$ , and not denotes  $\neg$ . Identifiers beginning with capital letters (e.g., Ax, By) denote variables that during clingo's execution are instantiated by appropriate terms. Identifiers beginning with a lower-case letter (e.g., a, b) are constants. The \_ (underscore) character specifies that the argument can be ignored in the current rule.



**Table 2** SeCloud: DSL for security architecture patterns [6]

| Fact  | Description   |
|---|---|
| <b>Security architecture patterns</b>       |   |
| security_pattern<br>(id,pat,cp,inp,int,out) | id is the unique identification of security architecture pattern pat. This pattern consists of a list of components cp. The last three parameters inp, int and out denote, respectively, the input, the internal, and the output channels related to the pattern.   |
| pattern_attribute<br>(pat,as,secp,ts)       | pat is a security architecture pattern suitable for satisfying the security property secp by mitigating threat ts at the attack surface as.<br>$as \in \{external\_interface, internal\_interface\}$ ,<br>$secp \in \{authenticity, integrity, authorization\}$ ,<br>$ts \in \{spoofing, tampering, elevation\_of\_privilege\}$ . |

### Domain-Specific Language

SeCloud provides a Domain-Specific Language (DSL) for specifying system architecture artifacts, security artifacts, and security architecture patterns. The goal is to provide a precise specification of such artifacts and patterns to facilitate the automatation of TARA activities. Table 1 describes our DSL for specifying selected system architecture artifacts and security artifacts. The DSL for specifying system architecture artifacts consists of different architecture views: logical architecture, task architecture, and platform (a.k.a.

hardware) architecture. Our DSL provides a predicate **allocation** to specify the allocation from logical components to task containers and from task containers to hardware units. A hardware unit may be annotated as either a public element or a gateway element. The former specifies a hardware unit that may be accessed by external users (e.g., attackers), while the latter specifies a gateway sitting between a trusted and untrusted network. We expect that the user of SeCloud specifies both the system architecture artifacts and security artifacts (at least the identified assets). This specification can be done manually or derived in an automated fashion

**Table 3** Instantiation of the OpenID Connect pattern. The assumptions represent only a selection

|                         | Description   | SeCloud Specification   |
|-------------------------|---|---|
| Name                    | OpenIDConnect   | NAME=openIDConnect;   |
| Structure               |   | COMPONENTS=[s1,s2,server];<br>INTERNAL_CH=[int1,int2,int3];<br>OUTPUT_CH=[out1];  |
| Intent                  | This pattern is used when a client (Service 1) wants to authenticate herself before accessing cloud services (Service 2). The authentication process is performed by the OpenIDConnect Server and confirmed by (Service 2). | TYPE_SEC_PROPERTY=[authenticity];<br>TYPE_THREAT=[spoofing];<br>TYPE_ATTACK_SURFACE=[external_interface, internal_interface]; |
| Problem addressed       | This pattern prevents spoofing attacks performed by the client. The client may be an external entity or an internal entity.   |   |
| Assumptions (selection) | Service 1 has has obtained the credentials from OpenIDConnect Server.   | TYPE_ASSUMPTION=has_obtained_credentials_from_OpenID_Connect_Server;<br>COMPONENTS=[s1,server];                               |

from a model-based system engineering tool, as shown in Section “[Security Pattern Synthesis](#)”.

*Example:* We provide an example on how a user of SeCloud may specify one logical component, one task container, and one hardware unit from the logical, task, and platform architectures illustrated, respectively, in Figs. 3, 4, and 5.

```
% Representation of a logical component
component_out(client,client_out_port).

% Representation of a task
task_out(task_client,tClient_out_port).

% Representation of a task
hardware_out(external,ext_out_port).
public(external).

% Representation of allocation (component -> task)
allocation(client,task_client).
allocation(client_out_port,tClient_out_port).

% Representation of allocation (task -> hardware)
allocation(task_client,external).
allocation(tClient_out_port,ext_out_port).
```

As discussed in Section “[Security Reasoning Principles](#)”, SeCloud may reason about the above representation in order to, e.g., infer that task `task_client` and logical component `client` are public elements by interpreting the facts specified in the `allocation` predicate.

Our DSL provides predicates specifically tailored for specifying security architecture patterns. It consists of predicates that allow for the specification of

- (i) the attributes of security architecture patterns, and
- (ii) the instantiation of security architecture patterns.

The attributes of a security architecture pattern includes the intent of the pattern and the problem addressed by the pattern, while the instantiation of a security architecture pattern consists of pattern components and channels. Specifically, the pattern attribute consists of the desired security property achieved by the pattern (i.e., intent of the pattern), the threat addressed by the pattern, and the attack surface suitable to instantiate the pattern. The attack surface is an abstract term that describes whether an attack path includes public elements or not. It provides a distinction between internal interfaces, which correspond to attack paths originating from within the system boundary without involving public elements, and external interfaces, which correspond to attack paths that do involve public elements. Table 2 provides an overview and description of such predicates and the corresponding facts.

The current implementation of SeCloud supports five security architecture patterns, namely Cilium, TLS, Mutual

Authentication, OpenID Connect, and Service Mesh Istio. Our previous article [6] explained the specification of the predicates from Table 2 for the Mutual Authentication pattern. This article focuses on providing explanations for the specification of the OpenID Connect pattern.

**OpenID Connect.**<sup>4</sup> OpenID Connect is an authentication protocol built on top of OAuth 2.0. OpenID Connect allows a client to authenticate herself when accessing web or cloud services. An OpenID Connect server verifies the client’s credentials (e.g., username and password) and issues identity tokens [6]. The authenticity of these identity tokens is subsequently verified by a cloud service, ensuring a secure authentication process against spoofing attacks. OpenID Connect may be used by either internal cloud services or an external client trying to access an internal cloud service. The instantiation of the OpenID-Connect pattern is shown in Table 3.

Based on the content of OpenIDConnect described in Table 3, SeCloud instantiates the predicates `security_pattern` and `pattern_attribute` as follows:

```
% Pattern instantiation
security_pattern(id,openIDConnect,(s1,s2,server),
none,(int1,int2,int3),(out1)).

% Pattern attribute
pattern_attribute(openIDConnect,
attack_surface(external_interface,internal_interface),
property(authenticity),
threat(spoofing)).
```

## Security Reasoning Principles

SeCloud provides reasoning principles to automate TARA activities based on the proposed DSL. That is, by leveraging SeCloud’s DSL, one can specify system architecture artifacts, security artifacts, and security architecture patterns, and query clingo’s reasoning engine to automate TARA activities. SeCloud automates the computation of four TARA activities, namely threat condition, attack paths, threat scenarios, and the recommendation of security architecture patterns to mitigate threat scenarios.

SeCloud performs its analysis at the task architecture, in particular for enumerating attack paths. To this end, SeCloud derives relevant information annotated to either the logical architecture or the platform architecture to the task architecture. For example, assets are specified in the logical architecture, and public or gateway hardware units are specified in the platform architecture. SeCloud derives assets and public elements (gateway is derived in a similar way) to tasks as follows:

<sup>4</sup> <https://openid.net/connect>

```

% Reasoning rule for deriving assets to tasks
asset(T1) :-
    asset(C1),
    is_component(C1),
    is_task(T1),
    allocation(C1,T1).

% Reasoning rule for deriving public elements to tasks
public(T1) :-
    public(HW1),
    is_hardware(HW1),
    is_task(T1),
    allocation(T1,HW1).

```

*Example:* Consider, e.g., the asset **Service Broker** in the logical architecture depicted in Fig. 3. Based on the first rule, the task **SB**, from the task architecture illustrated in Fig. 4, will become an asset.

**Threat Conditions.** SeCloud considers the assets provided as input information by the user to compute threat conditions. A threat condition includes the security property of the asset that shall be protected from a security perspective. The current implementation of SeCloud considers three security properties, namely authenticity, authorization, and integrity. SeCloud implements the following rule to enable the automated computation of threat conditions.

```

% Reasoning rule for computing threat conditions
threat_condition(ID_TC,PROPERTY,ASSET) :-
    asset(ASSET),
    security_property(PROPERTY),
    get_tc_unique_id(ASSET,PROPERTY,ID_TC).

% facts specifying the considered security properties
security_property(authenticity).
security_property(authorization).
security_property(integrity).

```

*Example:* Consider, e.g., the asset **SB** in the task architecture illustrated in Fig. 4. SeCloud computes three threat conditions for **SB**, i.e., one threat condition for each security property, namely authenticity, authorization, and integrity. Notice that additional properties may be considered, such as availability.

**Attack Paths.** SeCloud considers the specified system architecture artifacts along with an intruder model to compute attack paths. Our intruder model is based on the intruder models proposed by [14] and [4], which are referred to as the external and internal intruders, respectively.

- The **external intruder** [14] may inject malicious data into the cloud through public interfaces. The intruder intends to violate the security property of assets (inside the system boundary) from outside the system boundary.

- The **internal intruder** [4] is a powerful intruder that may access every port within the task/container architecture, all within the system boundary. As a result, the internal intruder may inject malicious data into other containers (assets), ultimately compromising the security properties of these assets.

The initial step in computing attack paths based on the external intruder is the identification of the entry points in the architecture. This information is obtained from the specified system architecture artifacts in which public elements are assigned to hardware units in the platform architecture. SeCloud performs the attack path analysis in the task architecture, where entry points may be inferred through the facts specified in the allocation predicate, i.e., every task allocated to a public hardware are themselves considered public elements. The assets are specified to components within the logical architecture. Similarly, which tasks are assets may be inferred through the facts specified in the allocation predicate. Based on the existing information flow specified in the task architecture, SeCloud computes every possible path starting from the asset and tracing backward until reaching an entry point. SeCloud implements recursive rules for computing such attack paths:

- **Base case:** The identified asset task and its input port are added to the path.
- **Inductive case 1:** If Task (possibly the identified asset) and its input port are in the path, and there exists another task (Task') that writes to Task, then Task' and its output port are added to the path.
- **Inductive case 2:** If Task and its output port are in the path, and there exists another task (Task') that writes to Task, then Task and its input port are added to the path.

The inductive cases continue until reaching a port belonging to a public element. For details about our rules, we refer the interested reader to SeCloud's implementation [11].

*Example:* Consider the task architecture illustrated in Fig. 4, where **Client** is a public element and **SB** is an asset. One of the attack paths computed by SeCloud is:

Client → Ingress → SB

The key step in computing attack paths, based on the internal intruder, involves identifying assets and determining which tasks might be able to write to those identified assets. SeCloud operates under the assumption that all ports within the system boundary are accessible to the intruder. This assumption is valid because each container port is deployed to a cloud service, often managed by a third-party provider. SeCloud implements a reasoning rule for computing attack paths based on the internal intruder. This rule is outlined below. This rule checks whether another task (SOURCE) can write to the task asset through a signal communication.

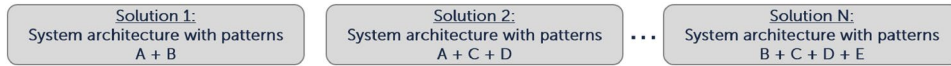


Fig. 7 SeCloud: Illustration of all possible solutions (system architecture with patterns)

Notice that this rule is applicable only if SOURCE is not a public element.

```
% Reasoning rule for computing attack
paths (internal intruder)
%% Intruder may masquerade (i.e., pretend to be)
a port of a container.
attack_path(ID_AP, SOURCE, ASSET) :-
    asset(ASSET),
    is_task(ASSET),
    is_task(SOURCE),
    not public(SOURCE),
    task_out(SOURCE, SRC_PORT),
    task_inp(ASSET, ASSET_PORT),
    signal(ID_SIGNAL, SRC_PORT, ASSET_PORT),
    define_ap_unique_id(SOURCE, ASSET, ID_AP).
```

```
% Reasoning rule for computing threat scenarios
threat_scenario(ID_TS, TS_DESCRIPTION, SOURCE, THREAT, ASSET) :-
    asset(ASSET),
    threat_condition(ID_TC, PROPERTY, ASSET),
    get_stride_mapping(PROPERTY, THREAT),
    attack_path(ID_AT, SOURCE, ASSET),
    get_threat_description(SOURCE, THREAT, TS_DESCRIPTION)
    define_ts_unique_id(ASSET, PROPERTY, ID_AT, ID_TS).
```

Example: Consider the task architecture illustrated in Fig. 4, where Client is a public element and SB is an asset. Additionally, consider the security property authentication for SB and the attack path Client → Ingress → SB. SeCloud may compute the following threat scenario, where the threat description is derived based on the selected arguments as shown in the above rule.

```
% Example of a threat scenario
threat_scenario(id_tr1,
    attacker_enters_malicious_input_at_interface,
    client_output_port,
    spoofing,
    sb).
```

Example: Consider the task architecture illustrated in Fig. 4, where SB is an asset. One of the attack paths computed by SeCloud is:

Event Broker (Kafka) → SB

**Threat Scenarios.** SeCloud considers the specified assets and their corresponding threat conditions to compute threat scenarios. Additionally, SeCloud also specifies as part of a threat scenario the source element that may initiate the attack. The source element is obtained from attack paths. SeCloud uses STRIDE [15] to map security properties (specified by threat conditions) to threats. This mapping is illustrated below for the currently three security properties considered by SeCloud.

- authenticity → spoofing
- authorization → elevation of privilege
- integrity → integrity

SeCloud automates the computation of threat scenarios with the following rule:

**Security Architecture Patterns.** SeCloud provides reasoning rules to enable the automated recommendation of security architecture patterns. To this end, SeCloud considers the computed threat conditions, threat scenarios, and attack paths. As discussed earlier, SeCloud provides a predicate, namely pattern\_attribute for specifying the intent and problem addressed by the pattern. By using this predicate along with the provided input information, SeCloud takes into account the following conditions for recommending security architecture patterns:

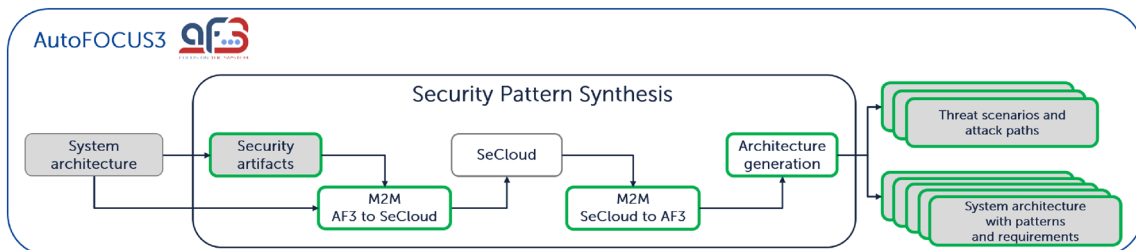


Fig. 8 Security Pattern Synthesis: High-level architecture. The artifacts introduced in this article are highlighted in green

$\text{attack\_surface} \in \text{pattern\_attribute}$   
 $\text{threat} \in \text{pattern\_attribute}$   
 $\text{security\_property} \in \text{pattern\_attribute}$

SeCloud will recommend a pattern whenever those conditions are met. The next reasoning rule implements the recommendation of security architecture patterns.

```

% Reasoning rule for recommending patterns
recommended_security_pattern(PAT, IDTS, IDAT) :
  pattern_attribute(PAT, AS, PROPERTY, THREAT) } = 1
  :- threat_condition(ID_TC, PROPERTY, ASSET),
     threat_scenario(ID_TS, TS_DESCRIPTION, _, THREAT, ASSET),
     get_attack_path_from_threat_scenario(ID_TS, ID_AT),
     get_attack_surface_from_attack_path(ID_AT, AS).

```

Once the pattern is recommended (i.e., the right side of the above rule holds), SeCloud instantiates the pattern into the system architecture. Our instantiation rules are tailored to each pattern, i.e., specific rules are implemented for each security architecture pattern. We show below how we instantiate the OpenIDConnect pattern.

```

% Example: Reasoning rule for instantiating patterns (OpenID Connect)
% This rule is specified for when the next element in the path
% (external intruder) is a gateway
% The prefix 'x' denotes that the pattern has been recommended by SeCloud
xsecurity_pattern(id_security_pattern(openidc, T1, T2), openidConnect,
  elem(T1, T2, openidServer(T1)), nuINPUTS, nuINTERNAL, nuOUTPUTS)
  :- recommended_security_pattern(openidConnect, ID_TS, ID_AT),
     get_source_from_attack_path(ID_AT, T1),
     public(T1),
     get_next_element_in_path(ID_AP, T1, NEXT),
     gateway(NEXT),
     get_next_element_in_path(ID_AP, NEXT, T2),
     not gateway(T2).

% Reasoning rule for instantiating the 'security_pattern' predicate
security_pattern(ID_PAT, openidConnect, ELEM, INPUT, INTERNAL, OUTPUT)
  :- xsecurity_pattern(ID_PAT, openidConnect, ELEM, INPUT, INTERNAL, OUTPUT).

```

The above rule considers the recommended security pattern, in particular the last argument (i.e., attack path) specified in `recommended_security_pattern`. From the attack path, SeCloud obtains the (malicious) source (i.e., the first element) and checks whether the source is a public element. This is always true if the attack path is computed based on the external intruder. Next, SeCloud checks the next element in the path (i.e., the next element after the source element). The OpenIDConnect pattern is intended to be instantiated between two services, with the first service (i.e., source) sending an identity token to the second service. It is crucial that the authentication of the token does not take place through a gateway, but rather by a cloud service within the system boundary. To ensure this, SeCloud checks whether the next element in the path is a gateway. If SeCloud identifies a gateway, it proceeds to retrieve the subsequent element in the path (referred as T2 in the above rule). As a result, SeCloud instantiates the pattern components, namely T1, T2 and `openidServer(T1)`. The pattern channels may be derived in another rule by taking into account the system architecture artifacts.

SeCloud computes pattern assumptions that shall be valid to ensure that the pattern works as intended. We show below one of the assumptions computed by SeCloud for the OpenID Connect pattern. It assumes that *the source service (i.e., T1) has obtained the credentials from the server (i.e. openidServer)*.

```

% Example: Reasoning rule for computing pattern
% assumptions (OpenID Connect)
assumption(ID, has_obtained_credentials_from_
  OpenIDConnect_Server,
  elem(T1, openidServer)) :-
  security_pattern(ID_PAT, openidConnect, elem
  (T1, T2, openidServer), _, _, _).

```

The pattern assumptions may translated into requirements that shall be realized during the system development. SeCloud categorizes these requirements into two groups, namely application requirements and infrastructure requirements. The former denotes requirements that need to be considered in the development of the applications themselves. The latter denotes requirements that need to be implemented when building the infrastructure and deploying the components. By splitting the requirements into these categories, our goal is to assist in selecting the most appropriate solution for their system. For example, a lower number of application requirements would require less effort from developers, as security measures have already been integrated into the system's infrastructure.

**Design-Space Exploration.** Security architecture patterns are recommended to mitigate threat scenarios. SeCloud explores which specified patterns are suitable to mitigate such threat scenarios. That is, SeCloud recommends a security architecture pattern when certain conditions are met. By building on the clingo solver, SeCloud computes all possible solutions (i.e., system architecture with security architecture patterns) that satisfies the implemented reasoning rules. Figure 7 depicts several possible solutions that may be computed by SeCloud.

We acknowledge the possibility of a solution explosion, where the number of solutions becomes vast, resulting in issues related to scalability and usability. To address those issues, SeCloud implements a set of constraints to streamline the search process, focusing on the most suitable solutions for the user's benefit. For example, SeCloud enables users to trace which threat scenario has been mitigated by a particular security architecture pattern. This traceability is facilitated through the `mitigated_by(ID_TS, ID_PAT)` predicate, which consists of two parameters, namely the threat scenario ID and the pattern ID. With the help of this predicate, SeCloud provides constraints to filter out solutions where (i) a threat scenario has not been mitigated, and (ii) more than one pattern is instantiated for the same

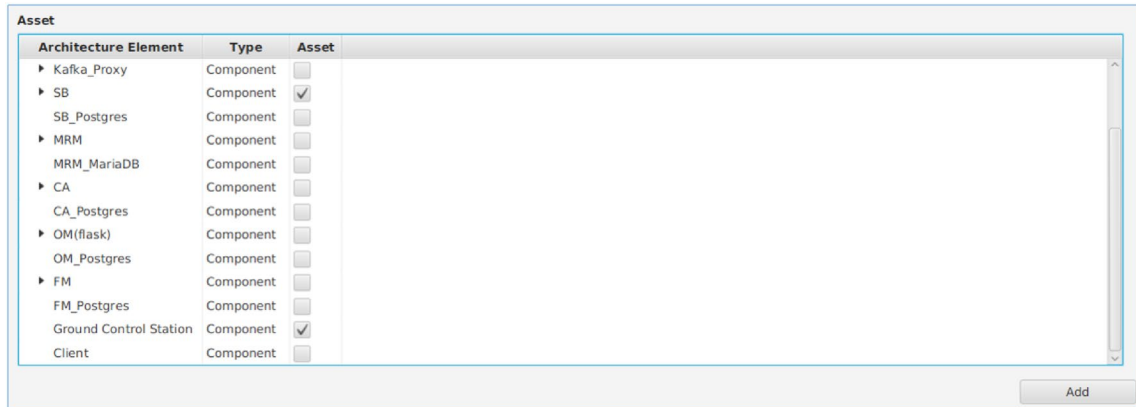
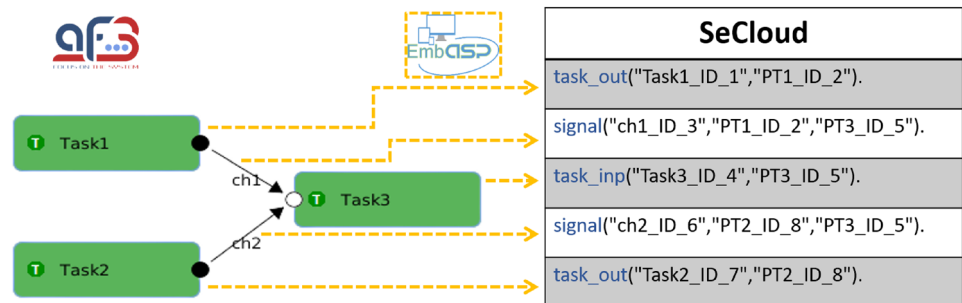


Fig. 9 Security Pattern Synthesis: Screenshot of the wizard for specifying assets

Fig. 10 Security Pattern Synthesis: Example of M2M transformation: AF3 to SeCloud



threat scenario. To further enhance the selection process, we implement pattern-based constraints where we investigate scenarios where certain solutions may not be optimal. For example, SeCloud filter out solutions where the TLS and Mutual Authentication pattern are recommended for the same element. This decision is taken to avoid redundant solutions where the element would unnecessarily carry two separate certificates, one for TLS and one for Mutual Authentication. Our article [6] provides descriptions of all constraints implemented by SeCloud.

By making use of clingo solver (including the specification of constraints), SeCloud achieves automatic search and optimization to reduce the number of solutions while preserving the most suitable solutions for the user’s benefit. This constitutes a significant advantage of using ASP solvers like clingo over other languages, e.g., Java, where the optimization process could be time-consuming and prone to errors.

### Security Pattern Synthesis

We propose Security Pattern Synthesis, a model-based systems engineering plugin for hardening cloud architectures with security architecture patterns. Security Pattern Synthesis has been implemented as part of the Design Space Exploration (DSE) of AutoFOCUS3 (AF3) [12]. SeCloud

functions as the backend of the proposed plugin to reason about the security of the system architecture.

The proposed plugin has been developed in Java, while SeCloud has been developed in the logic programming language clingo. EmbASP [16] is a framework for the integration of logic programming into external systems, such as systems implemented in Java. We use EmbASP to enable the Java representation of the predicates specified in SeCloud. That is, we use EmbASP for the model-to-model (M2M) transformation from AF3 (Java) to SeCloud (clingo) and from SeCloud (clingo) to AF3 (Java).

Figure 8 depicts the high-level architecture of Security Pattern Synthesis. The input artifacts and output artifacts are depicted in gray. The artifacts highlighted in green denote the artifacts introduced by this article.


- System architecture:** This input artifact consists of the designed system architecture in the modeling view of AF3. The system architecture consists of the following architecture views: logical architecture, task architecture, platform (a.k.a. hardware) architecture, and allocation tables. The publish/subscriber pattern for SOA is implemented in the logical architecture enabling the specification of publisher ports, subscriber ports, and topics. The task architecture consists of tasks realizing the implementation of logical components as software units. The


**Fig. 11 Security Pattern Synthesis:** Example of M2M transformation: SeCloud to AF3

## SeCloud

```

threat_scenario(ts_ID_0,
  attacker_gains_elevated_access_to_resources_and_sends_malicious_input_at_interface,
  "user_output_ID_10167",
  elevation_of_privilege,
  "Task_Ground_Control_Station_ID_9967").
        
```





| *Threat Scenario Analysis |  |             |      |                          |
|---------------------------|--|-------------|------|--------------------------|
| Name                      | Description                                  | Source      | Type | Asset                    |
| ▼ Threat Scenario A...    |  |             |      |                          |
| Threat Scenario 0         | attacker gains elevated access to resourc... | user_output | EOP  | Task_Ground_Control_S... |

Recommended Security pattern

Pattern configuration

Criteria for users

| Visualization |  |   |                  |                     |                             |                 |                       |
|---------------|--|---|------------------|---------------------|-----------------------------|-----------------|-----------------------|
| ▲ Model       | Patterns   | Configuration   | #Application Req | #Infrastructure Req | #Pattern Support Components | Select Solution | Export to AF3 Project |
| Solution 1    | mTLS(3)<br>Istio Service Mesh(1)<br>OpenID Connect(1)<br>TLS(1)              | "Task_Kafka_Proxy_ID_12808" is part of istio service mesh<br>"Task_Ingress_ID_12756" possesses a tls certificate<br>"Task_MRM_ID_12658" is part of istio service mesh<br>"Task_Dashboard_ID_12800" is part of istio service mesh  | 19               | 34                  | 4                           | Select          | Export                |
| Solution 2    | mTLS(1)<br>Istio Service Mesh(1)<br>OpenID Connect(1)<br>Cilium(1)<br>TLS(1) | "Task_MRM_ID_12658" is part of istio service mesh<br>"Task_Ingress_ID_12756" possesses a tls certificate<br>"Task_Dashboard_ID_12800" is part of istio service mesh<br>"Task_Kafka_Proxy_ID_12808" is part of istio service mesh  | 11               | 22                  | 5                           | Select          | Export                |
| Solution 3    | mTLS(2)<br>Istio Service Mesh(1)<br>OpenID Connect(2)<br>TLS(2)              | "Task_MRM_ID_12658" is part of istio service mesh<br>"Task_Ground_Control_Station_ID_12820" possesses a tls certifi<br>"Task_Dashboard_ID_12800" is part of istio service mesh<br>"Task_Kafka_Proxy_ID_12808" is part of istio service mesh<br>"Task_Ground_Control_Station_ID_12820" is part of istio service<br>"Task_Ingress_ID_12756" possesses a tls certificate   | 19               | 37                  | 6                           | Select          | Export                |
| Solution 4    | Istio Service Mesh(1)<br>OpenID Connect(2)<br>Cilium(1)<br>TLS(2)            | "Task_MRM_ID_12658" is part of istio service mesh<br>"Task_Dashboard_ID_12800" is part of istio service mesh<br>"Task_Kafka_Proxy_ID_12808" is part of istio service mesh<br>"Task_Ingress_ID_12756" possesses a tls certificate<br>"Task_Ground_Control_Station_ID_12820" is part of istio service<br>"Task_Ground_Control_Station_ID_12820" possesses a tls certifi   | 11               | 25                  | 7                           | Select          | Export                |
| Solution 5    | mTLS(1)<br>Istio Service Mesh(1)<br>OpenID Connect(1)<br>TLS(1)              | "Task_Ingress_ID_12756" possesses a tls certificate<br>"Task_Dashboard_ID_12800" is part of istio service mesh<br>"Task_SB_ID_12639" is part of istio service mesh<br>"Task_Event_Broker_Kafka_ID_12770" is part of istio service m<br>"Task_MRM_ID_12658" is part of istio service mesh<br>"Task_SB_Postgres_ID_12650" is part of istio service mesh<br>"Task_Kafka_Proxy_ID_12808" is part of istio service mesh  | 11               | 37                  | 4                           | Select          | Export                |
| Solution 6    | Istio Service Mesh(1)<br>OpenID Connect(2)<br>TLS(2)                         | "Task_Dashboard_ID_12800" is part of istio service mesh<br>"Task_Kafka_Proxy_ID_12808" is part of istio service mesh<br>"Task_Ground_Control_Station_ID_12820" is part of istio service<br>"Task_Ingress_ID_12756" possesses a tls certificate<br>"Task_Event_Broker_Kafka_ID_12770" is part of istio service m<br>"Task_MRM_ID_12658" is part of istio service mesh<br>"Task_SB_Postgres_ID_12650" is part of istio service mesh<br>"Task_SB_ID_12639" is part of istio service mesh<br>"Task_Ground_Control_Station_ID_12820" possesses a tls certifi | 11               | 40                  | 6                           | Select          | Export                |

**Fig. 12 Security Pattern Synthesis:** Screenshot of the wizard for visualizing architecture solutions with security architecture patterns

platform architecture consists of hardware units, where annotations are allowed to specify which hardware units are public or which hardware units are gateways. The allocation table provides a traceability between different architecture views. Specifically, the allocation table consists of allocations of logical components to tasks, and tasks to hardware units. By checking the allocation tables, one can check which tasks are public tasks or gateway tasks. The annotation of which elements are public or gateways is required by SeCloud to enable the enumeration of attack paths in an automated fashion.

- Security artifacts:** This input artifact consists of the results of the first TARA activity, namely asset identification. The security artifacts provided as input consists of assets specified as components designed in the logical architecture. The user may define one or more assets that needs to be protected from a security perspective. To this end, we have implemented a wizard to facilitate users in specifying assets. Figure 9 shows a screenshot of our wizard for specifying assets. This illustration showcases the specification of two assets, namely the logical components SB and Ground Control Station. This speci-

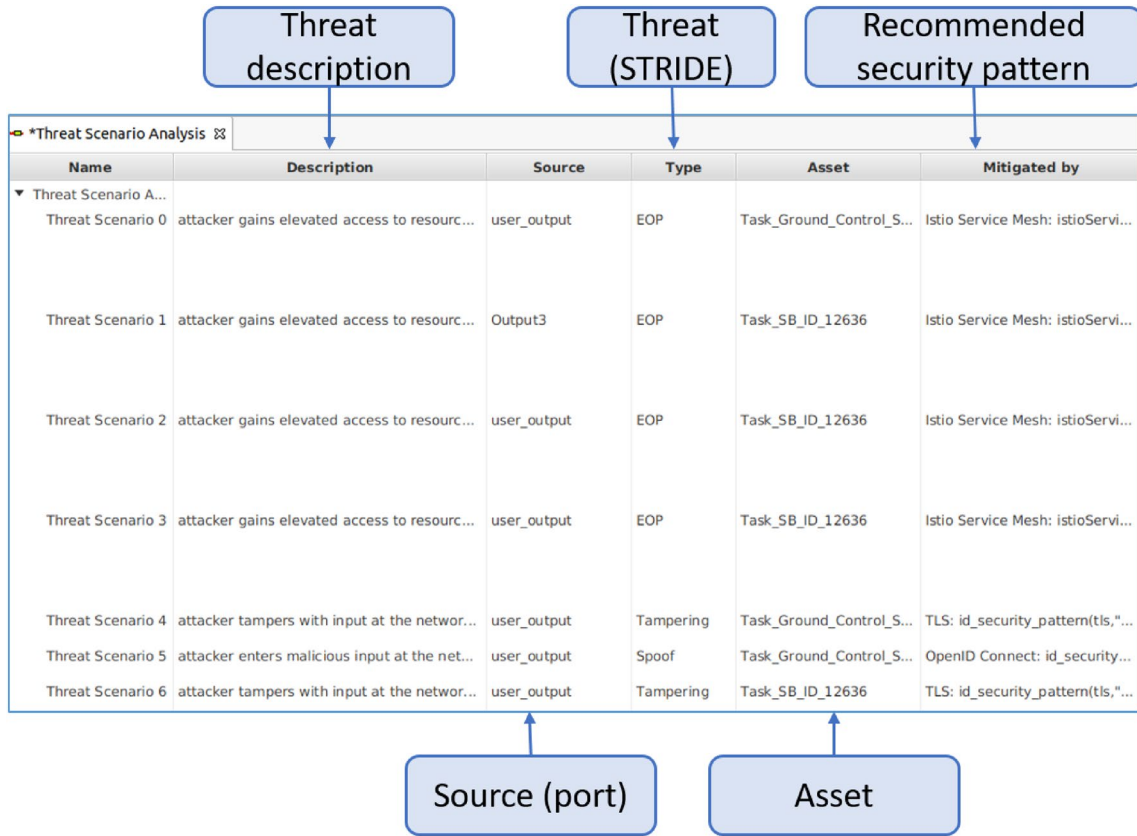


Fig. 13 Security Pattern Synthesis: Screenshot of the modeled threat scenarios in AutoFOCUS3

fication provides a traceability between assets and the logical architecture, denoting that such components of the logical architecture are annotated as assets. The introduction of our wizard improves the usability of SeCloud by transitioning from a command-line asset specification to a GUI-based specification.

from each architecture view of AF3, and the representation of security artifacts. Each predicate is represented by a Java class. The right side of Fig. 10 illustrates the instantiation of three predicates in SeCloud, namely task\_out(TASK\_ID,OUT\_PORT), task\_inp(TASK\_ID,INP\_PORT), and signal(SIGNAL\_ID,OUT\_PORT,INP\_PORT).

- M2M Transformation from AF3 to SeCloud:** This artifact provides a M2M transformation from AF3 to SeCloud. That is, we have implemented a model-to-model transformation to transform the system architecture models and security artifacts models to SeCloud models. Figure 10 illustrates a M2M transformation from a simplistic task architecture modeled in AF3 to SeCloud. The right side of Fig. 10 illustrates the representation of the predicates/facts specified in SeCloud based on the task architecture modeled in AF3 (left side). The task architecture contains three tasks, namely Task1, Task2 and Task3, and two signals connecting Task1 to Task3, and Task2 to Task3. The M2M transformation is implemented with the help of the EmbASP framework. We represent in Java each relevant predicate specified in the DSL of SeCloud. This includes the representation of architecture elements (e.g., components, tasks and hardware units)
- SeCloud:** SeCloud consists of a DSL for specifying system architecture artifacts, security artifacts, and security architecture patterns. Additionally, SeCloud consists of security reasoning rules for automating (i) the computation of threat conditions for the assets specified as input, (ii) the computation of threat scenarios for violating security properties of assets, (iii) the enumeration of attack paths for threat scenarios, and (iv) the recommending security architecture patterns for mitigating threat scenarios. SeCloud computes pattern requirements for each recommended security architecture pattern. These requirements shall be realized to ensure that the pattern works as intended. A more comprehensive description of SeCloud has been provided in Sect. 3.
- M2M Transformation from SeCloud to AF3:** This artifact provides a M2M transformation from SeCloud to AF3. That is, we have implemented a model-to-model transformation to



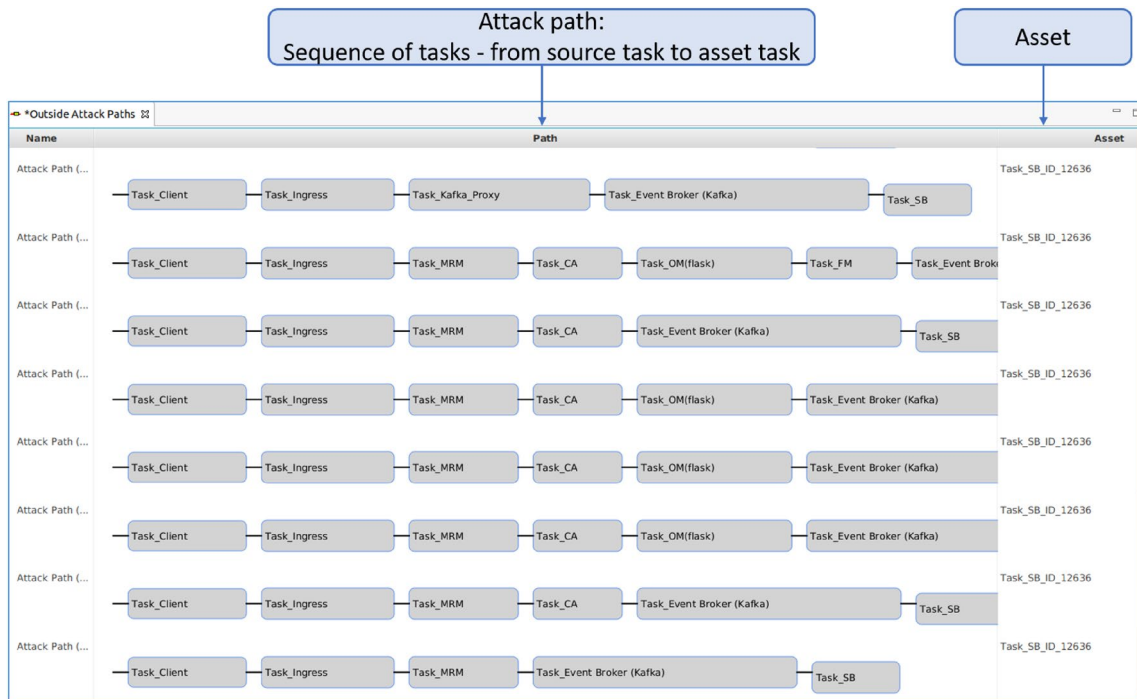


Fig. 14 Security Pattern Synthesis: Screenshot of the modeled attack paths in AutoFOCUS3

transform SeCloud models to AF3 models. The transformations primarily involve the following artifacts computed by SeCloud, namely (i) threat scenarios, (ii) attack paths, (iii) security architecture patterns, and (iv) pattern requirements. The transformation process is similar to the one shown in Fig. 10, albeit in a reverse order. Figure 11 illustrates a M2M transformation from a threat scenario computed by SeCloud to AF3. The upper part of Fig. 11 illustrates the facts in `threat_scenario` computed by SeCloud, while the lower part illustrates the transformed threat scenario modeled within AF3. The predicate `threat_scenario` contains the following arguments: threat scenario ID, threat scenario description, port of architecture element where the attack may be initiated, threat, and targeted asset. The M2M transformation is implemented with the help of EmbASP.

- Architecture generation:** This output artifact provides a set of solutions, i.e., system architectures along with their corresponding security architecture patterns. The system architectures are represented as AF3 models following our M2M transformation. We have implemented a user-friendly wizard for visualizing the solutions generated by SeCloud. The introduction of this wizard improves the usability of SeCloud by providing users with a GUI-based solution visualization rather than a command-line solution visualization. Figure 12 illustrates the developed wizard. This illustration showcases six architecture solutions computed by SeCloud

along with their respective security architecture patterns. The number displayed in parentheses next to the security architecture pattern denotes the required number of instances for that pattern, e.g., Solution 5 requires one instance for mTLS. The wizard describes three criteria, namely number of application requirements, number of infrastructure requirements, and number of pattern components. The intended outcome of these criteria is to assist the user of our plugin in determining the most suitable architecture solution for their needs. As described by the next artifact, threat scenarios and attack paths are also represented as AF3 models, which can be visualized once the user selects a suitable system architecture with its corresponding security architecture patterns.

- System architecture with patterns and requirements:** This output artifact provides the selected system architecture with security architecture patterns. The selected system architecture, along with the pattern requirements, is exported to the modeling view of AF3. The instantiation of security architecture patterns in the system architecture may be done automatically. In the current plugin implementation, Security Pattern Synthesis instantiates the OpenID Connect pattern in the task architecture. The specification for the other patterns is realized by means of security requirements. For example, a requirement for the Mutual Authentication pattern is ‘entities X and Y communicating over a channel Z

shall implement the mTLS protocol', where X, Y and Z are specific architecture elements. The provided security requirements shall be implemented and validated during the system development process. By using our plugin, patterns and requirements have been incorporated into the model, ensuring traceability between requirements and components of the architecture.

- **Threat scenarios and attack paths:** This output artifact provides threat scenarios and attack paths for the exported solution selected by the user. Both threat scenarios and attack paths are shown in the modeling view of AF3. Figure 13 depicts a screenshot of the threat scenarios in AF3. It contains the description of the threat scenario, the source (i.e., which port element initiates the threat scenario), the threat itself, and the targeted asset. The last column contains which security architecture pattern has been recommended to mitigate the corresponding threat scenario. Omitted in Fig. 13, our wizard also provides links to pattern requirements, i.e., ensuring traceability between a threat scenario and the requirements designed to mitigate that particular threat scenario. Figure 14 depicts a screenshot of the attack paths in AF3. This illustration showcases the attack paths computed by SeCloud based on the external intruder. The last column contains the targeted asset for each attack path, such as the asset Task\_SB\_ID\_12636 from the first attack path.

## Summary

We opted for a model-based approach to integrate SeCloud into AutoFOCUS 3 in the form of the Security Pattern Synthesis's plugin. This integration involved the translation of architecture layers within AutoFOCUS 3 into SeCloud's domain-specific language. This translation enables SeCloud to conduct a security analysis of the system architecture. Upon conducting the security analysis, SeCloud generates results that are subsequently translated back into AutoFOCUS 3. The introduction of Security Pattern Synthesis establishes traceability between (a) security artifacts and architecture elements, such as assets and components of the logical architecture, and (b) security artifacts, such as threat scenarios and requirements. As a result, users can effectively monitor and manage the outcomes of security analysis into AutoFOCUS3, including the requirements required for implementation. To further augment SeCloud's user experience, Security Pattern Synthesis introduces wizards designed to enhance usability. These wizards incorporate a user-friendly GUI-based asset specification and solution visualization. The wizard for solution visualization

is equipped with a set of criteria to assist the user in determining the most suitable solution for the system.

In our previous article [6], we have performed experiments to evaluate the performance overhead caused by SeCloud while computing architecture solutions. The performance overhead depends on the number of (a) assets, (b) attack paths leading to threat scenarios, and (c) security architecture patterns. SeCloud implements constraints to achieve automatic search and optimization to reduce the number of solutions while preserving the most suitable solutions for the user's benefit. The implementation of such constraints resulted in a reduction of the performance overhead from 3600 s to 0.06, as showed in [6]. While this article does not include a performance evaluation of Security Pattern Synthesis, reserving this activity for future endeavors, the results presented in this section (considering two assets) were computed in a matter of seconds. It is crucial to note, however, that the computation duration may vary based on the number of assets, attack paths, and security architecture patterns taken into consideration, as mentioned earlier.

## Related Work

Securing cloud application is an important goal. As we have explained in Section "Introduction", in the cloud landscape one finds a large number of frameworks and components that support the implementation of secure cloud applications. The security analysis of application architectures remains a largely manual activity that can be supported by analysis tools. For example, tools like kube-bench<sup>5</sup> can analyse whether a cloud deployment follows certain guidelines for secure deployment. Our work in this article is intended to build on such frameworks and tools by supporting their selection in the architecture design phase.

In other contexts, especially for safety-critical applications in the automotive or aerospace sector, regulations make it necessary to follow a more structured approach to security. For example, the automotive security standard ISO 21434 defines requirements for the management of security in automotive applications. This includes a systematic threat analysis and risk assessment. There is work on supporting such activities with model-based approaches, e.g., [17]. There are commercial tools for support threat analysis and risk assessment, such as itemis SECURE.<sup>6</sup>

Our work in this article is based on work for automating security-related activities for automotive applications. SeCloud is based on SecPat, which as proposed in [18] for an automotive use case in the context of ISO 21434. SecPat has been developed using DLV [19]. The intruder model in

<sup>5</sup> <https://github.com/aquasecurity/kube-bench>

<sup>6</sup> <https://www.itemis.com/en/products/itemis-secure/>.

this article decomposes the intruder model of [14], which was tailored to service-oriented automotive architectures.

For cloud applications, there also exist tools to support threat analysis and risk management. One example is commercial tool ThreatGet.<sup>7</sup> ThreatGet allows the automatic identification of threat scenarios and attack paths. It provides suggestions for potential security measures to be selected by the user. ThreatGet does not instantiate the selected security measures in the system architecture. As a result, it might be unclear for the user to identify which components are relevant to the selected security measures. With SeCloud, it is possible to compute a selection of architecture options where all identified threats are mitigated by a suitable selection of security measures. SeCloud instantiates the recommended security measures by making explicit which components are part of the security measure (e.g., mTLS for components A and B).

Microsoft's Threat Analysis tool<sup>8</sup> is another example of a commercial tool to support the security analysis for cloud architectures. It is based on STRIDE and helps architects in identifying potential threats and in documenting their mitigation. While SeCloud is also based on STRIDE, it enables the automated recommendation of architecture options that address all identified threat scenarios. SeCloud flexible definition using ASP makes it possible to extend the method to more fine-grained threat scenario models than STRIDE.

The large number of possible design choices for cloud applications makes the automation of development tasks very desirable. For example, there is existing work on automating the setup of multi-cloud environments [20], or on the configuration of cloud systems [21]. The overview article [22] gives an overview of various approaches to securing cloud-based microservices, including various tools for analysing and securing them. SeCloud is intended as a tool to help users select combinations of the tools and services outlined in [22] and combine them into a coherent security architecture that meets the needs of their application. To achieve this, the list of patterns in SeCloud needs to be expanded to cover a wider range of framework, tools, and services.

## Conclusion

This article proposes Security Pattern Synthesis, a model-based system engineering plugin. The proposed plugin enables the automation of key TARA activities, including threat scenarios, attack paths, and the recommendation of security architecture patterns for mitigating threat scenarios.

<sup>7</sup> <https://www.threatget.com>.

<sup>8</sup> <https://www.microsoft.com/en-us/securityengineering/sdl/threatmodeling>.

SeCloud [6] serves as the backend for this proposed plugin. This article extends the description of SeCloud by providing a detailed explanation of its core functionality, which was left out in our previous article. Security Pattern Synthesis has been developed with the intention of reducing the effort required by security engineers while carrying out a threat analysis on cloud architectures.

**Acknowledgements** We thank the German Ministry for Economic Affairs and Climate Action of Germany for funding this work through the LuFo V-3 project RTAPHM.

## Declarations

**Conflict of interest** On behalf of all authors, the corresponding author states that there is no Conflict of interest.

## References

- Rose S, Borchert O, Mitchell S, Connelly S. Zero Trust Architecture. Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD 2020. <https://doi.org/10.6028/NIST.SP.800-207>
- Carroll M, Kotzé P, Merwe A. Secure cloud computing: Benefits, risks and controls. In: Venter, H.S., Coetzee, M., Loock, M. (eds.) Information Security South Africa Conference 2011, ISSA 2011. ISSA, Pretoria, South Africa 2011. <https://doi.org/10.1109/ISSA.2011.6027519>
- Eliseev V, Miliukova E, Kolpinskiy S. Neural network cryptographic obfuscation for trusted cloud computing. In: Integrated Models and Soft Computing in Artificial Intelligence, 2021;pp. 201–207
- Oleshchuk VA, Kjøien GM. Security and privacy in the cloud a long-term view. In: 2011 2nd International Conference on Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology (Wireless VITAE), 2011;pp. 1–5 . <https://doi.org/10.1109/WIRELESSVI TAE.2011.5940876>
- ISO/SAE AWI 21434: Road vehicles - cybersecurity engineering. 2021
- Dantas YG, Schöpp U. SeCloud: Computer-aided support for selecting security measures for cloud architectures. In: Proceedings of the 9th International Conference on Information Systems Security and Privacy, Lisbon, Portugal, February 22–24, 2023; pp. 264–275. SciTePress, Setúbal, Portugal (2023). <https://doi.org/10.5220/0011901900003405> .
- Egele M, Brumley D, Fratantonio Y, Kruegel C. An empirical study of cryptographic misuse in android applications. CCS '13, 2013;pp. 73–84. Association for Computing Machinery, New York, NY, USA . <https://doi.org/10.1145/2508859.2516693>
- Mainka C, Mladenov V, Schwenk J, Wich T. Sok: Single sign-on security - an evaluation of openid connect. In: 2017 IEEE European Symposium on Security and Privacy (EuroS&P), 2017;pp. 251–266 . <https://doi.org/10.1109/EuroSP.2017.32>
- Aravatinos V, Voss S, Teuffl S, Hölzl F, Schätz B. AutoFOCUS 3: Tooling concepts for seamless, model-based development of embedded systems. In: Proc. 8<sup>th</sup> Int. Workshop Model-based Architecting of Cyber-Physical and Embedded Systems (ACES-MB), 2015;pp. 19–26
- Potassco project: Clingo: A grounder and solver for logic programs. <https://github.com/potassco/clingo>

11. SeCloud: <https://drive.google.com/file/d/1a5UqihDly9lyL3MRjgzcY9jx-xhwoG2o> (2022)
12. fortiss GmbH: AutoFOCUS3 2.21. Available at <https://af3.fortiss.org/>
13. Gelfond M, Lifschitz V. Logic programs with classical negation. In: ICLP. 1990
14. Dantas YG, Barner S, Ke P, Nigam V, Schöpp U. Automating Vehicle SOA Threat Analysis Using a Model-Based Methodology. In: Proceedings of the 9th International Conference on Information Systems Security and Privacy, Lisbon, Portugal, February 22–24, 2023;pp. 180–191. SciTePress, Setúbal, Portugal (2023). <https://doi.org/10.5220/0011786400003405>
15. Shostack A. Threat Modeling: Designing for Security. John Wiley & Sons, Inc., New York, NY, USA 2014. <https://doi.org/10.5555/2829295>
16. EmbASP. Available at <https://www.mat.unical.it/calimeri/projects/embasp/>
17. Jungebloud T, Nguyen N, Kim D, Zimmermann A. Hierarchical model-based cybersecurity risk assessment during system design. In: 38th IFIP TC 11 International Conference, SEC 2023 (IFIPSEC) 2023. To appear.
18. Dantas YG, Nigam V. Automating safety and security co-design through semantically rich architecture patterns. ACM Trans Cyber Phys Syst. 2023;7(1):5–1528. <https://doi.org/10.1145/3565269>.
19. Leone N, Pfeifer G, Faber W, Eiter T, Gottlob G, Perri S, Scarcello F. The DLV system for knowledge representation and reasoning. ACM Trans. Comput. Log. 2006;7<https://doi.org/10.1145/1149114.1149117>
20. Sousa G, Rudametkin W, Duchien L. Automated setup of multi-cloud environments for microservices applications. In: 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), 2016;pp. 327–334 . <https://doi.org/10.1109/CLOUD.2016.0051>
21. Etedali A, Lung C.-H, Ajila S, Veselinovic I. Automated constraint-based multi-tenant SaaS configuration support using XML filtering techniques. In: 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), vol. 2, 2017;pp. 413–418 . <https://doi.org/10.1109/COMPSAC.2017.69>
22. Minna F, Massacci F. Sok: Run-time security for cloud microservices. Are we there yet? Computers & Security 127, 2023;103119 <https://doi.org/10.1016/j.cose.2023.103119>

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

## Chapter 9

# Knowledge Representation and Reasoning for Safety and Security Co-Analysis

Chapter 9 provides an improved version of the Domain-Specific Language (DSL) for specifying system architecture artifacts, safety and security artifacts, and safety and security architecture patterns. Reasoning rules have been specified to enable the automation of safety and security architecture patterns. Additionally, reasoning rules have been specified to enable the identification of synergies between safety and security artifacts, and safety and security consequences caused by architecture patterns. The DSL and reasoning rules have been implemented as a dedicated logic programming tool, which has been validated using an example taken from the automotive domain.

**Contributing article:** Yuri Gil Dantas and Vivek Nigam: Automating Safety and Security Co-design through Semantically Rich Architecture Patterns. ACM Transactions on Cyber-Physical Systems 2023: Volume 7, Issue 1, Pages: 1-28.

**Copyright information:** © 2023 Association for Computing Machinery. <https://doi.org/10.1145/3565269>.

**Author contributions:** The concept for the publication was jointly developed by Yuri Gil Dantas and Vivek Nigam. Yuri Gil Dantas performed a literature review to identify the main conditions used by safety and security engineers for recommending safety and security architecture patterns. The review also focused on understanding the potential consequences associated with the use of these patterns. The outcome of the literature review enabled the authors in specifying semantically-rich (safety and security) architecture patterns for automated driving systems. Safety-wise, this article is a successor of article [39], where large parts of the article have been improved. After several research discussions with Vivek Nigam, Yuri Gil Dantas implemented the tool and carried out the experiments presented in

the article. Yuri Gil Dantas took the lead in writing the initial draft of the article, excluding Section 2 that was written by Vivek Nigam. Vivek Nigam assisted Yuri Gil Dantas in improving the article. Yuri Gil Dantas handled subsequent revisions and corrections.



# Automating Safety and Security Co-design through Semantically Rich Architecture Patterns

YURI GIL DANTAS, fortiss GmbH, Germany

VIVEK NIGAM, Federal University of Paraíba, Brazil, and Huawei Technologies Düsseldorf GmbH, Germany

During the design of safety-critical systems, safety and security engineers make use of architecture patterns, such as Watchdog and Firewall, to address identified failures and threats. Often, however, the deployment of safety architecture patterns has consequences on security; e.g., the deployment of a safety architecture pattern may lead to new threats. The other way around may also be possible; i.e., the deployment of a security architecture pattern may lead to new failures. Safety and security co-design is, therefore, required to understand such consequences and tradeoffs in order to reach appropriate system designs. Currently, architecture pattern descriptions, including their consequences, are described using natural language. Therefore, their deployment in system design is carried out manually by experts and thus is time-consuming and prone to human error, especially given the high system complexity. We propose the use of semantically rich architecture patterns to enable automated support for safety and security co-design by using Knowledge Representation and Reasoning (KRR) methods. Based on our domain-specific language, we specify reasoning principles as logic specifications written as answer-set programs. KRR engines enable the automation of safety and security co-engineering activities, including the automated recommendation of which architecture patterns can address failures or threats, and consequences of deploying such patterns. We demonstrate our approach on an example taken from the ISO 21434 standard.

CCS Concepts: • **Computing methodologies** → *Knowledge representation and reasoning*; • **Computer systems organization** → *Architectures*; • **Theory of computation** → *Logic*; • **Security and privacy**;

Additional Key Words and Phrases: Safety architecture patterns, security architecture patterns, automation, safety and security co-design, automotive vehicle systems

## ACM Reference format:

Yuri Gil Dantas and Vivek Nigam. 2023. Automating Safety and Security Co-design through Semantically Rich Architecture Patterns. *ACM Trans. Cyber-Phys. Syst.* 7, 1, Article 5 (February 2023), 28 pages.

<https://doi.org/10.1145/3565269>

## 1 INTRODUCTION

Safety-critical systems are systems whose failure may result in severe consequences to human life, including death [21]. Examples of safety-critical systems are autonomous vehicles and aircraft

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 830892.

Authors' addresses: Y. Gil Dantas, fortiss GmbH, Munich, Germany; email: [dantas@fortiss.org](mailto:dantas@fortiss.org); V. Nigam, Federal University of Paraíba, João Pessoa, Brazil, and Huawei Technologies Düsseldorf GmbH, Düsseldorf, Germany; email: [vivek.nigam@huawei.com](mailto:vivek.nigam@huawei.com).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2023 Association for Computing Machinery.

2378-962X/2023/02-ART5 \$15.00

<https://doi.org/10.1145/3565269>

flight control. The challenge for engineers is to ensure that such systems are safe at all times by, e.g., providing protective measures to reduce the risk of failures to an acceptable level.

This challenge increases substantially with the interconnectivity of safety-critical systems. For example, vehicle platoons share information about their speed or position with other vehicles through wireless communication to enable vehicles to quickly react to sudden speed reductions. The system interconnectivity brings security to the development life cycle of safety-critical systems, as an intruder might cause catastrophic events by remotely disabling safety functions. Intruders may attack such communication channels to infiltrate vehicles and, e.g., disable safety functions, thus reducing passenger safety [34] or even causing accidents [10].

These types of attacks have served as motivation for the new ISO 21434 standard for Automotive Cyber-Security [20]. The standard also advocates a closer alignment between safety and security in order to ensure vehicle safety. That is, it advocates interactions between safety and security to coordinate the exchange of relevant information such as threat scenarios and hazard information or where a security requirement might conflict with a safety requirement. These interactions are part of *safety and security system co-design*, where tradeoffs between safety and security are well understood, and optimal system designs are reached.

During system design, safety and security engineers deploy architecture patterns, i.e., patterns that are known to provide some type of guarantee for safety, e.g., fault tolerance, and security, e.g., separation. Examples of safety architecture patterns are Watchdog and Monitor Actuator, and examples of security architecture patterns are Firewall and Security Monitor.

Currently, however, architecture patterns are documented in a rather informal fashion [1, 18, 29, 33] using natural language. Therefore, it is the job of the safety and security engineers to correctly understand the textual description of patterns and propose manually the use of a particular pattern at a particular location of the system architecture. As system complexity grows, this task becomes more complicated. This is because often patterns used for one aspect may have consequences for other aspects [27]. These consequences are context dependent. It may be that placing a pattern in one location of the system architecture may have serious consequences, while placing the same pattern in another location of the system architecture may not have these consequences. For example, placing a firewall at a communication channel with safety-relevant information may unintentionally block safety-critical flows, while placing a firewall at a communication channel without safety-relevant information does not have safety consequences. A second complicating factor is the correct understanding under which conditions a pattern can be used for attaining a safety goal or mitigating a security threat. For example, placing a safety pattern that adds heterogeneous redundancy, i.e., different implementations for the primary and secondary channels, instead of homogeneous redundancy, i.e., same implementation for both channels, propagates assumptions on the independence or not between primary and secondary channels. These assumptions need to be carefully understood during the development of the system.

Instead of describing architecture patterns informally using natural language, a better approach is to provide more precise semantics by using **Domain-specific Languages (DSLs)**.<sup>1</sup> *Semantically rich architecture pattern* descriptions enable increased automated support during system design. In our previous work [6], we demonstrate how system safety design can be automated by using semantically rich safety patterns encoded as *knowledge bases* [3], e.g., checking whether a safety pattern placed in the system architecture can be correctly used to attain a safety goal.

---

<sup>1</sup>The semantics provided by DSL is not formal semantics, i.e., set of traces, but rather lightweight semantics, i.e., defining a vocabulary for which the meaning is uniformly understood in the corresponding domain. For example, it is clear in the automotive domain what ECU and CAN buses are.



This article's main goal is to enable safety and security co-design automation by using semantically rich architecture patterns. Our main contributions are:

- **DSL:** We considerably extend our previous work [6] with a DSL for security and for safety and security co-design. Moreover, we improve our DSL for safety to, e.g., more precisely specify the intent of safety patterns.
- **Semantically Rich Safety and Security Patterns:** We propose a safety and security architecture pattern template that contains semantic information provided by the proposed DSL. Due to space restrictions, we describe in the article only four patterns, two for safety and two for security. Notice that our machinery (described below) currently supports 10 patterns [8]. The supported patterns are Acceptance Voting, Homogeneous Duplex, Heterogeneous Duplex, Monitor Actuator, Simplex Architecture, Triple Modular Redundancy, Watchdog, Firewall, and Security Monitor.
- **Reasoning Principles:** We extend the safety reasoning principles proposed in our previous work [6], increasing their precision by, e.g., considering when a safety architecture pattern fails operationally. We specify security reasoning principles based on formal intruder models, e.g., path reachability used to determine whether an attack may pose a threat to some sub-component from outside the system, and when a security architecture pattern can be used to mitigate some types of threats. We specify safety and security co-design reasoning rules, e.g., conditions for when a security architecture pattern may cause safety failures and when safety architecture patterns may be targeted by intruders to reduce the system safety.
- **Automated Reasoning:** We automate our machinery by using the off-the-shelf solver DLV [23]. It enables the automated safety and security co-design with patterns. We demonstrate this by using an example taken from the ISO 21434 [20]. We refer to the whole machinery proposed in this article as SAFSECPAT. SAFSECPAT is capable of automating several activities currently carried out manually by an expert. SAFSECPAT is publicly available in [8].

The remainder of this article is organized as follows: Section 2 reviews basic safety and security concepts, the V-model used in vehicle system development, and a template used for describing patterns. Section 3 describes the running example taken from ISO 21434. It also illustrates the main artifacts constructed during the execution of the V-model. Section 4 describes our DSL for safety, security, and safety and security co-analysis. Sections 5 and 6 demonstrate with some examples how to semantically enrich patterns using the proposed DSL. Section 7 demonstrates how safety and security co-design can be supported by semantically rich architecture patterns by automation through DLV. Finally, we conclude by pointing out related and future work in Sections 8 and 9.

## 2 SAFETY AND SECURITY CONCEPTS, V-MODEL, AND PATTERNS

The process, methods, and artifacts that shall be produced during the development of vehicle embedded systems are detailed in the standards ISO 26262 [19] for safety and ISO 21434 [9, 20] for security. The overall process follows the so-called V-model, shown in Figure 1.

Before we enter into the details of the process, we briefly review some basic concepts in safety and security to set the terminology used in the remainder of the article. For both safety and security, an *item* is the system or combination of systems to implement a function at the vehicle level.

*Basic Safety Concepts:* The definitions of the following safety concepts are taken in their great majority from [2]. A *hazard* is a situation that can cause harm to users or businesses. A *failure* is an event that when occurring results in a deviation of the expected behavior of a function. An *error* is a deviation of the expected system behavior. A *fault* is the hypothesized cause of an error. A **Minimal Cut Set (MCS)** is a set of failures that when occurring at the same time (or in sequence) may lead to a (top-level) failure. This top-level failure is often associated with a hazard

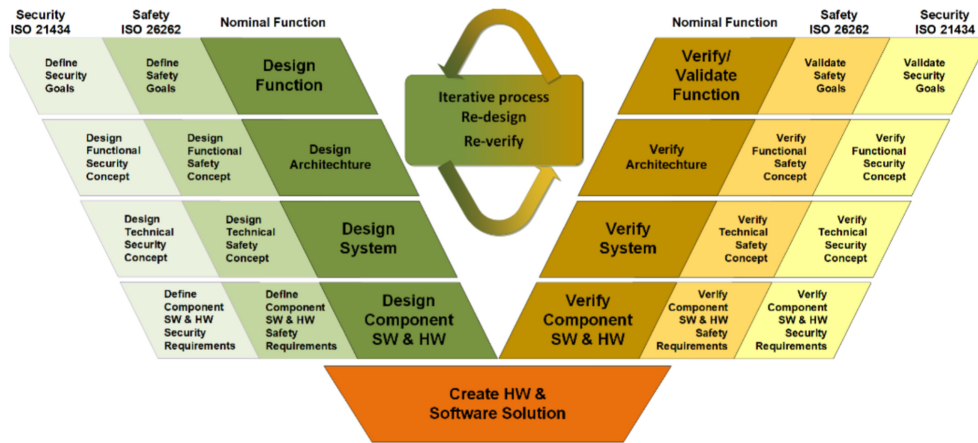


Fig. 1. Integrated V-model for system and safety [32].

using techniques such as **Hazard and Operability study (HAZOP)**. Normally, failures are associated with a set of predefined Guidewords that characterize intuitively the semantics of such failures. Examples of Guidewords are *loss* and *erroneous* that denote, respectively, a failure due to the loss of a function, i.e., a function not operating at all, and a failure due to an erroneous function behavior, e.g., a function not computing correctly some output value. The MCSs of a top-level failure are typically computed from a **Fault Tree Analysis (FTA)**, which is a deductive safety analysis method that decomposes failures using an and/or tree of sub-failures. *Fault tolerance* is a means to avoid service failures in the presence of faults. A *safety architecture pattern* (described in further detail in Section 2.1) is a system architecture solution that is known to provide some level of fault tolerance.

*Basic Security Concepts:* The definitions of the following security concepts are taken in their great majority from the ISO 21434 [20]. An *asset* is an object for which the compromise of its cybersecurity properties can lead to damage for an item's stakeholder. A *damage scenario* is an adverse consequence due to the compromise of a cybersecurity property of an asset. An *attack* is a deliberate action or interaction with the item or component or its environment that has potential to result in an adverse consequence. A *threat scenario* is a statement of potential negative actions that lead to a damage scenario. An *attack path* is a sequence<sup>2</sup> of actions that could lead to the realization of a threat scenario. A *cybersecurity property* is an attribute of an asset including confidentiality, integrity, and availability. A *cybersecurity risk* is the effect of uncertainty on road vehicle security expressed in terms of *attack feasibility* and *impact*. A *security architecture pattern* is a system architecture solution that is known to control security risks by mitigating threat scenarios.

At the top left of the V-model shown in Figure 1, one defines the item that will be the subject of safety and security analysis. The item provides details such as the preliminary architecture and operational conditions. Then safety and security analysis is performed in order to define safety and security goals (top left boxes in the left branch of the V-model depicted in Figure 1). Safety analyses, such as **Hazard Analysis and Risk Assessment (HARA)**, FTA, and HAZOP, identify losses, hazards, failures leading to hazards, and faults that may trigger such failures. Security analysis, such as **Threat Agent Risk Assessment (TARA)**, STRIDE,<sup>3</sup> Attack Trees [31], and path

<sup>2</sup>The ISO 21434 defines a path as a set and not as a sequence. We will use it here as a sequence.

<sup>3</sup>The threats considered by STRIDE are Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege.

analysis [20], identify key assets and their corresponding threats. Security risks are determined by assessing the attack feasibility and impact of each identified attack path in the system leading to a threat scenario. Typically, an attack path feasibility is evaluated by using different factors such as the time needed to carry out an attack and the knowledge/tooling needed by the intruder. Moreover, different categories for impact may be considered, such as financial, privacy, and safety. Safety-related impacts have the highest impact for safety-critical systems.

Based on the risk evaluation of such hazards and threats, functional safety and security concepts are formulated for the item by establishing safety goals/key risks on the system architecture [19] (second boxes in the left branch of the V-model depicted in Figure 1).

The safety functional concept establishes the level of criticality required by elements in the system. For safety, vehicle functions are assigned values between ASIL QM, A, B, C, and D, where QM has no safety criticality, while A, B, C, and D have ascending criticality requirements. Safety goals may also assign the level of tolerance to faults for functions as defined below:

- **Fail silent** is a more strict type of fault tolerance when compared to fail active (function fails without any measure) as the failure of the fail-silent function shall necessarily lead to the loss of the functionality (e.g., shut down the function). Thus, it shall not be possible that the faults of a fail-silent function, e.g., incorrect computations, are propagated within the system.
- **Fail safe** adds the requirement to fail silent in that if a function fails, then it shall necessarily switch to a safe state. *Safe state* is a state without significant harm to users or businesses. Moreover, as we will describe below with the architecture patterns, it is possible to effectively detect when a function is no longer functioning and switch to a safe state and trigger appropriate measures, e.g., inform the driver.
- **Fail operational** is the strongest type of fault tolerance as it requires that a function operate with the same level of safety even after facing a specific number of faults. For example, a lane-keeping function typically shall operate complying to the requirements of the highest level of criticality, i.e., ASIL D, after at least one fault occurred.

The security goals establish the properties, e.g., confidentiality, integrity, and availability, that need to be satisfied by the identified assets in the system architecture.

## 2.1 Safety and Security Architecture Patterns

Once the functional safety and security concepts are completed, the technical safety and security concepts are developed (third boxes in the left branch of the V-model depicted in Figure 1). This is done by further establishing safety and security requirements on the item system architecture, to comply, for example, with the level of safety criticality established and security properties required. Typically, safety and security engineers make use of architecture solutions called architecture patterns.

*Architecture patterns* are abstract solutions to recurrent system problems such as safety and security.<sup>4</sup> For example, safety engineers make use of a Triple Modular Redundancy to address failures (both erroneous and losses), thus avoiding hazards. Similarly, security engineers use firewalls to isolate the system architecture, thus reducing security risks. These architecture patterns are described in an abstract form and they are implementation agnostic. The description of architecture patterns makes them easier to understand and can be seen as guidelines for the design of the system architecture. It is not the part of architecture patterns to define exactly how pattern components shall be implemented, although requirements might be provided. The actual implementation of pattern components such as monitors shall be tailored to specific functions. For example,

<sup>4</sup>Other measures like testing and established coding practices may be used in addition to or instead of architecture patterns.

Table 1. Architecture Pattern Description Template

| Field                      | Description   |
|----------------------------|---|
| Pattern name               | Name of this pattern  |
| Structure                  | Block diagram of this pattern   |
| Intent                     | Textual description of the purpose of this pattern  |
| Problem addressed          | Textual description of the problem the pattern addresses  |
| Assumptions (requirements) | Assumptions necessary for using this pattern  |
| Consequences               | Textual description of the consequences to other concerns, e.g., security, performance, reliability |

monitors are often implemented using plausibility checks (tailored to specific functions). A collection of known patterns is available in the ISO 26262 [19] as well as in the literature [1, 29].

Since patterns are commonly used and many times different names are used, pattern templates have been proposed [1, 29, 33] as a means to uniformly describe a pattern. An example of a template is depicted in Table 1, which is similar to pattern templates appearing in the literature [1, 33]. The description of the pattern contains a name for which a pattern is known; its structure, typically shown as a block diagram; its intent, i.e., the purpose for which this pattern is normally used, e.g., to enable fail-operational level of fault tolerance; problem addressed, e.g., to control erroneous functions or loss of functions; assumptions (a.k.a. requirements) required to use this pattern, e.g., different types of implementations for the primary and redundant channels; consequences of using this pattern; and other concerns, such as security, performance, reliability, and costs.

### 3 RUNNING EXAMPLE

This section describes an example from the automotive domain, namely headlamp system, taken from the ISO 21434 standard [20]. We use this example to illustrate the concepts and process reviewed in Section 2, as well as to illustrate the machinery introduced in the next sections. Following the process described in Section 2, we start by providing a description of both the item, i.e., headlamp system, and the results of a safety analysis of the headlamp system.

*Functionalities and System Architecture.* A headlamp system is responsible for switching on/off the headlamp of a vehicle. The headlamp system has two specific features, namely high-beam light and low-beam light. The driver can turn on or off the headlamp and switch between high beam and low beam from the steering wheel. Since high beam may affect the visibility of drivers incoming from the opposite direction, it is a safety recommendation that a vehicle's headlamp is switched to low beam whenever an oncoming vehicle is approaching from the opposite direction. A proposed solution is to use a sensor, e.g., a camera, to detect approaching vehicles. If the headlamp is in high-beam mode, the headlamp system switches the headlamp automatically to low-beam mode when an oncoming vehicle is detected. It returns automatically the headlamp to high-beam mode if the oncoming vehicle is no longer detected [20].

Figure 2 depicts both the logical and the platform (a.k.a. hardware) architecture of the headlamp system, as well as the deployment table of the logical architecture to the platform architecture. The boxes in the logical architecture represent components, e.g.,  $P_{WRSWT}$ ,  $BdCTL$ , and the boxes in the platform architecture represent hardware units, e.g., Interface 1, CAN Bus 2. The black and white circles connected to components/hardware units are, respectively, output and input ports. The arrows connected to ports represent unidirectional channels between components/hardware units.

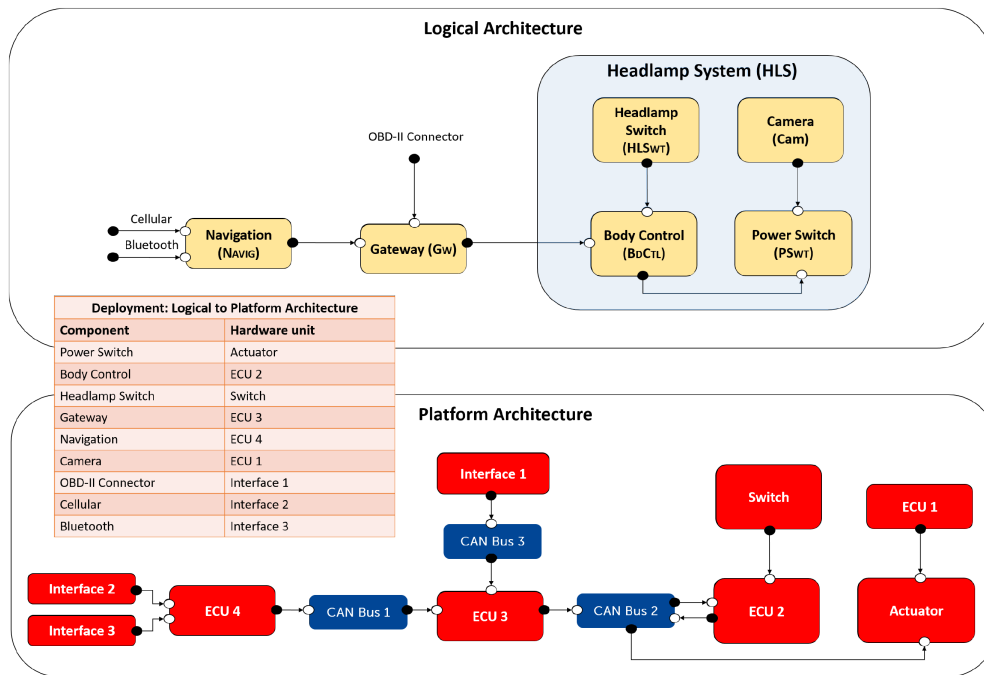


Fig. 2. Architecture of the headlamp system.

A Camera (CAM) detects oncoming vehicles and sends signals to the Power Switch (P<sub>WRSWT</sub>). The Body Control (B<sub>DCTL</sub>) sends signals to P<sub>WRSWT</sub>. These signals are requests from the driver (coming from the Headlamp Switch, H<sub>LSWT</sub>) to turn the headlamp on or off. Note that the channel from B<sub>DCTL</sub> to P<sub>WRSWT</sub> is deployed to a CAN bus (CAN Bus 2) in the platform architecture. The left-hand side of the logical architecture depicts external components that may access the headlamp system. There is a Gateway (G<sub>W</sub>) that controls access from other components located outside the headlamp system, e.g., Navigation (NAVIG). G<sub>W</sub> receives signals from an OBD-II Connector (OBDC<sub>onn</sub>). NAVIG has two interfaces, namely Cellular (CELL) and Bluetooth (BT). Both CELL and BT interfaces and OBDC<sub>onn</sub> may access B<sub>DCTL</sub> to, e.g., carry out software updates. Note that in the platform architecture the channel from NAVIG to G<sub>W</sub> is deployed to CAN Bus 1, the channel from G<sub>W</sub> to B<sub>DCTL</sub> is deployed to CAN Bus 2, and the channel from OBDC<sub>onn</sub> to G<sub>W</sub> is deployed to CAN Bus 3.

*Safety Analysis Results.* We now focus on the safety of the headlamp system. While not claiming to be comprehensive, we describe potential hazards, faults, and failures that can be identified from a safety analysis. We also describe safety goals that shall be met to address the identified hazards.

Table 2 describes the identified hazards for the headlamp system. The ASIL level of a hazard is assigned based on three parameters: Severity, Exposure, and Controllability [19]. *Severity* denotes the consequences to the life of the user of the system in the presence of a failure that leads to the hazard. *Exposure* denotes the possibility of the system being in a hazardous situation that can cause harm. *Controllability* denotes the extent to which the user of the system can control the system in the presence of a failure that leads to the hazard. For the sake of example, we assign ASIL C to **HZ1**: We consider the severity as life-threatening injuries (S3), the exposure as medium probability of happening (E3), and the controllability as difficult to control (C3). We assign ASIL A to **HZ2**: We consider the severity as light and moderate injuries (S1), the exposure as high probability of happening (E4), and the controllability as normally controllable (C2).

We list below the identified faults and failures that may lead to the presence of hazards **HZ1** and **HZ2**. Specifically, we consider failures of type erroneous and loss.

Table 2. Identified Hazard for the Headlamp System

| Hazard | Description   | ASIL |
|--------|---|------|
| HZ1    | Headlamp turns off unintentionally during night driving.              | C    |
| HZ2    | Unintended low beam of headlamp when no oncoming vehicle is detected. | A    |

Table 3. Safety Goals to Prevent the Presence of Identified Hazards

| Safety Goal | Description   | ASIL | Hazard |
|-------------|---|------|--------|
| SG1         | The system shall fail operational always after the 1st erroneous failure of the Body Control. The system shall transition to a safe state always after the 2nd failure of type erroneous. | C    | HZ1    |
| SG2         | The system shall fail silent always after most 1st erroneous failures on the Camera.  | A    | HZ2    |

- **FT1:** The Body Control is faulty, thus leading to not turning the headlamp on upon the driver's request. This may happen if fault **FT1** triggers a failure **FL1** of type erroneous. The failure **FL1** may lead to hazard **HZ1**.
- **FT2:** The logical channel between the Body Control and the Power Switch is faulty, leading to not turning the headlamp on upon the driver's request. This may happen if fault **FT2** triggers a failure **FL2** of type loss. The failure **FL2** may lead to hazard **HZ1**.
- **FT3:** The Camera is faulty, not providing the expected information to the Power Switch to enable the high-beam light. This may happen if fault **FT3** triggers a failure of type erroneous. The failure **FL3** may lead to hazard **HZ2**.

Since all of these failures can lead independently to the hazard, the minimal cut sets are  $\{\mathbf{FL1}\}$ ,  $\{\mathbf{FL2}\}$ , and  $\{\mathbf{FL3}\}$ .

Table 3 describes a safety goal to address **HZ1**. Notice that safety goals are often expressed as a negation of a hazard. Here, we consider a more specific safety goal to enable the reasoning of safety patterns. The safety goal **SG1** aims at avoiding potential harm always after the first failure of type erroneous. The system shall transition to a safe state always after the second failure. If achieved, this safety goal improves both the safety and availability of the headlamp system. **SG1** can be achieved by the implementation of safety patterns, implementing, e.g., fault tolerance tactics. **SG2** describes a safety goal that allows the system to fail silent due the low criticality of hazard **H2**. Note that we neglect fault **FT2**, as we consider that this fault is unlikely to happen.

*Security Analysis Results.* For demonstrating system safety, one shall argue that the defined safety goals are met. Since the headlamp system is safety critical, it is considered an asset.

The next step is to determine the threats to the headlamp system. Failures shall immediately be considered as threats as recommended by the ISO 26262 [19] and the ISO 21434 [20].<sup>5</sup> For example, one shall evaluate the risk of attacking the Body Control to cause it to fail. Notice that the intruder can also cause the CAN to fail. So, although from a safety perspective a CAN failure is very rare, from a security perspective a CAN denial-of-service attack may be carried out provided

<sup>5</sup>Notice, however, that security shall also consider threats that are not directly safety related, such as threat posed to privacy. Since our focus is on safety, we do not focus on these types of threats in this article.

the intruder can access that CAN. Section 7.2 demonstrates how the association of failures and threats can be derived by reasoning rules.

Once the threats are identified, one carries out a risk analysis. This is done by enumerating the attack paths leading to threats to the headlamp. To compute the attack paths, one identifies which are the platform architecture elements from which the intruder may access the system. We classify such elements as *public*. Consider the platform architecture of the headlamp system. We consider the following hardware units as public elements: Interface 1 (OBD-II Connector), Interface 2 (Cellular), and Interface 3 (Bluetooth). That is, an attack may access each of these hardware units to carry out attacks against the headlamp system. The exact attack path depends on the threat model considered. In Section 7.3, we consider a threat model based on the Dolev-Yao intruder [12] used in protocol security, where the intruder attempts logical attacks to the identified assets. The attack paths are enumerated based on this threat model.

Finally, security requirements with security countermeasures, e.g., security patterns, are proposed to mitigate the identified high-risk threats.

#### 4 KNOWLEDGE BASES FOR SAFETY AND SECURITY SYSTEM ARCHITECTURE

Our goal is to provide mechanisms to support the automated hardening of system architectures using safety and security patterns. To this end, we propose the use of **Knowledge Representation and Reasoning (KRR)** methods [3] (revisited in Section 4.1). KRR enables the specification of sophisticated reasoning principles that can be automated by reasoner tools such as DLV and clingo.

We have recently proposed the use of KRR for hardening system architectures using safety patterns [6]. The main outcome of this work was SAFPAT. SAFPAT consists of a DSL for embedded systems and safety reasoning principles for some selected safety patterns specified as disjunctive logic programs. Finally, we demonstrated how SAFPAT can recommend safety patterns in an automated fashion.

This article substantially extends the previously developed SAFPAT. More concretely, we extend SAFPAT [6] in the following ways:

- We extend our DSL to specify minimal cut sets, faults, failures, and safety goals, thus following more closely the process described in Section 2.
- We extend our DSL to more precisely specify safety patterns, including when a safety pattern enables a system to fail operational, fail safe, and fail silent, and which safety patterns are suitable for addressing life-critical (ASIL C and D) and low-critical (ASIL A and B) hazards.
- The main extension of SAFPAT is the introduction of security aspects to enable the automated recommendation of security patterns. Since SAFPAT now includes security aspects (in addition to safety aspects), we changed its name from SAFPAT to SAFSECPAT.
- Another important extension is the introduction of security consequences when applying safety patterns, and safety consequences when applying security patterns.
- We specify reasoning principles to specify assumptions required to use patterns.
- We specify constraints to limit the number of architecture solutions with patterns recommended by SAFSECPAT.

The focus of this section is on the DSL of SAFSECPAT. Sections 5 and 6 describe, respectively, our specification of safety and security patterns by example. Section 7 describes our reasoning principles that are automated by DLV. Next, we provide a brief overview on Knowledge Representation and Reasoning to help readers grasp the developed SAFSECPAT.

##### 4.1 Knowledge Bases and Answer-set Programming

KRR [3] is a mature field of Artificial Intelligence based on logic-based methods to represent and reason about knowledge bases. A *knowledge base* is a declarative representation of the

world/system. The declarative nature of knowledge bases enables the programming of reasoning rules by using existing logic programming engines such as DLV [23].

A disjunctive logic program  $M$  is a set of rules of the form  $a_1 \vee \dots \vee a_m \leftarrow \ell_1, \dots, \ell_n$  or where  $\ell, \ell_1, \dots, \ell_n$  are literals, that is, atomic formulas,  $a$ , or negated atomic formulas, *not*  $a$ . The interpretation of the default negation *not* assumes a *closed-world* assumption. That is, we assume to be true only the facts that are explicitly supported by a rule.

The semantics of a disjunctive logic program  $M$  is based on the stable model semantics [17]. We illustrate the semantics of logic programs with an example.<sup>6</sup>

Consider the program  $P_1$  with the following two rules:

$$a \vee b \quad c \leftarrow a.$$

$P_1$  has two answer-sets  $\{a, c\}$  and  $\{b\}$ . Intuitively, each answer-set is a minimal model of the logic program that makes each rule of the program to be true. Moreover, if a rule's head is empty, i.e.,  $m = 0$  from the set of rules above, then it is a constraint. For example, if we add the clause  $\leftarrow b$  to  $P_1$ , then the resulting program has only one answer-set  $\{a, c\}$ .

DLV [23] is an engine implementing disjunctive logic programs based on ASP semantics [17]. In the remainder of this article, we use the DLV notation writing  $:-$  for  $\leftarrow$  and  $\vee$  for  $\vee$ . For example, the program  $P_1$  is written as  $a \vee b$  and  $c :- a$ . As with DLV, capital letters  $X, Y, Z$  are variables that during execution are instantiated by appropriate terms, and minuscule letters  $a, b, c$  are constants. Variables or constants surrounded by  $[]$  are lists. The  $_$  (underscore) character specifies that the argument can be ignored in the current rule.

## 4.2 A Domain-specific Language for Embedded Systems

SAFSECPAT consists of a DSL for embedded systems. This DSL enables the specification of architecture elements, safety and security artifacts, and architecture patterns. The architecture elements and safety and security artifacts are specified by the user of SAFSECPAT, while the reasoning rules in Section 7 are under the hood. We illustrate our DSL using the headlamp system described in Section 3.

**4.2.1 Architecture Elements.** Table 4 describes selected architecture elements specified in our DSL, including components, sub-components, channels, and information flows.

*Example 4.1.* Consider the architecture of the headlamp system described in Section 3. A user may specify the logical architecture of the headlamp system as follows:

```
cp(cam). cp(bdCtl). cp(ps). cp(hlSwt). cp(hls). cp(gw). cp(navig). cp(obdC). cp(cell).
cp(bt). subcp(cam,hls). subcp(bdCtl,hls). subcp(ps,hls). subcp(hlSwt,hls).
ch(cmps_a,cam,ps). ch(hsbd,hlSwt,bdCtl). ch(bcps,bdCtl,ps). ch(gwbc,gw,bdCtl).
ch(navgw,navig,gw). ch(obdgw,obdC,gw). ch(ceInav,cell,navig). ch(btnav,bt,navig).
if(if1,[cmps]). if(if2,[hsbd,bcps]). if(if3,[obdgw,gwbc,bcps]).
if(if4,[ceInav,navgw,gwbc,bcps]). if(if5,[btnav,navgw,gwbc,bcps]).
```

The facts  $cp(hlSwt)$ ,  $cp(bdCtl)$ ,  $cp(ps)$ , and  $cp(hls)$  denote, respectively, the Headlamp Switch, the Body Control, the Power Switch, and the Headlamp system components. The fact  $subcp(bdCtl,hls)$  denotes that the Body Control is a sub-component of the Headlamp system. The fact  $ch(bcps_a,bdCtl,ps)$  denotes the logical communication channel  $bcps_a$  between the Body Control and the Power Switch. The information flow  $if2$  denotes data flows from channel  $hsbd$  to channel  $bcps$ .

<sup>6</sup>We refer to [3] for the precise formal semantics of logic programs.



Table 4. SAFSECPAT: Language for (Selected) Architecture Elements

| Fact  | Description  |
|---|--|
| <code>cp(id)</code>                                     | <code>id</code> is a component in the system.  |
| <code>subcp(id<sub>1</sub>,id<sub>2</sub>)</code>       | <code>id<sub>1</sub></code> is a sub-component of component <code>id<sub>2</sub></code> .  |
| <code>ch(id,id<sub>1</sub>,id<sub>2</sub>)</code>       | <code>id</code> is a logical channel connecting an output of component <code>id<sub>1</sub></code> to an input of component <code>id<sub>2</sub></code> . Notice that it denotes a unidirectional connection.  |
| <code>if(id,[ch<sub>1</sub>,...,ch<sub>n</sub>])</code> | <code>id</code> is an information flow following the channels in <code>[ch<sub>1</sub>,...,ch<sub>n</sub>]</code> .  |
| <code>ecu(id)</code>                                    | <code>id</code> is an Electronic Computing Unit (ECU) that can run components.   |
| <code>can(id)</code>                                    | <code>id</code> is a Controller Area Network (CAN) to communicate between ECUs.  |
| <code>interface(id)</code>                              | <code>id</code> is a hardware interface that allows the connection of external peripheral to components.   |
| <code>dep(id,id<sub>d</sub>)</code>                     | Component <code>id</code> is deployed (i.e., executed) in ECU <code>id<sub>d</sub></code> or in interface <code>id<sub>d</sub></code> , or alternatively, the logical channel <code>id</code> is deployed in CAN <code>id<sub>d</sub></code> to establish the communication between <code>id</code> 's components. |

A user may specify the hardware units of the platform architecture as follows. We only consider the hardware units shown in Figure 2, omitting, e.g., the communication medium between the Body Control and the Power Switch.

```
ecu(ecu1). ecu(ecu2). ecu(ecu3). ecu(ecu4). can(can1). can(can2). wireless(wl).
interface(int2). actuator(act). interface(int1). interface(int3). can(can3).switch(swt).
```

The fact `ecu(ecu1)` denotes the ECU `ecu1`. The facts `switch(swt)` and `actuator(act)` denote the switch `swt` and the actuator `act`, respectively. The fact `interface(int1)` denotes interface `int1`. The fact `wireless(wl)` denotes a wireless communication apparatus identified as `wl`.

The deployment of the logical architecture to the platform architecture is specified as follows:

```
dep(cam,ecu1) dep(bdCt1,ecu2). dep(gw,ecu3). dep(navig,ecu4). dep(navgw,can1).
dep(gwbc,can2). dep(bcps,can2). dep(obdC,can3). dep(cell,int1). dep(bt,int2).
dep(obdC,int3). dep(ceInav,wl). dep(btnav,wl).
```

The fact `dep(bdCt1,ecu2)` denotes that the Body Control is deployed to ECU `ecu2`. The fact `dep(gwbc,can2)` denotes that the channel from the Gateway to the Body Control is deployed to CAN bus `can2`. The fact `dep(btnav,wl)` denotes that the data transmission of logical channel `btnav` is performed via wireless.

**4.2.2 Safety and Security Artifacts.** Our DSL consists of safety and security artifacts described in Table 5. By *safety artifacts*, we refer to safety goals, hazards, faults, failures, and minimal cut sets. By *security artifacts*, we refer to potential threat and threat scenarios.<sup>7</sup> A potential threat associated with a hardware unit **HWCP** becomes a threat if there is a path **P** from a public element to **HWCP**. Table 5 also includes further elements (e.g., `ft2fl`, and `reachI`) needed to reason about safety and security as described in Section 7.

Motivated by [18], we consider safety goals that require the system to either fail operational, fail silent, or fail safe. These safety goals have great impact on the precision of SAFSECPAT in

<sup>7</sup>Our DSL enables the specification of further security artifacts such as damage scenario and risk determination as shown in [9]. We omit these security artifacts for the sake of presentation given that our focus is on safety and security co-design using architecture patterns.

Table 5. SAFSECPAT: Language for Safety and Security Artifacts

| Fact   | Description   |
|--|---|
| $hz(id_{hz}, [sys, hz_{sev}, hz_{exp}, hz_{ctl}])$           | $id_{hz}$ is a hazard for system $sys$ of severity $hz_{sev}$ , exposure $hz_{exp}$ , and controllability $hz_{ctl}$ .  |
| $ft(id_{ft}, [id_{cp}])$                                     | $id_{ft}$ is a fault associated with component $id_{cp}$ .  |
| $fl(id_{fl}, [fl_{tp}])$                                     | $id_{fl}$ is a failure of type $fl_{tp}$ , where $fl_{tp} \in \{err, loss\}$ .  |
| $ft2fl(id_{ft}, id_{fl})$                                    | $id_{ft}$ is a fault that triggers failure $id_{fl}$ .  |
| $mcs(id_{mcs}, [id_{fl}])$                                   | $id_{mcs}$ is a minimal cut set consisting of failure(s) $id_{fl}$ .  |
| $lmcs2hz([id_{mcs}], id_{hz})$                               | $id_{mcs}$ is a list of minimal cut sets that leads to hazard $id_{hz}$ .   |
| $sg(id_{sg}, [id_{hz}, f_{op}, f_{sl}, f_{sf}])$             | $id_{sg}$ is a safety goal to address hazard $id_{hz}$ . This safety goal requires the system to either fail operational ( $f_{op}$ ), fail silent ( $f_{sl}$ ), or fail safe ( $f_{sf}$ ) in the presence of $id_{hz}$ .                     |
| $public(id_{hw})$  | $id_{cp}$ is an HW unit that may be accessible by external users.   |
| $pThreat(id_{pt}, [id_{cp}, id_{hw}, pt_{tp}, pt_{sv}])$     | $id_{pt}$ is a potential threat associated with component $id_{cp}$ and HW unit $id_{hw}$ . $id_{pt}$ is of type $pt_{tp}$ , where $pt_{tp} \in \{con, int, avl\}$ , and of severity $pt_{sv}$ , where $pt_{sv} \in \{neg, maj, mod, sev\}$ . |
| $reachl(id_{cp}, id_{hw}, P)$                                | component $id_{cp}$ and HW unit $id_{hw}$ may be reached by an intruder through path $P$ in the technical architecture.   |
| $threat([id_{th}, P], [id_{cp}, id_{hw}, th_{tp}, th_{sv}])$ | $id_{th}$ is a threat associated with component $id_{cp}$ and HW unit $id_{hw}$ that may be reached through path $P$ . $id_{th}$ is of type $th_{tp}$ and of severity $th_{sv}$ (both as for $pThreat$ ).                                     |

recommending safety patterns, as, e.g., only a subset of safety patterns can ensure that the system fails operational. Whether the system fails operational, silent, or safe is specified as parameters of the predicate  $sg$  (see Table 5). Either of these parameters ( $f_{op}$ ,  $f_{sl}$ ,  $f_{sf}$ ) may be assigned to the following constants:

$$f_{op}, f_{sl}, f_{sf} \in \{allXfail, mostXfail, never\}, \text{ where}$$

$allXfail$  denotes always after  $X$  failures have been detected, where  $X$  is an integer. For example, the system shall fail operational always after the first failure has been detected. According to [18] and [25], safety patterns that implement plausibility checks (e.g., monitor-actuator pattern [1]) can only detect about 95% of the failures. Hence, we consider the constant  $mostXfail$  that denotes always after most  $X$  failures have been detected, where  $X$  is an integer. For example, the system shall fail safe after most first failures. This means that some of the failures will not be detected by the pattern and the system will not always fail safe. This is an important distinction between  $allXfail$  and  $mostXfail$ . In order to detect 100% of the failures, one shall consider robust patterns such as the dual self-checking pair pattern [18]. The constant  $never$  denotes that the system shall never fail operational, fail silent, or fail safe; e.g., the system shall never fail silent when a failure is detected.

*Example 4.2.* Consider the results of the safety analysis described in Section 3. A user may specify the safety analysis results in our DSL as follows:

```

hz(hz1,[h1s,s3,e4,c3]). hz(hz2,[h1s,s3,e3,c2]). ft(ft1,[bdCt1]). fl(f11,[err]).
sg(sg1,[hz1,all1fail,never,all2fail]). sg(sg2,[hz2,never,most1fail,never]).
ft(ft2,[bcps]). ft2f1(ft1,f11). fl(f12,[loss]). ft2f1(ft2,f12). ft(ft3,[cam]).
fl(f13,[err]). ft2f1(ft3,f13). mcs(mcs1,[f11]). mcs(mcs2,[f12]).
lmcs2hz([mcs1,mcs2],hz1). mcs(mcs3,[f13]). lmcs2hz([mcs3],hz2).

```

The facts  $hz(hz1, [h1s, s3, e4, c3])$  and  $hz(hz2, [h1s, s3, e3, c2])$  denote, respectively, the hazards **HZ1** and **HZ2** identified in Section 3. Consider hazard  $hz1$ . The safety goal for addressing hazard  $hz1$  is specified by the fact  $sg(sg1, [hz1, all1fail, never, all2fail])$ . The faults that trigger failures leading to  $hz1$  are specified as  $ft(ft1, [bdCt1])$  and  $ft(ft2, [bcps])$ . These faults are associated, respectively, to the Body Control and to the logical channel between the Body Control and the Power Switch. The fault  $ft1$  triggers failure  $fl(f11, [err])$  of type erroneous, and the fault  $ft2$  triggers failure  $fl(f12, [loss])$  of type loss. The minimal cut set  $mcs1$  consists of failure  $f11$  and  $mcs2$  consists of failure  $f12$ . Either minimal cut set  $mcs1$  or  $mcs2$  leads to hazard  $hz1$  (as specified by the fact  $lmcs2hz([mcs1, mcs2], hz1)$ ).

Security-wise, we expect the user (e.g., a security engineer) to provide all public elements as input to SAFSECPAT. The considered public elements for the headlamp system are shown in the example below. We, however, do not expect a user to provide (potential) threats as input to SAFSECPAT, even though it is completely possible to do so. Instead, we derive (potential) threats from identified faults and failures. The association of faults/failures and (potential) threats may be derived by reasoning rules, as demonstrated in Section 6.

*Example 4.3.* Consider the platform architecture of the headlamp system illustrated in Figure 2. Specified in our DSL, we consider the following hardware units as public:

```
public(int1). public(int2). public(int3).
```

These facts denote, respectively, the Interfaces  $int1$ ,  $int2$ , and  $int3$ .

**4.2.3 Safety and Security Architecture Patterns.** Our DSL enables the specification of safety patterns for addressing failures, and security patterns for mitigating threats. Table 6 describes the predicates for instantiating a pattern and for specifying the intent of a pattern. The former represents the necessary components (e.g., the faulty component for safety) and channels for the pattern. The latter represents the intent of the pattern, including for which type of failure or threat the pattern is suitable to be applied.

Sections 5 and 6 describe how to declaratively specify safety and security patterns by example using the predicates `safetyPattern`, `safetyIntent`, `securityPattern`, and `securityIntent` (see Table 6).

## 5 SPECIFICATION OF SAFETY ARCHITECTURE PATTERNS

This section illustrates how we can use SAFSECPAT to provide semantically rich description of safety architecture patterns that will enable the automated reasoning described in Section 7. In particular, we instantiate the pattern template described in Section 2 with two safety patterns. For each instantiation, we provide a high-level description of the pattern and its specification in SAFSECPAT. The pattern template includes pattern assumptions and security consequences from applying safety patterns. The assumptions described in this section are not meant to be comprehensive.

### 5.1 Dual Self-checking Pair with Fail Safe

The dual self-checking pair pattern [18] with fail safe consists of two pairs and a fault detector for each pair. Each pair consists of a primary and a secondary component that are identical and operate in parallel. The primary and secondary components from the second pair are developed with a

Table 6. SAFSECPAT: Language for Safety and Security Architecture Patterns

| Fact  | Description  |
|---|--|
| <code>safetyPattern(id,[name, [cp], [inp],[int],[out]])</code>  | name is a safety pattern of ID id. This pattern consists of a list of components (e.g., redundant components) cp. The last three parameters inp, int, and out denote, respectively, the input and the internal, and the output channels related to the pattern.  |
| <code>safetyIntent(name,[[fl<sub>tp</sub>], asil, f<sub>op</sub>,f<sub>sl</sub>,f<sub>sf</sub>])</code> | name is a safety pattern suitable for avoiding failures of type fl <sub>tp</sub> , where fl <sub>tp</sub> ∈ {err,loss}. The ASIL asil denotes to which ASIL the pattern is suitable to be applied, where asil ∈ {a,b,c,d}. Pattern name ensures the system to either fail operational (f <sub>op</sub> ), fail silent (f <sub>sl</sub> ), or fail safe (f <sub>sf</sub> ). |
| <code>securityPattern(id,[name, [cp],[inp],[out]])</code>   | name is a security pattern of ID id. This pattern consists of a list of components cp. The last three parameters inp, int, and out denote, respectively, the input, the internal, and the output channels related to the pattern.  |
| <code>securityIntent(name,[[th<sub>tp</sub>]])</code>   | name is a security pattern suitable for mitigating threats of type th <sub>tp</sub> , where th <sub>tp</sub> ∈ {con, int, avl}.  |

different design implementation in comparison to the components from the first pair. The computations from each pair are sent to their respective fault detector. While no failure is detected, the actuator receives the computations from the first pair. The fault detector requires exact agreement from the computations (i.e., identical output values). When there is no exact agreement between the computations from the first pair, the fault detector of the first pair sends a take-over signal to the fault detector of the second pair. This signal means that the computations from the second pair will be considered, and they will be sent to the actuator if the components produce identical output values. If a failure is also detected on the second pair, the fault detector transitions the system to a safe state. This pattern fails operational always after the first failure of type erroneous (i.e., failure on the first pair). It fails safe always after the second failure is detected (i.e., failure on the second pair). This pattern never fails silent. The instantiation of the dual self-checking pair pattern with fail safe is shown in Table 7.

We describe the specification of the dual self-checking pair pattern with fail safe in SAFSECPAT. Consider the language for safety patterns described in Table 6 and the structure of the pattern illustrated in Table 7. This pattern is instantiated as follows:

```
safetyPattern(idpat, [dualSelfCheckingPairFS, [pr1, se1, fd1, pr2, se2, fd2],
[inp1, inp2, inp3, inp4], [int1, int2, int3, int4, int5], [out1, out2, fs]]).
```

The safety intent of the dual self-checking pair pattern with fail safe is specified as follows:

```
safetyIntent(dualSelfCheckingPairFS, [[err], d, all1fail, never, all2fail]).
```

Consider the assumptions for the dual self-checking pair pattern from Table 7. SAFSECPAT creates these assumptions whenever the dual self-checking pair pattern is instantiated. The first assumption in Table 7 is specified as follows in SAFSECPAT:

```
assumption(dualSelfCheckingPairFS, are_independent, [pr1, se1, pr2, se2])
:- safetyPattern(idpat, [dualSelfCheckingPairFS, [pr1, se1, _, pr2, se2, _], _, _, _]).
```

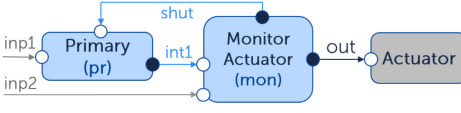
Table 7. Dual Self-checking Pair Pattern with Fail Safe

|                         | Description   | SAFSECPAT Specification  |
|-------------------------|---|--|
| Pattern name            | Dual self-checking pair pattern with fail safe  | NAME=dualSelfCheckingPairFS;   |
| Structure               |   | COMPONENT=[pr1, se2, fd1, pr2, se2, fd2];<br>INPUT_CH=[inp1, inp2, inp3, inp4];<br>INTERNAL_CH=[int1, int2, int3, int4, int5];<br>OUTPUT_CH=[out1, out2, fs];  |
| Intent                  | This pattern is suitable for high criticality hazards (ASIL C and D). This pattern fails operational always after the 1st erroneous failure, and it transitions the system to a safe state always after the 2nd failure has been detected. This pattern never fails silent.                                     | TYPE_FAIL=[err];<br>ASIL=d;<br>FAIL_OP=all1fail;<br>FAIL_SILENT=never;<br>FAIL_SAFE=all2fail;  |
| Problem addressed       | This pattern tolerates faults by avoiding failures of type erroneous.   |  |
| Assumptions             | The first and second pair shall be implemented using independent designs.<br>The the first and second pair shall be allocated to dedicated hardware units.<br>The fault detector shall be verified with respect to its correctness.   | TYPE_ASSUMPTION=are_independent;<br>COMPONENT=[pr1, se1, pr2, se2];<br>TYPE_ASSUMPTION=are_decoupled;<br>COMPONENT=[pr1, se1, pr2, se2];<br>TYPE_ASSUMPTION=are_verified_wrt_correctness;<br>COMPONENT=[fd1, fd2]; |
| Consequences (security) | There is a potential threat associated with the fault detectors. That is, as an intruder may carry out malicious actions to prevent the fault detectors from properly functioning. This potential threat is of type <b>integrity</b> if the failure associated with the primary component is of type erroneous. | COMPONENT=[fd1, fd2];<br>TYPE_THREAT=int;  |

## 5.2 Monitor-actuator Pattern

The monitor-actuator pattern [1] consists of a primary component and a monitor. The monitor consumes both the computations (i.e., outputs) from the primary component and the primary's original inputs such that the monitor can cross-check their computations to identify failures of type erroneous. While no failure is detected by the monitor (e.g., through the use of plausibility checks), the actuator receives the outputs from the primary component. If the monitor detects a failure on the primary component, the monitor initiates a corrective action by sending a shutdown signal to the primary component. That is, this pattern fails silent always after most first failures of type erroneous have been detected on the primary component. It neither fails operational nor fails safe. The instantiation of the monitor-actuator pattern is shown in Table 8.

Table 8. Monitor-actuator Pattern

|                         | Description   | SAFSECPAT Specification   |
|-------------------------|---|---|
| Pattern name            | Monitor-Actuator Pattern  | NAME=monitorActuator;   |
| Structure               |    | COMPONENT=[pr, mon];<br>INPUT_CH=[inp1, inp2];<br>INTERNAL_CH=[int1, shut];<br>OUTPUT_CH=[out]; |
| Intent                  | This pattern is suitable for low criticality hazards (ASIL A and B). This pattern fails silent always after most 1st failures. It never fails operational and it never fails safe.  | TYPE_FAIL=[err];<br>ASIL=b;<br>FAIL_OP=never;<br>FAIL_SILENT=all1fail;<br>FAIL_SAFE=never;      |
| Problem addressed       | This pattern tolerates faults by avoiding failures of type erroneous.   |   |
| Assumptions             | The monitor shall be verified with respect to its correctness.  | TYPE_ASSUMPTION=<br>are_verified_wrt_correctness;<br>COMPONENT=[mon];                           |
| Consequences (security) | There is a potential threat associated with the monitor, as an intruder may carry out malicious actions to prevent the monitor from properly functioning. This potential threat is of type integrity if the failure associated with the primary component is of type erroneous. | COMPONENT=[mon];<br>TYPE_THREAT=int;  |

We describe the specification of the monitor-actuator pattern in SAFSECPAT. Consider the language for safety patterns described in Table 6 and the structure of the pattern illustrated in Table 8. This pattern is instantiated as follows:

```
safetyPattern(idpat, [monitorActuator, [pr, mon], [inp1, inp2], [int1, shut], [out]]).
```

The safety intent of the monitor-actuator pattern is specified as follows:

```
safetyIntent(monitorActuator, [[err], b, never, most1fail, never]).
```

We specify the assumption for the monitor-actuator pattern from Table 8 as follows. SAFSECPAT creates this assumption whenever the monitor-actuator pattern is instantiated.

```
assumption(monitorActuator, are_verified_wrt_correctness, [mon])
:- safetyPattern(idpat, [monitorActuator, [_ , mon], [_ , _ , _]).
```

## 6 SPECIFICATION OF SECURITY ARCHITECTURE PATTERNS

This section illustrates how we can use SAFSECPAT to provide semantically rich description of security architecture patterns that will enable the automated reasoning described in Section 7. We instantiate the pattern template described in Section 2 with two security patterns. For each instantiation, we provide a high-level description of the pattern and its specification in SAFSECPAT. The pattern template includes pattern assumptions and safety consequences from applying the security pattern. The assumptions are not meant to be comprehensive.

Table 9. Firewall Pattern

|                       | Description   | SAFSECPAT Specification   |
|-----------------------|---|---|
| Pattern name          | Firewall  | NAME=firewall;  |
| Structure             |   | COMPONENT=[bus, fw, pr];<br>INPUT_CH=[inp1, inp2];<br>OUTPUT_CH=[out1, out2];   |
| Intent                | This pattern intercepts, filters, and blocks incoming and outgoing messages in the network layer.   | TYPE_THREAT=[avl, int];   |
| Problem addressed     | This pattern mitigates threats that violates the availability and integrity of the system.  |   |
| Assumptions           | Firewall shall be verified with respect to its correctness.<br><hr/> Security policies shall be specified.  | TYPE_ASSUMPTION=are_verified_wrt_correctness;<br>COMPONENT=[fw];<br><hr/> TYPE_ASSUMPTION=have_policies;<br>COMPONENT=[fw]; |
| Consequences (safety) | The deployed firewall might be faulty, e.g., it might erroneously block legit messages. Thus, there is a new fault associated with the deployed firewall that may trigger erroneous failures. | COMPONENT=[fw];<br>TYPE_FAILURE=err;  |

## 6.1 Firewall Pattern

The firewall pattern [33] is instantiated in Table 9. A firewall is placed between a bus (e.g., a CAN bus) and a hardware unit (e.g., a gateway). The bus receives and sends messages from the external and internal network, respectively. These messages are intercepted and analyzed by the firewall. The firewall mitigates threats of type availability and integrity. That is, the firewall controls the network access to the internal network according to predefined security policies (e.g., blacklisting IP addresses consuming more bandwidth than a given threshold) and can also inspect message content to detect intrusion attempts and anomalies [33].

We describe the specification of the firewall pattern in SAFSECPAT. Consider the language for security patterns described in Table 6 and the structure of the pattern illustrated in Table 9. The firewall pattern is instantiated as follows:

```
securityPattern(idpat, [firewall, [bus, pr, fw], [inp1, inp2], _, [out1, out2]]).
```

The security intent of the firewall pattern is specified as follows:

```
securityIntent(firewall, [[avl, int]]).
```

The first assumption for the firewall pattern from Table 9 is specified as follows:

```
assumption(firewall, are_verified_wrt_correctness, [fw])
:- securityPattern(idpat, [firewall, [_, _, fw], _, _, _]).
```

## 6.2 Security Monitor Pattern

The security monitor pattern is instantiated in Table 10. This pattern mitigates threats that violate the integrity of the system. We consider security monitors that mitigate such threats by monitoring

Table 10. Security Monitor Pattern

|                       | Description  | SAFSECPAT Specification   |
|-----------------------|--|---|
| Pattern name          | Security Monitor   | NAME=securityMonitor;   |
| Structure             |  | COMPONENT=[pr, mon];<br>INPUT_CH=[int1, int2];<br>INTERNAL_CH=[int1, shut];<br>OUTPUT_CH=[out];                         |
| Intent                | This pattern intercepts, filters, and blocks incoming and outgoing messages from components.   | TYPE_THREAT=[int];  |
| Problem addressed     | This pattern mitigates threats that violates the integrity of the system.  |   |
| Assumptions           | Firewall shall be verified with respect to its correctness.<br>Security policies shall be specified.   | TYPE_ASSUMPTION=are_verified_wrt_correctness;<br>COMPONENT=[mon];<br>TYPE_ASSUMPTION=have_policies;<br>COMPONENT=[mon]; |
| Consequences (safety) | The deployed monitor might be faulty, e.g., it might erroneously block legit messages. Therefore, there is a new fault associated with the deployed monitor that may trigger erroneous failures. | COMPONENT=[mon];<br>TYPE_FAILURE=err;   |

incoming and outgoing messages from components and enforcing security policies in the application layer. Whenever a security policy is violated, the monitor can initiate a corrective action by sending a shutdown signal to the component [16].

We describe the specification of the security monitor pattern in SAFSECPAT. Consider the language for security patterns described in Table 6 and the structure of the pattern illustrated in Table 10. The security monitor pattern is instantiated as follows:

```
securityPattern(idpat, [securityMonitor, [pr, mon], [inp1, inp2], [int1, shut], [out]]).
```

The security intent of the security monitor pattern is specified as follows:

```
securityIntent(securityMonitor, [[int]]).
```

The second assumption for the security monitor described in Table 10 is specified as follows:

```
assumption(securityMonitor, have_policies, [mon])
:- securityPattern(idpat, [securityMonitor, [_, mon], [_, _], _]).
```

## 7 SAFETY AND SECURITY REASONING PRINCIPLES

This section describes selected safety and security reasoning principles that can be automated by using solvers, such as DLV [23].

### 7.1 Safety Reasoning

Building on top of [6], we specify as logic programs safety reasoning principles to determine when (1) a failure can be avoided, (2) a minimal cut set can be avoided, (3) a fault can be tolerated, (4) a hazard can be controlled, and (5) a safety goal can be satisfied. We introduce five new facts to



specify safety reasoning principles for (1), (2), (3), (4), and (5). Note that four of the new facts receive attributes from the safety pattern intent as an argument in order to make explicit how they have been addressed by the pattern. These attributes consist of ASIL values (e.g., a), fail-operational values (e.g., all1fail), fail-silent values (e.g., all2fail), and fail-safe values (e.g., never).

- `avoided(IDFL,ATTRSINTENT)` denotes that failure IDFL is avoided with a safety pattern intent ATTRSINTENT (e.g., avoided by a pattern that fails silent always after the first failure).
- `avoidedMCS(IDMCS,ATTRSINTENT)` denotes that minimal cut set IDMCS is avoided with a safety pattern intent ATTRSINTENT.
- `tol(IDFT,ATTRSINTENT)` denotes that fault IDFT is tolerated with a safety pattern intent ATTRSINTENT.
- `ctl(IDHZ,ATTRSINTENT)` denotes that hazard IDHZ is controlled with a safety pattern intent ATTRSINTENT.
- `satisfied(IDSG)` denotes that safety goal IDSG is satisfied.

Specified by the next rule, a failure is avoided if a pattern is associated to the faulty component TARGET, and the pattern is able to avoid failures of type TYPE checked by `#member(TYPE,PATTYPE)`.

```
avoided(IDFL,[IDPAT | ATTRSINTENT]) :- f1(IDFL,[TYPE]),
    ft(IDFT,[TARGET]), ft2f1(IDFT,IDFL), #member(TYPE,PATTYPE),
    getSafPatTarget(IDPAT,TARGET), safetyIntent(PAT,[PATTYPE | ATTRSINTENT]),
    safetyPattern(IDPAT,[PAT | ATTRSPAT]).
```

A minimal cut set IDMCS is avoided if at least one failure of its set has been avoided. A fault is tolerated if the failures triggered by that fault are avoided. The rules for both `tol` and for `avoidedMCS` are omitted here. Next, we specify a rule for hazard controllability.

```
ctl(IDHZ,ATTRSINTENT) :- hz(IDHZ,ATTRSHZ),
    lmc2shz(LMCS,IDHZ), getMinIntent(LMCS,ATTRSINTENT).
```

A hazard is controlled if each MCS in the list of LMCS is avoided. This is checked by the fact `getMinIntent(LMCS,ATTRSINTENT)`. In addition, `getMinIntent(LMCS,ATTRSINTENT)` returns the minimal attributes needed for controlling hazard IDHZ (see example below). This is relevant to show the minimal attributes (taken from the pattern intent) required to control the hazard.

*Example 7.1.* Consider two failures FLA and FLB that lead to hazard HZA. Safety pattern SPA avoids failure FLA with the intent attributes `[c,all1fail,never,all2fail]`, and safety pattern SPB avoids failure FLB with the intent attributes `[d,all1fail,never,never]`. The fact `getMinIntent(LMCS,ATTRSINTENT)` will return the minimal attributes for controlling hazard HZA, i.e., `[c,all1fail,never,never]`.

The next rule denotes when a safety goal is satisfied. A safety goal IDSG is satisfied if hazard IDHZ is controlled with higher or equal attributes than the ones required by the safety goal (similarly to Example 7.1). These checks are done by the fact `checkHigherOrEqual(IDHZ,IDSG)`.

```
satisfiedSG(IDSG) :- sg(IDSG,[IDHZ | ATTRSSG]),
    getHazardASIL(IDHZ,ASIL), ctl(IDHZ,ATTRSCTL), checkHigherOrEqual(IDHZ,IDSG).
```

**7.1.1 Recommendation of Safety Architecture Patterns.** We introduce our reasoning rule for recommending which safety patterns may be used at which place of the system architecture to avoid failures provided as input information by the user. This is specified by the following rule:

```
xsafetyPattern([nuIDSAFPAT,PAT,CTR],[PAT,[TARGET,[nuRED,CTR],[nuCKR,CTR]],
[nuINP,CTR],[nuINT,CTR],[nuOUT,CTR]]) v
nxsafetyPattern([nuIDSAFPAT,PAT,CTR],[PAT,[TARGET,[nuRED,CTR],[nuCKR,CTR]],
```

```
[nuINP,CTR],[nuINT,CTR],[nuOUT,CTR]])
:- f1(IDFL,[FLTYPE]), ft(IDFT,[TARGET]), ft2f1(IDFT,IDFL),
   exploreSafPat(PAT), safetyIntent(PAT,[PATFLTYPE | ATTRSINTENT]),
   #member(FLTYPE,PATFLTYPE), getSafIntentASIL(PAT,PATASIL), mcs(IDMCS,FAILURES),
   #member(IDFL,FAILURES), lmcshz(IDMCS,IDHZ), asil(IDHZ,HZASIL),
   higherEqualThan(PATASIL,HZASIL), counterSafPat(CTR).
```

It specifies the recommendation (`xsafetyPattern`) or not (`xnsafetyPattern`) of a safety pattern. Specifically, this rule specifies that a safety pattern is recommended to avoid a failure IDFL of type FLTYPE triggered by fault IDFT associated to component TARGET that leads to hazard IDHZ if the safety pattern is suitable for both avoiding FLTYPE as checked by `#member(TYPE,PATTYPE)` and addressing the ASIL of IDHZ as checked by `higherEqualThan(PATASIL,HZASIL)`. PATTYPE is taken from the pattern intent.

Note that the prefix “x” in front of `safetyPattern` is to make explicit that the safety pattern has been automatically recommended by SAFSECPAT. Omitted here, we have a rule for mapping `xsafetyPattern` to `safetyPattern`. The fact `exploreSafPat(PAT)` denotes that SAFSECPAT shall explore whether the safety pattern PAT is suitable to avoid a given failure. Which patterns shall be considered by SAFSECPAT is provided by the user. The fact `counterSafPat(CTR)` denotes a counter CTR to ensure that each safety pattern has a unique ID. The constants starting with `nu` do not appear in the baseline architecture. Whenever a safety pattern is recommended, SAFSECPAT ensures that both the components and channels related to the recommended pattern are created. These components and channels are prefixed with `nu` so that one can easily identify the increments in the architecture modified by SAFSECPAT.

As the system complexity grows and with it the number of failures and locations where a safety pattern can be placed, the number of pattern recommendations may rapidly increase. To keep the number of models manageable, we use DLV constraints to *not consider models* where, e.g., the same instance of a pattern is recommended more than once and more than one suitable pattern is recommended to avoid a given failure (i.e., to avoid that two distinct patterns avoid the same failure).

**7.1.2 Safety Architecture Pattern Recommendation for the Headlamp System.** We apply SAFSECPAT to the headlamp system described in Section 3 to automate the recommendation of which safety architecture patterns could be used to avoid the identified failures. We run our recommendation machinery explained in the section above for a number of safety patterns, including the dual self-checking pair pattern, the heterogeneous duplex pattern [1], the monitor-actuator pattern, and the watchdog pattern [1].

We consider the defined safety goals SG1 and SG2 to address hazards HZ1 and HZ2, respectively. The goal is to provide safety patterns suitable for (1) avoiding the failures leading to hazards HZ1 and HZ2 and (2) satisfying the safety goals. Figure 3 illustrates one architecture solution provided by SAFSECPAT that achieves this goal. Note that Figure 3 omits the external components from the headlamp system due to the lack of space.

SAFSECPAT recommended the dual self-checking pair pattern for avoiding erroneous failures (i.e., failure FL1) on the Body Control. This pattern satisfies the safety goal SG1, as it fails operational always after the first failure, and it fails safe always after the second failures. SAFSECPAT recommended the monitor-actuator pattern for avoiding erroneous failures (i.e., failure FL3) on the Camera. This pattern satisfies the safety goal SG2, as it fails silent always after the first failure. The pattern assumptions generated by SAFSECPAT can help engineers to deploy the pattern-related components into the platform architecture.

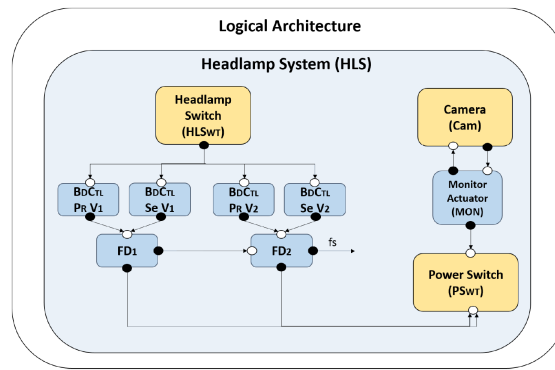


Fig. 3. Headlamp system with patterns.

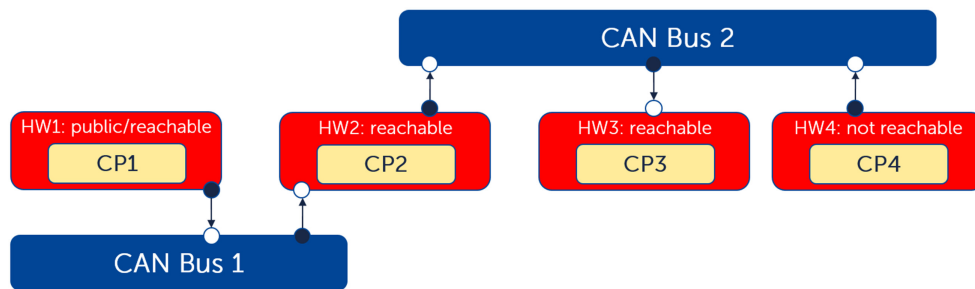


Fig. 4. Illustration of the intruder reachability.

## 7.2 Security Reasoning

This article proposes the use of KRR for security with architecture patterns. The goal is to provide automated methods for automating the recommendation of security architecture patterns to mitigate threats. As a basis to achieve this goal, we specify security reasoning principles to determine when (1) a potential threat becomes a threat and (2) a threat can be mitigated by a security pattern.

We introduce our intruder model before introducing these reasoning principles.

**7.2.1 Threat Model.** We assume a threat model inspired by the traditional Dolev-Yao intruder model [12] widely used for security protocol verification. Intuitively, the DY intruder is the most powerful symbolic intruder. She can have access and manipulate any information to which she has access, i.e., information that appears in a channel reachable from a public interface that is not encrypted or encrypted with a key that possesses the decryption key.

More precisely, our intruder model has the following capabilities inspired by the Dolev-Yao model for CAN bus communication channels:

- **Base Case:** The intruder can reach any public hardware/interface.
- **Inductive Case 1:** If the intruder can reach hardware HW and there is a component CP deployed in HW that writes on a CAN bus CAN, then the intruder can also reach CAN.
- **Inductive Case 2:** If the intruder can reach the CAN Bus CAN and there is a component CP deployed in a hardware HW that reads from the CAN, then the intruder can reach HW.

For example, consider the topology depicted in Figure 4. Assume that the hardware unit HW1 is public; e.g., it has a wireless interface. Therefore, the intruder can reach to HW1. Furthermore, assume that a component, CP1, deployed in HW1 writes to CAN bus CAN Bus 1; then from the inductive case 1, the intruder can reach CAN Bus 1; i.e., she can (in principle) write into CAN Bus 1. Furthermore, assume that a component CP2 deployed in hardware HW2 reads from CAN Bus 2; then from inductive case 2, the intruder can also reach HW2. Similarly, since there is a component in HW2

that writes into CAN Bus 2 and a component in HW3 that reads from CAN Bus 2, the intruder can reach CAN Bus 3 and HW3. However, since there is no component in HW4 that reads from CAN Bus 2 but possibly only writes into CAN Bus 2, the intruder cannot reach HW4.

We can program/customize this intruder model as logic programming. For example, the intruder model described above is specified by a number of rules in SAFSECPAT; the rules for the **Base Case** and **Inductive Case 1** are, respectively, shown below, while the case for **Inductive Case 2** follows similarly and is elided:

```
reachI(CP,HWCP,[HWCP]) :- public(CP), dep(CP,HWCP). % Base Case
reachI(CH,CAN,[CAN|PATH]) :- writesToCan(CP,CAN), dep(CH,CAN),
    dep(CP,HWCP), reachI(CP,HWCP,PATH), not #member(CAN,PATH). % Inductive Case 1
```

**Remarks:** Notice that while we are inspired by the Dolev-Yao intruder model, there is a key difference. Since we are not taking into account the contents of the exchanged messages, the intruder model shown above does not take into account the fact whether the messages are encrypted nor does it take into account how the exchanged messages are actually used. This means that the reachability to components and communication channels is an over-approximation, which may lead to false positives. To make the analysis more precise, one could include more information about the messages exchanged, e.g., whether a message is encrypted. This is, however, left out of the scope of this article as it deals with different phases of development.

We also notice that the threat model could be further refined by considering vulnerabilities as done in [28]. This would mean the extension of our DSL with vulnerabilities and would, in principle, enable a more refined analysis of the possible attacks. Indeed, we believe that it is possible to specify logic programs encoding the rules described in [28]. We leave this exercise to future work.

**7.2.2 Security for Safety.** To demonstrate the safety of the system, security engineers shall ensure that intruders cannot trigger identified failures. Hence, we specified security reasoning principles for deriving potential threats from identified failures.

Inspired by [13, 14, 20], we derive a potential threat from a failure based on the type and severity of the hazard led by the failure. A failure of type `err` that leads to a hazard of severity `S3` is mapped to a potential threat (`pThreat`) of type `int` (integrity) and of severity `sev`. A failure of type `loss` that leads to a hazard of severity `S2` is mapped to a `pThreat` of type `avl` (availability) and of severity `maj`. A failure of type `err` that leads to a hazard of severity `S1` is mapped to a `pThreat` of type `int` and of severity `mod`. Currently, we are not considering a mapping from a failure type to confidentiality. The mapping from failure to potential threat is specified by the next rule:

```
pThreat(IDFL,[TARGET,HWTARGET,SECTYPE,SECSEV]) :- f1(IDFL,[SAFTYPE]), ft(IDFT,[TARGET]),
    ft2f1(IDFT,IDFL), dep(TARGET,HWTARGET), typeMap(SAFTYPE,SECTYPE),
    hz(IDHZ,[_,SAFSEV,_,_]), mcs(IDMCS,FLISTFAIL), #member(IDFL,FLISTFAIL),
    lmcs2hz(LIDMCS,IDHZ), #member(IDMCS,LIDMCS), severityMap(SAFSEV,SECSEV).
```

Specified by the following rule, a potential threat becomes a threat if the hardware unit **HW-TARGET** can be reached through a path `PATH` (as described in the threat model).

```
threat([IDPT,PATH],[TARGET,HWTARGET,SECTYPE,SECSEV]) :-
    pThreat(IDPT,[TARGET,HWTARGET,SECTYPE,SECSEV]), reachI(TARGET,HWTARGET,PATH).
```

*Example 7.2.* Consider the failure `f11` and hazard `hz1` identified in Section 3. Failure `f11` that leads to hazard `hz1` is mapped to the potential threat `pt1`. The potential threats become a threat as `ecu2` can be reached by three paths, including the path from the Bluetooth (deployed into interface `int3`) to the Body Control (deployed into ECU `ecu2`):

```

    pThreat(pt1,[bdCt1,ecu2,err,sev]).
    threat([pt1,[ecu2,can2,ecu3,can1,ecu4,int3],[bdCt1,ecu2,err,sev]).

```

The intruder path [ecu2, can2, ecu3, can1, ecu4, int3] can be read from right to left.

We introduce one new fact, namely `mit(IDTH)`, for when a threat IDTH is mitigated. We omit the rule for `mit` here. In a nutshell, a threat IDTH is mitigated if a suitable security pattern is placed in the architecture for mitigating the type of threat violated by IDTH.

**7.2.3 Recommendation of Security Architecture Patterns.** We introduce our reasoning rule for recommending which security patterns may be used at which place of the system architecture to mitigate threats provided as input information by the user or derived by safety artifacts. Our rules for recommending security patterns are tailored to the pattern. For example, the firewall and the security monitor patterns are applied to distinct places in the system architecture. That is, the firewall is placed between a CAN bus and a hardware unit, and the security monitor is placed to an individual hardware unit.<sup>8</sup> The next rule specifies the placement (`xsecurityPattern`) or not (`nxsecurityPattern`) of the firewall pattern:

```

xsecurityPattern([nuIDSECPAT,firewall,CTR],[firewall,[HWUNIT,COMM,[nuCKR,CTR]],
[nuINP,CTR],[nuINT,CTR],[nuOUT,CTR]]) v
nxsecurityPattern([nuIDSECPAT,firewall,CTR],[firewall,[HWUNIT,COMM,[nuCKR,CTR]],
[nuINP,CTR],[nuINT,CTR],[nuOUT,CTR]])
:- threat(IDTR,ATTRSTR), getThreatType(IDTR,TRTYPE), getThreatTarget(IDTR,TARGET),
getThreatPath(IDTR,PATH), can(COMM), hw(HWUNIT),#subList([HWUNIT,COMM],PATH),
exploreSecPat(firewall), securityIntent(firewall,ATTRSINTENT),counterSecPat(CTR),
getSecIntentThreatType(firewall,PATTRTYPE), #member(TRTYPE,PATTRTYPE).

```

This rule specifies that the firewall pattern is recommended to mitigate a threat IDTR exploited through path PATH if a firewall is placed between a can COMM and a hardware unit HWUNIT, where both COMM and HWUNIT are in PATH as checked by `#subList([HWUNIT,COMM],PATH)`. The firewall shall be able to mitigate the type of threat violated by IDT as checked by `#member(TRTYPE,PATTRTYPE)`. We leave to future work the use of severity of threats as a condition to recommend security patterns.

**7.2.4 Security Architecture Pattern Recommendation for the Headlamp System.** We now apply SAFSECPAT to the headlamp system described in Section 3 to automate the recommendation of which security architecture patterns could be used to mitigate the identified threats. We run our recommendation machinery for the firewall pattern and the security monitor pattern.

We consider the six threats derived from the identified failures on the headlamp system. These threats target either the ECU `ecu2` (i.e., Body Control) or the CAN `can2` (i.e., logical communication between the Body Control and the Power Switch) from the public elements, that is, `int1` (i.e., OBD-II C.), `int2` (i.e., Cellular), `int3` (i.e., Bluetooth). Two of these threats are shown below:

```

threat([pt1,[ecu2,can2,ecu3,can1,ecu4,int3],[bdCt1,ecu2,int,sev]).
threat([pt2,[can2,ecu3,can3,int1],[bcps,can2,av1,sev]).

```

Our goal is to provide security patterns suitable for mitigating these threats that violate the availability (`av1`) and the integrity (`int`) of the headlamp system. Figure 5 illustrates one architecture solution provided by SAFSECPAT that achieves this goal. SAFSECPAT recommended the firewall

<sup>8</sup>Security patterns for automotive is an ongoing research topic [5]. In principle, we can also specify reasoning principles for security patterns that ensure confidentiality such as the symmetric encryption pattern. For patterns that require encryption SAFSECPAT would not make any visible changes in the system architecture. Instead, SAFSECPAT would provide security requirements such as “Channel X shall be encrypted.”

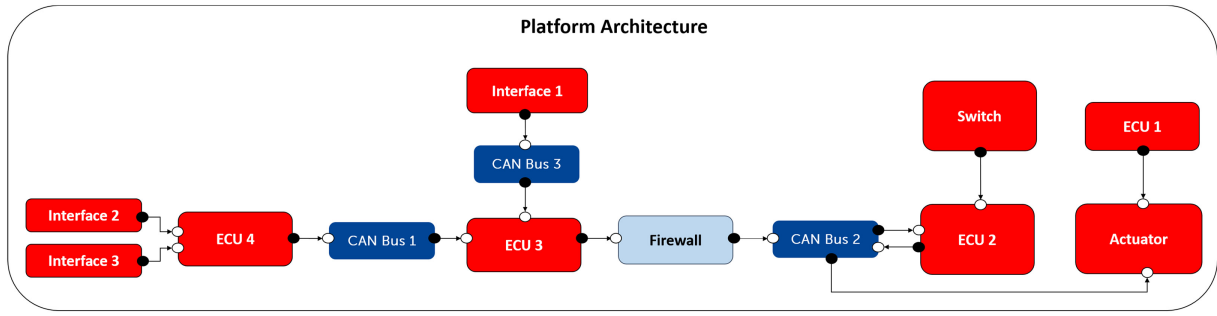


Fig. 5. Headlamp system with security architecture pattern (Firewall).

pattern for mitigating all derived threats. The firewall is placed between `ecu3` and `can2` so that it can intercept, filter, and block incoming messages (possibly malicious) from public elements.

### 7.3 Safety and Security Co-analysis Reasoning

This article proposes the use of KRR for safety and security co-analysis with architecture patterns. The goal is to provide automated methods to reason about the consequences of safety architecture patterns to security, and of security architecture patterns to safety.

**7.3.1 Security Consequences Caused by Safety Architecture Patterns.** We specified reasoning principles for determining when a security pattern can cause consequences to security. The deployment of a safety pattern may lead to a new (potential) threat to the system, as an intruder may perform malicious actions to prevent the deployed safety pattern from properly functioning (e.g., not avoiding failures). The following reasoning rule specifies this consequence:

```
pThreat(IDPAT, [CP, CHECKER, SECTYPE, SECSEV]) :- safetyPattern(IDPAT, ATTRSPAT),
  getSafPatChecker(IDPAT, CHECKER), getSafPatTarget(IDPAT, TARGET), ft(IDFT, [TARGET]),
  f1(IDFL, [FAILTYPE]), ft2f1(IDFT, IDFL), dep(CP, CHECKER), lmc2hz(LIDMCS, IDHZ),
  typeMap(FAILTYPE, SECTYPE), hz(IDHZ, [_, SAFSEV, _, _]), mcs(IDMCS, FLISTFAIL),
  #member(IDFL, FLISTFAIL), #member(IDMCS, LIDMCS), severityMap(SAFSEV, SECSEV).
```

There is a new potential threat `IDPAT` (same id of the safety pattern) associated with the `CHECKER` of the safety pattern (e.g., a fault detector) if `CHECKER` is monitoring a faulty component `TARGET`. This potential threat becomes an actual threat if `CHECKER` can be reached by an intruder.

**7.3.2 Security Consequences on the Headlamp System.** Consider the headlamp system with safety patterns illustrated in Figure 3. The deployment of a monitor-actuator to tolerate faults on the Camera leads to a new potential threat. This potential threat, however, does not lead to a threat since an intruder cannot reach the monitor from a public element. The deployment of the dual self-checking pair pattern to tolerate faults on the Body Control leads to a new threat since the ECU `ecu2` (i.e., Body Control) reads from the CAN `can2`. This threat can be, in principle, mitigated by the same firewall (illustrated in Figure 5) deployed between `ecu3` and `can2`.

**7.3.3 Safety Consequences Caused by Security Architecture Patterns.** We specified reasoning principles for determining when a security pattern can cause consequences to safety. The deployment of a security pattern may lead to new faults and failures, as the deployed security pattern can be faulty; e.g., a firewall might erroneously block messages. The following reasoning rules specify the consequences of deploying the firewall pattern. There is a new fault `IDPAT` (same ID of the pattern) associated with the firewall `FW`. This fault triggers a failure `IDFL` of type `erroneous`:

```
ft(IDPAT, [FW]) :- securityPattern(IDPAT, [firewall, [_, _, FW], _, _, _]).
f1(IDFL, [err]) :- securityPattern(IDPAT, [firewall, [_, _, FW], _, _, _]),
  ft(IDPAT, [FW]), createID(IDPAT, firewall, IDFL).
```

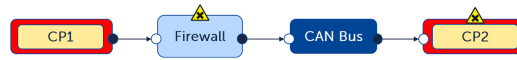


Fig. 6. Illustration of a cascading failure due to the deployment of a firewall.

```

ft2f1(IDPAT, IDFL) :- securityPattern(IDPAT, [firewall, [_ , _ , FW], _ , _ , _]),
  ft(IDPAT, [FW]), f1(IDFL, [err]), createID(IDPAT, firewall, IDFL).
  
```

We also specify rules to check whether there is any cascading failure due to the deployment of a security pattern, i.e., to check whether the fault associated to a pattern component triggers an identified hazard. Figure 6 illustrates a cascading failure caused by the deployment of a firewall.

The functionality of component CP2 depends on the signals sent by component CP1. We assume a fault in CP2 that triggers failures leading to an identified hazard HZ. Assume that the deployed Firewall erroneously blocks messages from CP1. As a result, the failures from Firewall might lead to hazard HZ.

**7.3.4 Safety Consequences on the Headlamp System.** Consider the headlamp system with the firewall pattern illustrated in Figure 5. The deployment of the firewall pattern leads to a new fault (triggering erroneous failures) on the firewall. These failures might lead to a cascading failure affecting the Body Control (e.g., when a user attempts to perform a software update via the OBD-II Connector). However, since the functionality of the headlamp system is independent of its external components, this cascading failure might not cause any harm.

**7.3.5 Safety and Security Consequences.** An integration activity is required to harmonize the safety and security consequences. This activity requires a manual analysis by safety and security engineers to assess the impact of the new faults and threats caused by the deployment of architecture patterns.

The manual analysis may include the assessment of whether (1) the new faults may lead to high criticality hazards and (2) the new threats are associated with components placed at communication channels with safety-relevant information. In case scenario (1) or (2) is found, safety and security engineers may either use measures like testing, simulation, or formal verification techniques to minimize the risks of these faults and threats or run SAFSECPAT to recommend further architecture patterns. For example, SAFSECPAT may recommend the Heterogeneous Duplex pattern associated with a faulty firewall, where the second instance of the firewall shall be implemented by a different security team, and the fault detector shall check whether the outputs from both firewalls match. Notice, however, that when using SAFSECPAT, new safety and security consequences will be found (e.g., new potential threats related to the Heterogeneous Duplex pattern). As a result, another manual analysis shall be carried out until a consensus is found between safety and security engineers. We leave to future work the investigation on how to improve SAFSECPAT to find optional design solutions between safety and security consequences.

## 8 RELATED WORK

Similar to our approach, [11] proposes a methodology to harden system architectures by automating the choice of safety patterns to avoid failures. They provide a hardening strategy that consists of (1) *component selection*, which selects a component of the architecture that a safety pattern shall be added; (2) *pattern selection*, which selects a pattern from a pre-defined library of patterns, and (3) *component substitution*, which replaces the selected component by its hardened version with a safety pattern. This strategy is automated by the SAT4J solver [4]. The key difference to our work is that we provide means to harden system architectures with security patterns, in addition to safety patterns. We also provide means to automate the consequences of applying security patterns to

safety and vice versa. Safety-wise, our reasoning principles enable a more precise recommendation of safety patterns as we specify a detailed intent for each safety pattern.

Once a safety pattern is selected, there shall be assumptions to ensure that the selected pattern is correctly applied to the system. Recently, [33] proposed a methodology for ensuring the application of safety and security patterns using contracts. By *contract*, they refer to a pair of assumptions and properties such that the properties only hold if the assumptions hold. Relying on the instantiation of a pattern template that includes both assumptions and properties (similar to the template used by this article), they proposed a safety case argument pattern to guide the assurance of systems using patterns. The specification of architecture pattern contracts are, however, done using informal descriptions only. We provide a specification of architecture patterns that enables automation, including the generation of assumptions for each architecture pattern. As future work, we plan to investigate how to extend SAFSECPAT to support an architecture pattern with contracts.

Safety and security co-analysis using patterns has been addressed by some previous work [24, 29]. We have been greatly inspired by [24], which proposed a pattern-based approach for safety and security co-analysis, and by [29], with security analysis of safety patterns. A key difference to our work is that we propose automated reasoning methods with safety and security patterns, whereas previous activities were done manually.

Model-based models and methods have been proposed for safety and security co-analysis, using languages such as GSN and Attack Trees and their combination [22, 26, 29, 30]. The key purpose of these approaches is to elucidate and document arguments demonstrating safety and security. Therefore, the artifacts produced often lie in high levels of abstraction, e.g., expressing high-level safety and security goals or not addressing the fact that safety and security have different semantics and different risk assessment methods. Our work complements these approaches by providing means to automate reasoning based on declarative semantics provided by answer-set programs. Indeed, we have developed a plugin that integrates the safety-related parts of SAFSECPAT into the model-based system engineering tool AutoFOCUS3 [15] to provide automated safety reasoning in a model-based engineering development [7].

Some previous works proposed the use of security guide-words to identify information that is relevant for safety [13, 14]. For example, [14] provided a mapping involving SGM guide-words, CIA triad, and STRIDE nomenclature. Using threat categories (e.g., STRIDE) enables a systematic identification of threat scenarios, possibly easing the recommendation of security patterns. We believe that finer reasoning principles can be obtained by using more specific guide-words such as those proposed by [13, 14]. This is left for future work.

Finally, we have recently demonstrated in a white paper [9] that SAFSECPAT can also perform security analysis following the ISO 21434 risk assessment.

## 9 CONCLUSION

We have proposed the use of semantically rich safety and security patterns for enabling automated support for safety and security co-design. We have proposed a DSL that enables the specification of safety and security concepts. We demonstrated its use in the description of several well-known patterns. By using KRR methods, we demonstrate how one can specify reasoning principles as answer-set programs. As a result, it can automate several activities, e.g., when a potential threat can be derived from identified failures, and when a potential threat becomes a threat (including the attack path). Moreover, our machinery (SAFSECPAT) can automatically recommend which pattern can be used at which place of the system architecture to address failures or threats, as well as make explicit consequences of deploying such patterns.



We are investigating how to extend our plugin [7] to integrate the security-related parts of SAFSECPAT into the model-based engineering tool AutoFOCUS3 [15]. The scalability of SAFSECPAT shall also be investigated. SAFSECPAT currently provides all possible architecture solutions to the user. In our recent article [7], we have defined four criteria to help the user in selecting the most suitable architecture for the system. We have carried out some initial experiments regarding the computation time of SAFSECPAT. We believe that the computation time of SAFSECPAT increases depending on the number of safety or security artifacts (e.g., on the number of faults). We have applied SAFSECPAT to an industrial use case taken from the automotive domain [7], where eight faults have been identified. SAFSECPAT took around 10 minutes to compute all solutions. Given that the focus of SAFSECPAT is on design time and not runtime, SAFSECPAT's performance requirements may range on hours or even days. In the future, a dedicated study shall be carried out to determine exactly the scalability of SAFSECPAT.

## REFERENCES

- [1] Ashraf Armoush. 2010. *Design Patterns for Safety-critical Embedded Systems*. Ph.D. Dissertation. RWTH Aachen University.
- [2] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.* 1, 1 (2004), 11–33.
- [3] Chitta Baral. 2010. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.
- [4] Daniel Le Berre and Anne Parrain. 2010. The sat4j library, release 2.2. *J. Satisf. Boolean Model. Comput.* 7, 2–3 (2010), 59–64.
- [5] Betty H. C. Cheng, Bradley Doherty, Nick Polanco, and Matthew Pasco. 2019. Security patterns for automotive systems. In *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS Companion '19)*. IEEE, 54–63. <https://doi.org/10.1109/MODELS-C.2019.00014>
- [6] Yuri Gil Dantas, Antoaneta Kondeva, and Vivek Nigam. 2020. Less manual work for safety engineers: Towards an automated safety reasoning with safety patterns. In *ICLP*.
- [7] Yuri Gil Dantas, Tiziano Munaro, Carmen Cărlan, Vivek Nigam, Simon Barner, Shiqing Fan, Alexander Pretschner, Ulrich Schöpp, and Sergey Tverdyshev. 2022. A model-based system engineering plugin for safety architecture pattern synthesis. In *Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'22), Online Streaming*, Luís Ferreira Pires, Slimane Hammoudi, and Edwin Seidewitz (Eds.). SCITEPRESS, 36–47. <https://doi.org/10.5220/0010831700003119>
- [8] Yuri Gil Dantas and Vivek Nigam. 2021. <https://github.com/ygdantas/safsecpat>.
- [9] Yuri Gil Dantas, Vivek Nigam, and Harald Ruess. 2020. Security engineering for ISO 21434. *CoRR* abs/2012.15080 (2020). arXiv:2012.15080
- [10] Yuri Gil Dantas, Vivek Nigam, and Carolyn Talcott. 2020. A formal security assessment framework for cooperative adaptive cruise control. In *IEEE Vehicular Networking Conference (VNC'20)*.
- [11] Kevin Delmas, Rémi Delmas, and Claire Pagetti. 2015. Automatic architecture hardening using safety patterns. In *SAFECOMP (Lecture Notes in Computer Science, Vol. 9337)*. Springer, 283–296.
- [12] Danny Dolev and Andrew Chi-Chih Yao. 1983. On the security of public key protocols. *IEEE Trans. Inf. Theory* 29, 2 (1983), 198–207. <https://doi.org/10.1109/TIT.1983.1056650>
- [13] Juergen Duerrwang, Kristian Beckers, and Reiner Kriesten. 2017. A lightweight threat analysis approach intertwining safety and security for the automotive domain. In *SAFECOMP*.
- [14] J. Duerrwang, M. Braun, , R. Kriesten, and A. Pretschner. 2018. Enhancement of automotive penetration testing with threat analyses results. *SAE Intl. J. of Transportation Cybersecurity and Privacy* 1, 2 (2018), 91–112.
- [15] fortiss GmbH. 2020. *AutoFOCUS 2.19*. <https://www.fortiss.org/en/publications/software/autofocus-3>.
- [16] Richard Gay, Heiko Mantel, and Barbara Sprick. 2011. Service automata. In *FAST (Lecture Notes in Computer Science, Vol. 7140)*, Gilles Barthe, Anupam Datta, and Sandro Etalle (Eds.). Springer, 148–163.
- [17] Michael Gelfond and Vladimir Lifschitz. 1990. Logic programs with classical negation. In *ICLP*.
- [18] R. Hammett. 2001. Design by extrapolation: An evaluation of fault-tolerant avionics. In *20th Digital Avionics Systems Conference (DASC'01) (Cat. No.01CH37219)*, Vol. 1. 1C5/1–1C5/12 vol.1. <https://doi.org/10.1109/DASC.2001.963314>
- [19] ISO26262. 2018. ISO 26262, road vehicles - functional safety - Part 6: Product development: Software level. Available at <https://www.iso.org/standard/43464.html>.
- [20] ISO/SAE AWI 21434. 2020. Road vehicles - cybersecurity engineering.

- [21] John C. Knight. 2002. Safety critical systems: Challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19–25 May 2002, Orlando, Florida, USA*, Will Tracz, Michal Young, and Jeff Magee (Eds.). ACM, 547–550. <https://doi.org/10.1145/581339.581406>
- [22] Antoaneta Kondeva, Carmen Carlan, Harald Ruess, and Vivek Nigam. 2019. On computer-aided techniques for supporting safety and security co-engineering. In *WoSoCer*.
- [23] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. 2006. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Logic* 7 (2006), 64 pages.
- [24] Helmut Martin, Zhendong Ma, Christoph Schmittner, Bernhard Winkler, Martin Krammer, Daniel Schneider, Tiago Amorim, Georg Macher, and Christian Kreiner. 2020. Combined automotive safety and security pattern engineering approach. *Reliab. Eng. Syst. Saf.* 198 (2020), 106773.
- [25] MIL-STD-2165. 1985. Military Standard Testability Program for Electronic Systems and Equipments.
- [26] Gabriel Pedroza. 2018. Towards safety and security co-engineering - Challenging aspects for a consistent intertwining. In *ESORICS*.
- [27] Ludovic Pietre-Cambacedes and Marc Bouissou. 2013. Cross-fertilization between safety and security engineering. *Reliab. Eng. Syst. Saf.* (2013).
- [28] Nikolaos Polatidis, Michalis Pavlidis, and Haralambos Mouratidis. 2018. Cyber-attack path discovery in a dynamic supply chain maritime risk management system. *Computer Standards & Interfaces* 56 (2018), 74–82.
- [29] Christopher Preschern, Nermin Kajtazovic, and Christian Kreiner. 2013. Security analysis of safety patterns. In *PLoP*.
- [30] Magdy El Sadany, Christoph Schmittner, and Wolfgang Kastner. 2019. Assuring compliance with protection profiles with threatget. In *SAFECOMP 2019 Workshops*.
- [31] Adam Shostack. 2014. *Threat Modeling: Designing for Security*. Wiley.
- [32] Martin A. Skoglund, Fredrik Warg, and Behrooz Sangchoolie. 2018. In search of synergies in a multi-concern development lifecycle: Safety and cybersecurity. In *SAFECOMP 2018 Workshops*. Springer.
- [33] Irfan Sljivo, Garazi Juez Uriagereka, Stefano Puri, and Barbara Gallina. 2020. Guiding assurance of architectural design patterns for critical applications. *J. Syst. Archit.* 110 (2020), 101765. <https://doi.org/10.1016/j.sysarc.2020.101765>
- [34] Wired. 2015. Hackers Remotely Kill a Jeep on the Highway-With Me in It. <https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/>.

Received 16 July 2021; revised 27 February 2022; accepted 14 September 2022

# Chapter 10

## A Framework for Assessing the Safety and Security of CACC Systems

Chapter 10 proposes a formal framework for the safety and security verification of CACC platoons. The proposed framework consists of CACC platoon model, an intruder model, and security mechanism specifications aimed at mitigating attacks based on the intruder model. The specification of the security mechanisms may be derived from pattern requirements often defined during security activities performed on the left side of the V-model. The proposed framework has been validated with a number of attacks taken from the literature and novel attacks discovered by using our formal machinery.

**Contributing article:** Yuri Gil Dantas, Vivek Nigam, and Carolyn L. Talcott: A Formal Security Assessment Framework for Cooperative Adaptive Cruise Control. VNC 2020: 1-8

**Copyright information:** © [2020] IEEE. Reprinted, with permission, from Yuri Gil Dantas; Vivek Nigam; Carolyn Talcott, *A Formal Security Assessment Framework for Cooperative Adaptive Cruise Control*, 2020 IEEE Vehicular Networking Conference (VNC), December 2020.

**Author contributions:** The concept for the publication was jointly developed by Yuri Gil Dantas, Vivek Nigam, and Carolyn Talcott. Together, they discussed the expansion of Carolyn Talcott's framework [38] to incorporate the assessment of CACC system security. This article proposes a formal framework for assessing the security of CACC systems, with Vivek Nigam focusing on formalizing the vehicle platoon behavior and Yuri Gil Dantas formalizing the intruder models, attacks, and countermeasures. The simulations were conducted by Yuri Gil Dantas, while the initial draft of the article was jointly written by Yuri Gil Dantas and Vivek Nigam. Carolyn Talcott provided valuable input in enhancing the article, and Yuri Gil Dantas handled subsequent revisions and corrections.

# A Formal Security Assessment Framework for Cooperative Adaptive Cruise Control

Yuri Gil Dantas  
*fortiss GmbH*  
 München, Germany  
 dantas@fortiss.org

Vivek Nigam  
*fortiss GmbH*  
 München, Germany  
 nigam@fortiss.org

Carolyn Talcott  
*SRI International*  
 Melno Park, USA  
 clt@esl.sri.com

**Abstract**—For increased safety and fuel-efficiency, vehicle platoons use Cooperative Adaptive Cruise Control (CACC) where vehicles adapt their state, incl. speed and position, based on information exchanged between vehicles. Intruders, however, may carry out attacks against CACC platoons by exploiting the communication channels used to cause harm, e.g., a vehicle crash. Therefore, during design-phase, engineers should provide evidence supporting platoon security. This paper proposes a formal framework for the security verification of CACC platoons to provide such evidence based on precise mathematical models. Our vehicle platoon models support the specification of both cyber, e.g., communication protocols, and physical, e.g., speeds, position, vehicle behaviors. Moreover, we propose intruder models that are parametric on his capabilities of manipulating communication channels, i.e., message injection and blocking. Our model is implemented enabling the automated formal verification involving both platoon and intruder models. We validate our machinery with a number of attacks taken from the literature and novel attacks discovered by using our formal machinery.

**Index Terms**—attacks, formal verification, platoon, security

## I. INTRODUCTION

Cooperative Adaptive Cruise Control (CACC) increases fuel-efficiency [21] and safety of vehicle platoons [2], typically heavy-weight cargo vehicles (e.g., trucks). This is accomplished by reducing vehicle reaction times by relying on information, such as speed, direction and position, exchanged between vehicles in addition to the vehicles' sensors.

The use of CACC also greatly increases a platoon's attack surface as communication channels may be exploited by intruders. For example, as pointed out by [22], intruders may inject messages with false information into the CACC communication channels leading to vehicle crashes, thus causing harm and financial losses. Intruders can carry out such attacks for financial motivations to, e.g., steal the transported cargo.

Designing secure systems is challenging as intruders may carry out attacks by exploiting corner-cases or implicit requirements overseen by developers. For example, a number of communication protocols have been shown to be vulnerable to attacks, some of which have been discovered decades after they have been developed [13]. The safety and security of vehicle platooning have the additional complexities of cyber-physical systems, including speed, time to react, and position. Engineers have to ensure that intruders cannot exploit these aspects, as in the injection attacks described by [22].

This paper proposes the use of formal verification as a means to provide further evidence about the security of platoons using CACC. An advantage of formal verification over, e.g., simulation analysis, lies on the fact that its methods are based on precise mathematical models that specify the behavior of the analyzed system. By using formal verification, implicit requirements are made explicit thus exposing existing vulnerabilities. Moreover, from such models, automated tools can determine whether undesired events are possible by traversing all behaviors including corner-cases.

Existing formal frameworks for platooning [8], [10] and other agent-based cyber-physical systems [19], [20] have successfully been used to verify the safety of agent-based cyber-physical systems, such as platoon joining maneuvers and strategies used by Unmanned Aerial Vehicles [15]. These frameworks, however, do not take into account security aspects. They do not include intruders and therefore, it is not possible to verify in such frameworks whether an intruder may attack a system and cause harm, e.g., a vehicle crash.

To the best of our knowledge, this paper proposes the first formal framework to consider platooning, CACC and security. Our main contributions are three-fold:

- **Vehicle Platoon Behavior Specification:** Our first contribution is a platoon model that includes specifications of both cyber aspects, e.g., specifications for the communication protocols, and physical aspects, e.g., speed, acceleration, positions of vehicles. Our model enables the specification of a wide range of vehicle strategies for executing platooning based on soft-constraints [4], a general algebraic framework for specifying optimization problems. That is, our model can accommodate a number of strategies including those expressed as classical, fuzzy and probability theories and their combination. For example, strategies for maintaining distances between vehicles that are both safe and fuel-efficient can be reduced to an optimization problem based on soft-constraints.
- **Intruder Models:** Our second contribution consists of formal intruder models that subvert communication channels to carry out attacks. These intruder models are parametric on the intruder capabilities, i.e., the capability of either blocking messages from a communication channel or injecting messages into communication channels.
- **Automated Verification:** Our third contribution is the

implementation of our models, both platoon and intruder models, in Maude [5], an efficient formal verification tool based on Rewriting Logic. Our specifications are executable. That is, users can automatically invoke Maude's search mechanisms to formally verify their platooning specifications for the verification of safety, *e.g.*, vehicles not crashing, by taking into account security, *e.g.*, in scenarios where an intruder may block or inject messages.

We validate our machinery in realistic scenarios, some taken from the literature [9], [22], and some new attacks that have been discovered by using our formal framework.

## II. ATTACKS

This section describes both the threat model and a set of possible attacks scenarios against a CACC platoon. To the best of our knowledge, some of the attack scenarios, namely, those described in Sections II-D, II-E and II-F, are new. They have been discovered using formal verification, as potential breaches became clearer after formalizing the platoon model, in particular its communication protocols and the modes (or roles) in which a vehicle can operate.

### A. Threat Model

We consider a CACC platoon, with one leader and  $n$  followers, where new vehicles may join the platoon after a negotiation phase. We assume that the platoon vehicles navigate on a straight road, and that vehicles can communicate using peer-to-peer connections or by broadcasting messages [23]. We also assume that all messages are signed using vehicles secret keys that cannot be guessed by intruders, and contain adequate measures to ensure freshness, such as using timestamps or nonces, to avoid replay attacks.

The goal of our intruder is to cause a crash between two legitimate vehicles. To this end, the intruder either injects false messages into the CACC communication channels or jams (*i.e.*, blocks) legitimate messages from the CACC communication channels. The actual capability used by the intruder depends on the attack scenario. We consider scenarios where the intruder (1) injects false messages only, (2) blocks messages only, and (3) both injects and blocks messages. To ensure that injected messages are valid, we assume that the intruder is able to obtain encryption keys from any vehicle in the platoon. The same assumption is considered by previous related work like, *e.g.*, [22] and [9]. For simplicity, we assume that the intruder has obtained the leader's encryption key.

Given the leader's encryption key, the intruder makes valid connections with a target vehicle (*i.e.*, a follower or a joining vehicle). For example, assume an attack scenario where both capabilities (*i.e.*, injecting and blocking) are required. The intruder blocks all messages originated from the leader and injects (impersonating the leader) false messages to either followers or vehicles joining the platoon.

### B. Injection of False Msgs against Follower

In this attack, an intruder sends false position and speed values to a vehicle in order to cause a crash with the preceding

vehicle. This attack works because CACC algorithms ensure that a vehicle maintains a desired distance from the preceding vehicle based on the received messages from other vehicles in the platoon (especially from the leader). This attack has been previously demonstrated through simulations by *e.g.*, [22].

The attack scenario is illustrated in Figure 1a. This scenario is composed of two vehicles: a leader (*ldr*) and a follower (*flw<sub>1</sub>*). Illustrated by the green arrows, such vehicles exchange information to ensure that *flw<sub>1</sub>* keeps a safe distance from *ldr*. The red cross illustrates that the legitimate messages from the leader are blocked by the intruder while the attack is in progress. Next, the intruder impersonates *ldr* to send high position and speed values to *flw<sub>1</sub>*. The follower *flw<sub>1</sub>* adapts its distance based on the high false values sent by the intruder. As a result, a crash between *flw<sub>1</sub>* and *ldr* is expected, as illustrated by the right-hand side of Figure 1a.

### C. Slow-Injection of False Msgs

The goal of the previous attack (Section II-B) is a quick crash between two vehicles. To this end, the intruder injects extreme false position and speed values into the CACC communication channels. As discussed by [22] and [9], however, existing countermeasures (a.k.a plausibility checks) are able to detect such extreme values, and thus mitigate the attack.

Recently, [9] proposed a smarter variation of the previous attack in order to bypass existing countermeasures that checks whether incoming values highly deviates from the previous received ones. To this end, the intruder injects messages with false information into the CACC communication channels modifying the values of speed and position with a small increase rate after each message. This attack has been demonstrated through simulations by [9].

### D. Injection of False Msgs against Joining Vehicle

A new vehicle may join a platoon after a negotiation phase with the leader of the platoon. During this negotiation phase, the leader sends the platoon information to this vehicle, including the position and speed of the last vehicle, so that the joining vehicle can adapt itself to catch up to the platoon.

An intruder may impersonate the leader to send false information during this negotiation phase. For example, assume an attack scenario composed of two vehicles: the leader (*ldr*) of the platoon and a vehicle (*veh*) that wishes to join the platoon. The intruder may inject (as *ldr*) high position and speed values to *veh* during the negotiation phase, while blocking all messages originated from *ldr*. Eventually, *veh* crashes into *ldr*, as *veh* adapts its acceleration based on the received values.

Countermeasures against injection attacks usually check on messages exchanged between platoon members (*e.g.*, followers). The attack scenario presented above targets a vehicle that has not yet joined the platoon. Hence, this attack would be successfully carried out against such countermeasures. To the best of our knowledge, this is the first attack scenario targeting a vehicle before joining the platoon.

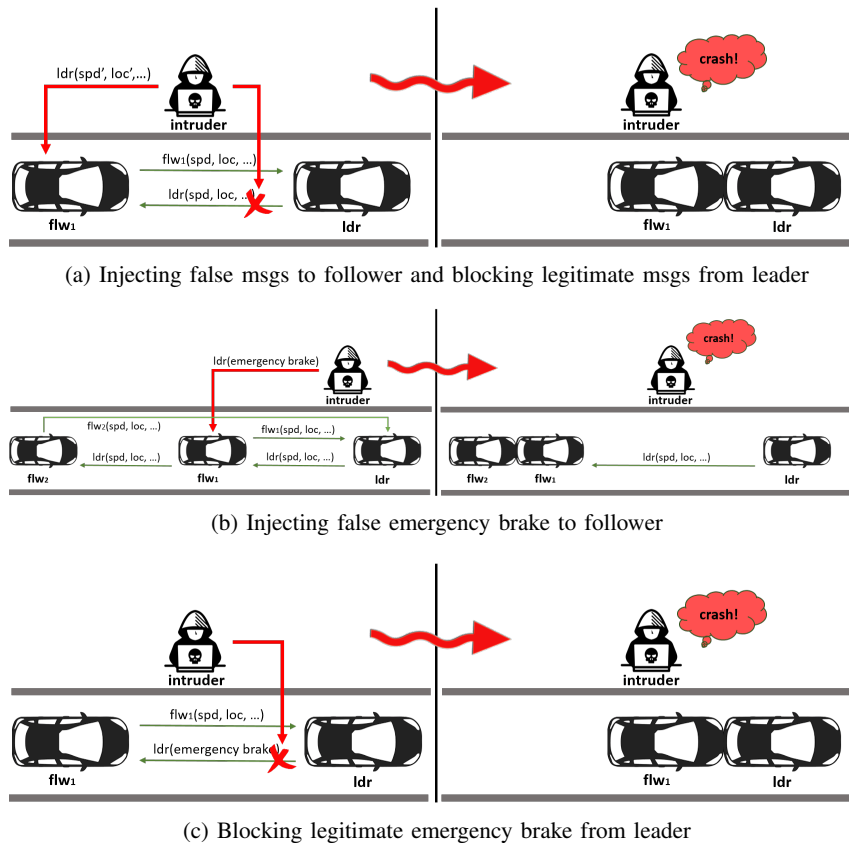


Fig. 1: Illustration of three attacks (before and after the attacks have been) carried out by the intruder

#### E. Injection of False Emergency Brake Msgs

Emergency brake is a safety-type message that may be triggered by any vehicle in the platoon to avoid crashes. For example, the leader may trigger an emergency brake if an obstacle is detected in its path. Then each follower receives an emergency brake message from the leader, and immediately actuates it by stopping the vehicle.

An intruder, however, might take advantage of this situation to carry out attacks. Figure 1b illustrates an attack scenario using emergency brake messages. This scenario is composed of three vehicles: a leader (*ldr*) and two followers (*flw*<sub>1</sub> and *flw*<sub>2</sub>). The goal of this attack is a crash between *flw*<sub>1</sub> and *flw*<sub>2</sub>. To this end, the intruder injects a false emergency brake message to *flw*<sub>1</sub> only. This message results in a crash as *flw*<sub>1</sub> immediately stops and *flw*<sub>2</sub> keeps driving, yet following the previously received information (*e.g.*, position and speed). The crash is illustrated by the right-hand side of Figure 1b. Our hypothesis is that the intruder does not need to block messages from the leader in order to successfully carry out this attack.

#### F. Blocking Legitimate Emergency Brake Msgs

Instead of injecting false emergency brake messages, the intruder may block legitimate emergency brake messages from the CACC communication channels in order to cause a crash. An attack scenario with this purpose is illustrated in Figure 1c.

The intruder monitors the channels till a legitimate emergency brake message is triggered by the leader (*ldr*). At this point, *ldr* stops the vehicle and the intruder blocks the message to avoid that any follower (*flw*<sub>1</sub>) can receive and trigger emergency brake as well. As a result, *flw*<sub>1</sub> keeps driving the vehicle till crashing into *ldr*. This crash is illustrated on the right-hand side of Figure 1c. This attack scenario is another result from our formal verification.

### III. SOFT-AGENTS MODEL FOR PLATOONING

Soft-Agents [19] is a rewriting logic framework for the specification and verification of (autonomous) cyber-physical agents. The framework can be found at [19], [20]. The framework is implemented in the rewriting logic language Maude [5]. It provides the general machinery (data-structures, functions, sorts) for the specification of the behavior of agents, *e.g.*, agent capabilities and effects of actions. The semantics of how the system evolves is specified by a small number of rewrite rules defined in term of the general machinery.

Figure 2 depicts the general architecture of a soft-agent, or simply agent. An agent has its own local knowledge base that contains, *e.g.*, its current perceived speed, position, and direction of the other agents. Further data may be obtained by sensing the environment or by sharing of information between agents through communication channels. Using its

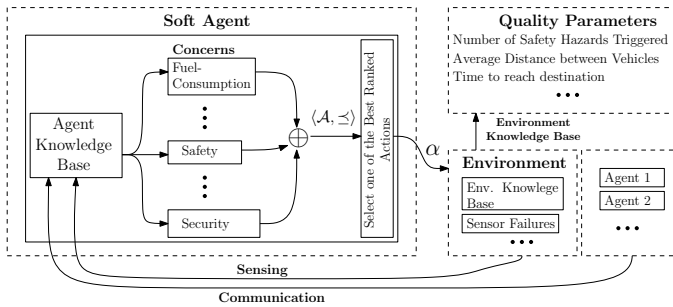


Fig. 2: Soft-Agent Architecture

local knowledge base, the agent decides which action ( $\alpha$ ) to perform according to its different concerns specified as a soft constraint (optimization) problem [4]. For example, if the distance to the vehicle in front is too great, the fuel consumption concern kicks in and attempts to reduce it by accelerating. Similarly, if the distance is dangerously short, then the safety concern kicks in and attempts to increase it by decelerating. As soft constraints subsume other constraint systems, *e.g.*, classical, fuzzy and probabilistic, it is possible to formally specify a wide range of decision algorithms.

#### A. Platooning Model

We instantiated the general framework (data structures, sorts, types, soft constraints) provided by the Soft Agents framework for specifying platoon scenarios, enabling their formal verification. While the complete implementation can be found at [6], we describe some of this machinery below.

**Knowledge Base:** Vehicles have a local knowledge base (lkb). It represents the vehicle's view of the world, *e.g.*, the speed and position of itself and of the other vehicles. Formally, a vehicle knowledge base is composed by a set of grounded facts, *p*, *i.e.*, facts not containing variable symbols, of the form *p*, or associated with a timestamp, *p@t*, where *t* is natural number. We list the main facts below. We assume that each vehicle has a unique identifier written *id*.

- $\text{clock}(t)$  denotes that the current time is *t*;
- $\text{atloc}(id, pos) @ t$  denotes that the vehicle *id* has at time *t* the position of value *pos*. We assume that vehicles navigate on a straight road as it is enough for the verification of the attacks described in Section II. Therefore, *pos* is a value representing the position on this road.
- $\text{speed}(id, spd) @ t$  denotes that the vehicle *id* has at time *t* the speed of value *spd*;
- $\text{maxAcc}(id, acc)$  denotes that the vehicle *id* can accelerate (and for simplicity also decelerate) at any time with value *acc*.
- $\text{platoon}(idL, [id1, \dots, idn]) @ t$  denotes that at time *t*, the platoon led by *idL* has the sequence of follower vehicles *id1*, *...*, *idn*;
- $\text{mode}(id, md) @ t$  denotes that the vehicle *id* at time *t* is in mode *md* which include: *nonplatoon* when all the vehicle's platooning functionalities are not active, *i.e.*, the vehicle is driven by a human driver; *leading()* when

the vehicle leads a platoon; *following(idL)* when *id* is following the platoon led by *idL*; *emergency* when *id* is in emergency brake mode;  $\text{fuseRear}(idL, idB)$  when *id* is in the process of joining platoon led by *idL* and shall join behind vehicle *idB*.

- $\text{safe}(id, min, max)$  denotes that the distance to the preceding vehicle of *id* is considered safe if it is between the values *min* and *max*;
- $\text{fuel}(id, min, max)$  denotes that the distance to the preceding vehicle of *id* is considered to be fuel efficient if it is between the values *min* and *max*;
- $\text{histSpd}(id, id1, spd1; \dots; spdn) @ t$  denotes that vehicle *id* has the *n* last speed values, *spd1*; *...*; *spdn*, of vehicle *id1*. This fact is used to build plausibility checks as detailed in Section V.
- $\text{histGap}(id, gap1; \dots; gapn) @ t$  denotes that vehicle *id* has the *n* last gap measurements, *gap1*; *...*; *gapn*, to the following vehicle.

*Example 3.1:* The following local knowledge base of vehicle  $v(1)$  specifies that he is following vehicle  $v(0)$ . The vehicle  $v(1)$  has speed 20 and position 945 distance units. He believes to be immediately behind vehicle  $v(0)$  with a gap of 55 distance units. The vehicle  $v(1)$  has a maximum acceleration of 3 acceleration units. Moreover, he keeps track of the three last speed values, 25, of  $v(0)$ .

```
lkb : (clock(3) (atloc(v(1), loc(945)) @ 3)
      (mode(v(1), following(v(0))) @ 3)
      (speed(v(1), 20) @ 3) (gapNext(v(1), 55) @ 3)
      (idNext(v(1), v(0)) @ 3) maxAcc(v(1), 3)
      (histSpd(v(1), v(0), 25 @ 3; 25 @ 2; 25 @ 1) @ 3)
      (histGap(v(1), 55 @ 3; 55 @ 2; 55 @ 1) @ 3)
      fuel(v(1), 1, 3) safe(v(1), 2, 4))
```

**Sensors:** A vehicle is equipped with three sensors *locS*, *speedS* and *gapS*. They measure, respectively, the vehicle's location, speed and the gap to the vehicle immediately ahead. As we illustrate below, at each tick, vehicles use these sensors to query the environment knowledge base and update the vehicle's local knowledge base. While it is not the focus of this work, it is possible to evaluate the robustness of agents with respect to sensor faults as described in [15].

**Communication Channels and Protocols:** We assume that vehicles may communicate using peer-to-peer connections or by broadcasting messages. Based on this assumption, we implement a number of protocols for platooning including:

- **Heartbeat from Follower to Leader (HFL):** A follower vehicle sends periodically a (time-stamped) message to the leader with information such as its current speed, position.
- **Heartbeat from Leader to Follower (HLF):** The platoon leader sends periodically a message to each follower with information of all vehicles in the platoon such as their speeds and positions.
- **Emergency Brake:** Any vehicle in the platoon may broadcast an emergency brake message informing that it is activating its emergency brakes.
- **Heartbeat from Joining Vehicle to Leader (HJL):** A vehicle that wants to join a platoon sends a heartbeat to the platoon leader, such as its current position and speed.

- **Heartbeat from Leader to Joining Vehicle (HLJ):** The platoon leader sends to the vehicle that is joining the platoon information, such as the position and speed of the last vehicle in the platoon.

**Actions:** Vehicles decide to accelerate or decelerate. Since there may be infinitely many possibilities of acceptable speeds (for safety and fuel efficiency), we abstract actions by using facts of the form  $\text{act}(\text{id}, \text{vmin}, \text{vmax})$  denoting a set of actions of changing  $\text{id}$ 's speed to values between  $\text{vmin}$  and  $\text{vmax}$ . Actions are evaluated with a value that is the result of a soft constraint problem specification described next.

**Soft Constraints:** The evaluation of possible actions is done by taking account the vehicle's concerns specified as a soft constraint problem. To evaluate our verification machinery, we implemented a strategy that depends on the vehicle's mode.

When in following mode, a vehicle has two main concerns: Fuel-Saving and Safety. The former attempts to close the gap to the vehicle immediately in front, while the latter attempts to keep a safe distance to the vehicle immediately in front. These are specified by the knowledge items  $\text{safe}$  and  $\text{fuel}$ .

Our machinery uses these two parameters to determine which (set of) actions are the most highly ranked. This is accomplished by attempting to satisfy both concerns, safety and fuel-saving. If this is not possible, then safety is given priority over fuel-saving. When in emergency mode, the vehicle has only the concern of stopping the vehicle.

*Example 3.2:* Assume the knowledge-base of vehicle  $v(1)$  in Example 3.1. Since its maximum acceleration is 3 and its current speed is 20, it may choose any speed between 17 and 23. From the safety concern, it attempts to keep a distance (in terms of time) between 2 and 4. From the current gap of 55 units and speed of the vehicle in front of 25, all speeds between 17 and 23 are acceptable. However, the fuel-saving concern attempts to keep a distance (in terms of time) between 1 and 3. Since the gap is too great and the vehicle in front is faster, only speeds between  $75/4$  and 23 are acceptable. The vehicle picks the average speed of 20.75, *i.e.*,  $v(1)$  accelerates.

**Vehicle Configuration and System Configuration:** A *vehicle configuration* contains its local knowledge base ( $\text{lkb}$ ), sensors ( $\text{sensors}$ ), and the events ( $\text{evs}$ ) that are to be processed. The events are used to define the execution semantics described in Section III-B.

*Example 3.3:* The following is an example of a vehicle configuration for  $v(1)$ , where  $\text{LKB}$  is the local knowledge base in Example 3.1. Finally, it has a single event  $\text{tick} @ 0$  that specifies that this vehicle is ready to observe the state and schedule new actions.

```
[v(1) : veh | lkb : LKB
  sensors : (locS speedS gapS),
  evs : (tick @ 0)]
```

Finally, a *system configuration* is composed of a collection of vehicle configurations,  $\text{vconfi}$ , for  $0 \leq i \leq n$ , and an environment  $[\text{eId} | \text{kb}] : \{ [\text{eId} | \text{kb}] \text{vconfi1} \dots \text{vconfin} \}$ . The environment knowledge base,  $\text{kb}$ , contains the true state of the world which may be different to the information of the local knowledge bases of vehicles.

## B. Executable Semantics

The execution semantics of our platooning model follows the general semantics described in [19], [20]. Formally, an execution is a finite sequence of system configurations, written  $\mathcal{S}_0 \rightarrow \mathcal{S}_1 \rightarrow \dots \rightarrow \mathcal{S}_n$ , where each transition  $\mathcal{S}_i \rightarrow \mathcal{S}_{i+1}$  follows the executable semantics described below. In practice, an execution can be constructed in an automated fashion using the rewriting tool Maude [5]. We illustrate the semantics by using the platooning model described above.

The execution semantics follows an event-based approach. Vehicles have events of form  $\text{ev} @ \tau$  denoting an event  $\text{ev}$  that should be executed after  $\tau$  time units. If  $\tau$  is zero, then it is executed immediately. There are two types of events: tasks and actions. All events of the form  $\text{task} @ 0$  are executed, typically producing actions to be executed and new tasks for later execution. Suppose the smallest non-zero task delay is  $d$ . Then  $d$  time units pass as follows: execute all actions with zero delay; update the configuration to pass one unit of time; repeat until  $d$  units of time have passed.

Consider as an example the initial system configuration with two vehicles  $v(0)$  and  $v(1)$ , where  $\text{LKB}$  is as in Example 3.1 and  $\text{LKB1}$  is similar, just that the facts for the platoon, location and speed of  $v(0)$  are as specified, respectively, by  $(\text{platoon}(v(0), v(1)) @ 0)$   $(\text{atloc}(v(0), \text{loc}(1000)) @ 0)$   $(\text{speed}(v(0), 25) @ 0)$ :

```
{ [eId] | kb }
[v(0) : veh | lkb : LKB1,
  sensors : (locS speedS gapS),
  evs : (tick @ 0)]
[v(1) : veh | lkb : LKB,
  sensors : (locS speedS gapS),
  evs : (tick @ 0)] }
```

At this point, the system checks for tasks of each vehicle, namely the following tasks:

- 1) Update their local knowledge base with the information extracted by the sensors, location, speed and gaps to vehicles in the front.
- 2) Executes their soft constraint machinery to determine which range of actions they shall perform. Since  $v(0)$  is the leader, it decides to maintain its speed at 25. This is specified by the event  $\{u(1), \text{actSpeed}(v(0), 22, 28)\} @ 0$  specifying that any speed between 22 and 28 is ranked as maximum denoted by  $u(1)$ . Thus the vehicle picks the average 25. On the other hand, as described in Example 3.2,  $v(1)$  computes the event  $\{u(1), \text{actSpeed}(v(1), 75/4, 23)\} @ 0$ , choosing to accelerate to speed 20.75.
- 3) Finally, the vehicles follow the communication protocols described in Section III-A. For this example, the leader vehicle  $v(0)$  creates an event to send a heartbeat to  $v(1)$ :  $(\text{actSnd}(v(0), \text{msg}(v(0), v(1)), \text{hb12f}(v(0), 25, \text{loc}(1000), \text{kb}) @ 0) @ 0)$  denoting the action to send a message from  $v(0)$  to  $v(1)$  containing as payload a heartbeat from leader to follower



(hb12f) with  $v(0)$ 's current speed and location, and kb that is a collection of facts which is elided.

Similarly, the following vehicle  $v(1)$  creates an event to send a heartbeat to the leader  $v(0)$  containing  $v(1)$ 's speed and position.

The following step in the execution semantics is to advance time and carry out all these events. This results in updating the speed of the vehicles, and sending both heartbeat messages. The contents of speed and location in these messages are then processed, updating the vehicle's local knowledge bases.

### C. Safety Verification (w/o Intruders)

We illustrate next how we can use the machinery described above to reason about the behavior of platoons and demonstrate properties. In particular, we are interested in determining whether two vehicles can crash with each other.

A vehicle crash may happen if the control measures are not adequately set or its assumptions are not met. Example 3.4 illustrates how our machinery can be used to demonstrate this. Moreover, an intruder may exploit the communication channels to cause an accident, as shown in Section IV.

*Example 3.4:* Consider the system configuration,  $S_0$ , from the previous section with two vehicles  $v(0)$  leading the platoon, and  $v(1)$  following  $v(0)$ . Assume the same parameters for the concerns *safe* and *fuel* as in Example 3.1.

Formally, an execution leads to a crash if it leads to a system configuration such that the location of  $v(0)$  is less or equal to the location of  $v(1)$ . Let  $\text{crash}(S)$  return true if  $S$  consist of system configuration with a crash and false otherwise.

We can use the following command in Maude:

```
search[1] S0 => S1 such that crash(S1) .
to search for an execution starting at  $S_0$  and ending at a configuration  $S_1$  such that  $\text{crash}(S_1)$  returns true, i.e., a configuration where the vehicles  $v(0)$  and  $v(1)$  crash.
```

Running this command, Maude does not find any such  $S'$ , thus providing evidence that the parameters for *safe* and *fuel* are correctly set.

However, if we use  $S_0'$ , where the speed of  $v(1)$  is 40 instead of 20, and run the command

```
search[1] S0' => S1 such that crash(S1) .
then Maude finds in 52ms a configuration  $S_1$  where a crash happened. This result means that the chosen parameters do not work w.r.t. safety. Indeed, one could expect a crash when the speed of a vehicle is much greater than the speed of its preceding vehicle.
```

By using this type of reasoning, engineers can verify whether the specified behavior of platoon is safe according to the assumptions used, recalibrating concerns whenever needed. In the example above, one should make sure that the vehicle  $v(1)$  is not much greater than the speed of  $v(0)$ .

## IV. INTRUDER MODEL

This section introduces an intruder model, formalizing the threat model discussed in Section II. The intruder is capable of impersonating an honest vehicle, injecting messages, and

blocking message from communication channels. These capabilities enable us to carry out similar verification done for safety, but now considering a malicious intruder. For example, it is possible to analyze whether platoons are vulnerable to the attacks enumerated in Section II.

Our intruder model is similar to [17], for the security verification of Industry 4.0 applications, in that the intruder model is parametrized by its capabilities. Here we consider two capabilities: injecting messages signed by honest participants and blocking specific messages from communication channels.

An intruder model ( $\text{intSpec}$ ) is integrated to a system configuration  $\text{system}$ , forming an *intruder configuration*:  $\{ \text{system} ; \text{intSpec} \}$ , where  $\text{intSpec}$  has the following shape:

```
[iid : intruder | (v2vMsgsL : msgList)
  (blockActSnd : ids) caps]
```

It contains the intruder id  $\text{id}$ ; the sequence of messages,  $\text{msgList}$ , that the intruder may inject; and the vehicles,  $\text{ids}$ , whose output communications are blocked by the intruder.

The execution semantics described above is extended to accommodate the intruder:

- **Message Injection (INJ):** The intruder may choose *at any moment* of a system execution to inject the first message,  $\text{msg}$ , in its list of messages  $\text{msgList}$ . This results in the injection of  $\text{msg}$  to its destination in the system configuration  $\text{system}$ , and the list  $\text{msgList}$  is updated by deleting  $\text{msg}$ .
- **Blocking (BLK):** The vehicles in  $\text{ids}$  are jammed during the whole attack execution. This means that all outgoing messages of a vehicle in  $\text{ids}$  are blocked.

Our model is parametric w.r.t. the intruder capabilities. It requires little effort to include other capabilities to the intruder model in  $\text{caps}$ . For example, it is possible add capabilities where the intruder tampers, *i.e.*, modifies messages sent by vehicles; or periodically sends messages from a set of messages, instead of in a list; or only starts blocking a message after some particular time has elapsed.

As we describe in Section V, however, the intruder can carry out all the attacks described in Section II using the two capabilities specified above. The following example illustrates how one of the attacks can be modeled using our machinery:

*Example 4.1:* Consider the system described in Section III-A with two vehicles  $v(0)$  and  $v(1)$ , with, respectively, speeds 20 and 25, and locations, 1000 and 945. Moreover, consider the intruder with a single message:

```
msg(v(0), v(1), hb12f(v(0), 70, loc(1070), none))
```

The intruder impersonates  $v(0)$  informing  $v(1)$  that his speed is 70 and location is 1070. Since  $v(1)$  does not double check the contents of this message, it will start accelerating. This will lead to a crash as the actual speed of  $v(0)$  is 20.

This can be determined in an automated fashion using the following Maude's search command, where  $\text{isys0}$  is the intruder configuration described above:

```
search isys0 =>* isys1 such that crash(isys1) .
```

|      | Attack Scenario                                 | Capability | Countermeasure | # States | Execution Time (min) | Attack Successful |
|------|---|------------|----------------|----------|----------------------|-------------------|
| II-B | Injection of False Msgs against Follower        | INJ + BLK  | N              | 15351    | 0.034                | Y                 |
|      | Injection of False Msgs against Follower        | INJ + BLK  | COMM           | -        | 120                  | -                 |
|      | Injection of False Msgs against Follower        | INJ + BLK  | SNSR           | -        | 120                  | -                 |
|      | Injection of False Msgs against Follower        | INJ        | N              | -        | 120                  | -                 |
| II-C | Slow-Injection of False Msgs against Follower   | INJ + BLK  | N              | 3315284  | 52.764               | Y                 |
|      | Slow-Injection of False Msgs against Follower   | INJ + BLK  | COMM           | 3286681  | 53.251               | Y                 |
|      | Slow-Injection of False Msgs against Follower   | INJ + BLK  | SNSR           | -        | 120                  | -                 |
|      | Slow-Injection of False Msgs against Follower   | INJ        | N              | -        | 120                  | -                 |
| II-D | Injection of False Msgs against Joining Vehicle | INJ + BLK  | N              | 9408     | 0.023                | Y                 |
|      | Injection of False Msgs against Joining Vehicle | INJ + BLK  | COMM           | 9408     | 0.027                | Y                 |
|      | Injection of False Msgs against Joining Vehicle | INJ + BLK  | SNSR           | 9407     | 0.023                | Y                 |
|      | Injection of False Msgs against Joining Vehicle | INJ        | N              | -        | 120                  | -                 |
| II-E | Injection of False Emergency Brake Msgs         | INJ + BLK  | N              | 593      | 0.002                | Y                 |
|      | Injection of False Emergency Brake Msgs         | INJ        | N              | 2218     | 0.011                | Y                 |
| II-F | Blocking Legitimate Emergency Brake Msgs        | BLK        | N              | 6539     | 0.013                | Y                 |

TABLE I: Evaluation of the attack scenarios described in Section II. Some experiments were aborted after 120 minutes.

Maude finds an instance of the intruder configuration `isys1` in which  $v(0)$  and  $v(1)$  crash.

## V. VERIFICATION RESULTS

Our goal is to evaluate our machinery on a number of attack scenarios, including the ones described in Section II. To this end, we formalized such attack scenarios using the intruder model presented in Section IV. We run the search command in Maude to automatically check whether two vehicles crash under the presence of the intruder. We run all experiments on a 1.90GHz Intel Core i7-8665U with 16GB of RAM running Ubuntu 18.04 LTS with kernel 5.4.0-47-generic and Maude 3.

Table I summarizes our main results. It contains each attack scenario described in Section II, the capability used by the intruder, *i.e.*, either injection (INJ) or blocking (BLK), whether a countermeasure has been used against the attack, the number of states explored, the execution time of each search command, and whether the intruder was able to bypass any countermeasure and successfully cause a crash between two vehicles.

For the attacks described in Sections II-B, II-C, and II-D we evaluate their effectiveness with and without a countermeasure. We formalized two types of countermeasures, abbreviated in Table I as COMM (communication-based) and SNSR (sensor-based) based on similar countermeasures proposed by [9]. The COMM countermeasure works as follows. Whenever a vehicle receives a message with the speed of the preceding vehicle, the countermeasure checks it against the local `histSpd`. The countermeasure is triggered if the incoming speed value deviates from 30% w.r.t. the average of the last `n` speed values received by the vehicle. The SNSR countermeasure estimates the speed of the preceding vehicle based on the information obtained from the gap sensor. That is, we estimate the speed of the preceding vehicle by computing  $(spd + (gap2 - gap1)) / 2$ , `spd` as the speed of the vehicle and `gap2` and `gap1` as the last two gap distance measurements (taken from the local `histGap`). The SNSR countermeasure is triggered if the incoming speed value deviates from 30% w.r.t. the estimated speed of the preceding vehicle.

Our intruder using both capabilities has successfully carried out the attacks II-B, II-C, and II-D against a platoon without countermeasure. Attack II-B, however, has not led to a crash when the countermeasures were deployed. This result was expected as attack II-B sends high speed values to a target vehicle. We run the search command to look for a crash between two vehicles without the countermeasure being triggered. We could not find any crash in 120 minutes.

The attack II-C bypassed the communication-based countermeasure, but not the sensor-based countermeasure. This result confirms the findings of [9] that feeding countermeasures with information from local sensors can be effective against slow-injection attacks. Next, the attack II-D led to a crash even when the countermeasures were deployed. The attack II-D is carried against a vehicle during the negotiation phase. This attack is effective as the considered countermeasures are only valid for platoon members (using their local `histSpd` or `histGap`).

Interestingly, neither of those three attacks led to a crash using the injection capability only (*i.e.*, no blocking at the same time). This happens because the target vehicle receives legit and false messages during the attack, and dynamically adapts its acceleration based on the received messages. That is, the target vehicle accelerates when receiving a false message from the intruder, *e.g.*, with high speed values, and decelerate when receiving legit messages from the leader. Therefore, we speculate that anti-jamming countermeasures, *e.g.*, UFH-UDSSS [18], could serve as an additional layer of defense against injection attacks for CACC platoons.

Finally, both the attacks using emergency brake messages led to a crash. In particular, the attack II-E is effective even without blocking messages from the communication channels. This is due to the fact that vehicles immediately stop driving upon receiving an emergency brake message regardless of any message (usually blocked by the intruder) sent by the leader.

## VI. RELATED WORK

We have been inspired by [3], [22], [9] that investigated the attacks described in Sections II-B and II-C. This paper

advances these previous work by proposing a formal framework enabling engineers to formalize platoon and intruder behaviors, and formally verify these models in an automated fashion. Our framework provides evidence for the security of platooning based on precise mathematical models, thus complementing the evidence obtained with the use of simulation based methods as in [3], [22], [9]. Further, we also propose three new attacks that have not been considered before.

A number of formal frameworks have been proposed for the specification and verification of cyber-physical agents [8], [10], [14], [15], [19], [20] including for vehicle platooning [8], [10], [14]. A key difference is that our work also considers how intruders may cause harm, whereas these existing frameworks focused on the safety of systems without considering security and intruder models. For example, [8] uses Statistical Model Checking to evaluate the impact of sensor and network faults to the safety of systems. We have in our previous work [15] also used Statistical Model Checking together with the Soft Agents framework for the verification of UAV strategies in the presence of sensor faults. We find it intriguing and leave it to future work the combination of intruder and fault models and their verification techniques for vehicle platooning.

A countermeasure against false position values has been proposed by [12]. It employs low-power beaconing messages to check whether incoming messages indeed comes from physically close vehicles, thus mitigating remote attacks from intruders not located near to platoon members. It seems possible to extend our model to accommodate such aspects following our previous work on Cyber-Physical Security Protocols [16]. We leave this investigation to future work.

## VII. CONCLUSION

This paper proposes to the best of our knowledge the first formal security framework for the specification and verification of vehicle platoon using CAAC. Our model can express communication protocols used by vehicles to exchange information about their physical states, such as speed and position, and can express vehicle behavior including how information exchanged is used to make decision about speed and position. Finally, our framework has an intruder model that is parametric to capabilities, *i.e.*, message injection and blocking. We demonstrated the effectiveness of our framework by formalizing existing attacks and proposing three new attacks.

We are considering a number of future work directions. We are considering other physical features, *e.g.*, the use of Cyber-Physical Security Protocols [16] to enable verification with proximity-based countermeasures [12]. We are implementing further intruder capabilities that reflect the capabilities of the Dolev-Yao model [7], but taking care not to fall into undecidable verification problems. We are also considering abstract models, such as those considered in [11], to enable completeness of our automated verification. We are planning to extend our framework to also support reasoning with multi-lane highway scenarios and reasoning with fault models as in [15]. Finally, we are integrating our machinery into an existing MBS tool, namely AutoFOCUS 3 [1].

## ACKNOWLEDGMENT

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 830892. Nigam is partially supported by NRL grant N0017317-1-G002, and CNPq grant 303909/2018-8. Talcott was partially supported by ONR grants N00014-15-1-2202 and N00014-20-1-2644, and NRL grant N0017317-1-G002.

## REFERENCES

- [1] AF3 – AutoFOCUS 3. More information at <https://af3.fortiss.org/>.
- [2] White paper: Automated driving and platooning issues and opportunities. Available at <https://tinyurl.com/yxzepft3>, 2015.
- [3] M. Amoozadeh, A. Raghuramu, C. Chuah, D. Ghosal, H. M. Zhang, J. Rowe, and K. N. Levitt. Security vulnerabilities of connected vehicle streams and their impact on cooperative driving. *IEEE Commun. Mag.*, 53(6):126–132, 2015.
- [4] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *J. ACM*, 44(2):201–236, 1997.
- [5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.
- [6] Y. G. Dantas, V. Nigam, and C. Talcott. <https://github.com/ygdantas/SoftAgents-Platoon.git>. 2020.
- [7] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
- [8] S. Hyun, J. Song, S. Shin, and D. Bae. Statistical verification framework for platooning system of systems with uncertainty. In *APSEC*, 2019.
- [9] M. Iorio, F. Risso, R. Sisto, A. Buttiglieri, and M. Reineri. Detecting Injection Attacks on Cooperative Adaptive Cruise Control. In *2019 IEEE Vehicular Networking Conference, VNC*, pages 1–8, 2019.
- [10] M. Kamali, L. A. Dennis, O. McAree, M. Fisher, and S. M. Veres. Formal verification of autonomous vehicle platooning. *Sci. Comput. Program.*, 148:88–106, 2017.
- [11] M. I. Kanovich, T. B. Kirigin, V. Nigam, A. Scedrov, and C. L. Talcott. Time, computational complexity, and probability in the analysis of distance-bounding protocols. *Journal of Computer Security*, 25(6), 2017.
- [12] H. Kim and T. Kim. Vehicle-to-vehicle (V2V) message content plausibility check for platoons through low-power beaconing. *Sensors*, 19(24):5493, 2019.
- [13] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *TACAS*, pages 147–166, 1996.
- [14] P. Mallozzi, M. Sciancalepore, and P. Pelliccione. Formal verification of the on-the-fly vehicle platooning protocol. In *SERENE*, 2016.
- [15] I. Mason, V. Nigam, C. L. Talcott, and A. V. D. Brito. A framework for analyzing adaptive autonomous aerial vehicles. In *SEFM*, pages 406–422, 2017.
- [16] V. Nigam, C. Talcott, and A. A. Urquiza. Towards the automated verification of cyber-physical security protocols: Bounding the number of timed intruders. In *ESORICS*, 2016.
- [17] V. Nigam and C. L. Talcott. Formal security verification of industry 4.0 applications. In *ETFA*, pages 1043–1050, 2019.
- [18] C. Pöpper, M. Strasser, and S. Capkun. Anti-jamming broadcast communication using uncoordinated spread spectrum techniques. *IEEE J. Sel. Areas Commun.*, 28(5):703–715, 2010.
- [19] C. Talcott, V. Nigam, F. Arbab, and T. Kappé. Formal specification and analysis of robust adaptive distributed cyber-physical systems. In M. Bernardo, R. D. Nicola, and J. Hillston, editors, *SFM*. 2016.
- [20] C. L. Talcott, F. Arbab, and M. Yadav. Soft agents: Exploring soft constraints to model robust adaptive distributed cyber-physical agent systems. In *Software, Services, and Systems - Essays Dedicated to Martin Wirsing*, pages 273–290, 2015.
- [21] S. van de Hoef, K. H. Johansson, and D. V. Dimarogonas. Fuel-efficient en route formation of truck platoons. *IEEE Trans. Intell. Transp. Syst.*, 19(1):102–112, 2018.
- [22] R. W. van der Heijden, T. Lukaseder, and F. Kargl. Analyzing attacks on cooperative adaptive cruise control (CACC). In *2017 IEEE Vehicular Networking Conference, VNC*, pages 45–52. IEEE, 2017.
- [23] T. L. Willke, P. Tientrakool, and N. F. Maxemchuk. A survey of inter-vehicle communication protocols and their applications. *IEEE Communications Surveys Tutorials*, 11(2):3–20, 2009.

# Chapter 11

## Conclusion

The goal of this thesis has been to propose methods to enable the automation of safety and security co-analysis activities in the design phase of safety-critical systems (left side of the V-model). The motivation behind this goal is that many of such activities are performed manually by experts. This manual approach is required due to the lack of explicit semantics associated with key artifacts computed by safety and security activities performed throughout the design of safety-critical systems. We hypothesize that the introduction of lightweight semantics to key artifacts advances the state-of-the-art by enabling the automation of several safety and security activities. The proposed lightweight semantics provide a defined vocabulary, along with its semantics, with uniformly understood meanings in the corresponding domain. Towards achieving the goal of the thesis and validating our hypothesis, we list below some of the research activities that have been carried out:

1. Investigation of which artifacts are key to enable the automation of safety and security co-analysis activities so that we can provide lightweight semantics to those artifacts.
2. Introduction of lightweight semantics to the identified key artifacts.
3. Development of a language enabling the specification of the identified key artifacts.
4. Development of reasoning principles to reason, query, and infer new information based on the developed language.

By incorporating lightweight semantics into key artifacts, i.e., system architecture artifacts, safety and security artifacts, and safety and security architecture patterns, we have proposed a Domain-Specific Language (DSL) to enable the precise specification of such artifacts. Knowledge Representation and Reasoning (KRR) [48] plays a crucial role in automation by providing languages to specify semantics and formalize knowledge representation, along with reasoning rules to enable automation. Our DSL has been developed in a logic programming solver implemented based on KRR. The DSL has been developed following a Model-Based Systems Engineering (MBSE) approach [47] commonly used in the automotive industry.

From a safety perspective, the methods developed in this thesis have successfully automated the recommendation of safety architecture patterns to address safety artifacts,

such as hazards, faults, and failures. Pattern requirements are also automatically computed for each recommended safety architecture pattern. From a security perspective, this thesis adopted the safety-informed security approach [34], where security analysis focuses on the safety artifacts identified in the safety analysis. This thesis aimed to automate Threat Analysis and Risk Assessment (TARA) activities by taking as input safety artifacts identified in the safety analysis. Based on the safety-informed security approach, the methods developed in this thesis have successfully enable the automation of the following TARA activities, namely asset identification (including damage scenarios), impact rating, and threat scenarios. These results highlight the synergies resulting from the combination of safety and security analysis. This thesis has proposed the formalization of an intruder model, which served as the basis to enable the automation of the attack path analysis activity. This thesis has successfully enabled the automated recommendation of security architecture patterns (including pattern requirements) to address threat scenarios. Additionally, the thesis proposes methods to enable the automated computation of consequences arising from safety or security architecture patterns. That is, our methods derive safety artifacts caused by the instantiation of security architecture patterns, and security artifacts caused by the instantiation of safety architecture patterns.

As a result of the developed DSL and reasoning rules, we have developed MBSE plugins for reasoning about the safety and security of system architectures. We demonstrated that the DSL, in conjunction with the reasoning rules, has the potential to act as a backend solution within a MBSE framework. That is, it showcases the viability of using the developed DSL and reasoning rules as integral components of an MBSE framework, providing capabilities for reasoning about the safety and security of system architectures.

The thesis also investigated the benefits of connecting the introduced lightweight semantics with formal verification. Formal verification activities are performed on the right side of the V-model to verify whether requirements specified on the left side of the V-model are met by the developed system [23]. We developed a formal framework to assess the safety and security of Cooperative Adaptive Cruise Control (CACC) systems. We showcase that the pattern requirements, automated by the developed DSL, in conjunction with the reasoning rules, may assist formal verification experts in formalizing security mechanisms. Once security mechanisms have been formalized, formal verification experts are able to verify the system's properties. That is, these experts may rigorously assess and verify the effectiveness of these security mechanisms in ensuring the system's properties.

In conclusion, the proposed lightweight semantics, coupled with the adoption of KRR and MBSE approach, has validated our hypothesis. That is, the introduction of lightweight semantics has paved the way for automated safety and security co-analysis activities performed during the design of safety-critical systems. This research contributes to the field of safety and security co-analysis by providing a foundation for precise specification, interpretation, and decision-making based on the meaning of key artifacts. The integration of the proposed DSL and reasoning rules into MBSE plugins holds promise for reducing the burden on safety and security engineers, ultimately enhancing the efficiency and effectiveness of system development in terms of safety and security.

# Supplementary References

- [1] John C. Knight. Safety critical systems: Challenges and Directions. In *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pages 547–550. ACM, 2002.
- [2] Christian Schlehuber and Dominik Renkel. Merging Worlds - Aligning Safety and Security. In *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - Third International Conference, RSSRail 2019, Lille, France, June 4-6, 2019, Proceedings*, volume 11495 of *Lecture Notes in Computer Science*, pages 284–295. Springer, 2019.
- [3] Xiaobo Qu, Lingshu Zhong, Ziling Zeng, Huizhao Tu, and Xiaopeng Li. Automation and connectivity of electric vehicles: Energy boon or bane? *Cell Reports Physical Science*, 3(8):101002, 2022.
- [4] Paul Rook. Controlling software projects. *Softw. Eng. J.*, 1(1), 1986.
- [5] Tiago Amorim, Helmut Martin, Zhendong Ma, Christoph Schmittner, Daniel Schneider, Georg Macher, Bernhard Winkler, Martin Krammer, and Christian Kreiner. Systematic Pattern Approach for Safety and Security Co-engineering in the Automotive Domain. In *Computer Safety, Reliability, and Security - 36th International Conference, SAFECOMP 2017, Trento, Italy, September 13-15, 2017, Proceedings*, volume 10488 of *Lecture Notes in Computer Science*, pages 329–342. Springer, 2017.
- [6] Martin A. Skoglund, Fredrik Warg, and Behrooz Sangchoolie. In Search of Synergies in a Multi-concern Development Lifecycle: Safety and Cybersecurity. In Barbara Gallina, Amund Skavhaug, Erwin Schoitsch, and Friedemann Bitsch, editors, *Computer Safety, Reliability, and Security - SAFECOMP 2018 Workshops, ASSURE, DECSoS, SASSUR, STRIVE, and WAISE, Västerås, Sweden, September 18, 2018, Proceedings*, volume 11094 of *Lecture Notes in Computer Science*, pages 302–313. Springer, 2018.
- [7] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7, 2006.
- [8] Nuno Silva and Rui Lopes. Electronic Reliability Estimation: How Reliable Are the Results? In *Computer Safety, Reliability, and Security - SAFECOMP 2012 Workshops*:

- Sassur, ASCoMS, DESEC4LCCI, ERCIM/EWICS, IWDE, Magdeburg, Germany, September 25-28, 2012. Proceedings*, volume 7613 of *Lecture Notes in Computer Science*, pages 319–327. Springer, 2012.
- [9] ISO. ISO 26262: Road vehicles — Functional safety — Part 1: Vocabulary, 2018. Available at <https://www.iso.org/standard/43464.html>.
- [10] ISO. ISO 26262: Road vehicles — Functional safety — Part 3: Concept phase, 2018. Available at <https://www.iso.org/standard/43464.html>.
- [11] Nancy G. Leveson and John P. Thomas. *STPA Handbook*. 2018. Available at [https://psas.scripts.mit.edu/home/get\\_file.php?name=STPA\\_handbook.pdf](https://psas.scripts.mit.edu/home/get_file.php?name=STPA_handbook.pdf).
- [12] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004.
- [13] Ashraf Armoush. *Design Patterns for Safety-Critical Embedded Systems*. PhD thesis, RWTH Aachen University, 2010.
- [14] Christopher Preschern, Nermin Kajtazovic, and Christian Kreiner. Safety Architecture Pattern System with Security Aspects. *Trans. Pattern Lang. Program.*, 4:22–75, 2019.
- [15] Vard Antinyan. Revealing the Complexity of Automotive Software. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1525–1528. ACM, 2020.
- [16] Samo Vodopivec, Melita Hajdinjak, Janez Bester, and Andrej Kos. Vehicle interconnection metric and clustering protocol for improved connectivity in vehicular ad hoc networks. *EURASIP J. Wirel. Commun. Netw.*, 2014:170, 2014.
- [17] Stephen D. Gantz Daniel R. Philpott. *FISMA and the Risk Management Framework: The New Practice of Federal Cyber Security*. 2012.
- [18] Mani Amoozadeh, Hui Deng, Chen-Nee Chuah, H. Michael Zhang, and Dipak Ghosal. Platoon management with cooperative adaptive cruise control enabled by VANET. *Veh. Commun.*, 2(2):110–123, 2015.
- [19] Apollo. An Open Autonomous Driving Platform. Available at <https://github.com/ApolloAuto/apollo>.
- [20] Abdullahi Chowdhury, Gour C. Karmakar, Joarder Kamruzzaman, Alireza Jolfaei, and Rajkumar Das. Attacks on Self-Driving Cars and Their Countermeasures: A Survey. *IEEE Access*, 8:207308–207342, 2020.

- [21] WIRED. Hackers remotely kill a jeep on the highway-with me in it, 2015. Available at <https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/>.
- [22] Daniel Zelle, Timm Lauser, Dustin Kern, and Christoph Krauß. Analyzing and Securing SOME/IP Automotive Services with Formal and Practical Methods. In *ARES 2021: The 16th International Conference on Availability, Reliability and Security, Vienna, Austria, August 17-20, 2021*, pages 8:1–8:20. ACM, 2021.
- [23] ISO/SAE. ISO/SAE 21434: Road vehicles — Cybersecurity engineering, 2021. Available at <https://www.iso.org/standard/70918.html>.
- [24] Betty H. C. Cheng, Bradley Doherty, Nick Polanco, and Matthew Pasco. Security Patterns for Automotive Systems. In *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019*, pages 54–63. IEEE, 2019.
- [25] Alexander van den Berghe and Koen Yskout. Security Pattern Catalog. Available at <https://securitypatterns.distrinet-research.be/>.
- [26] Adam Shostack. *Threat Modeling: Designing for Security*. Wiley, 2014.
- [27] Betty H. C. Cheng, Bradley Doherty, Nicholas Polanco, and Matthew Pasco. Security Patterns for Connected and Automated Automotive Systems. *Journal of Automotive Software Engineering*, 1:51–77, 2020.
- [28] Christian Plappert, Florian Fenzl, Roland Rieke, Ilaria Matteucci, Gianpiero Costantino, and Marco De Vincenzi. SECPAT: Security Patterns for Resilient Automotive E / E Architectures. In *30th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2022, Valladolid, Spain, March 9-11, 2022*, pages 255–264. IEEE, 2022.
- [29] Alexander van Den Berghe, Koen Yskout, and Wouter Joosen. A reimagined catalogue of software security patterns. In *Proceedings of the 3rd International Workshop on Engineering and Cybersecurity of Critical Systems, EnCyCriS 2022, Pittsburgh, Pennsylvania, 16 May 2022*, pages 25–32. ACM, 2022.
- [30] ED 202A. Airworthiness security process specification. 2014.
- [31] SAE J3061. Cybersecurity guidebook for cyber-physical vehicle systems. 2016.
- [32] Elena Lisova, Irfan Šljivo, and Aida Čaušević. Safety and Security Co-Analyses: A Systematic Literature Review. *IEEE Systems Journal*, 13(3):2189–2200, 2019.
- [33] Antoaneta Kondeva, Vivek Nigam, Harald Ruess, and Carmen Cărlan. On Computer-Aided Techniques for Supporting Safety and Security Co-Engineering. In *IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops 2019, Berlin, Germany, October 27-30, 2019*, pages 346–353. IEEE, 2019.



- [34] Raj Gautam Dutta, Feng Yu, Teng Zhang, Yaodan Hu, and Yier Jin. Security for Safety: A Path Toward Building Trusted Autonomous Vehicles. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2018, San Diego, CA, USA, November 05-08, 2018*, page 92. ACM, 2018.
- [35] Stéphane Paul. On the Meaning of Security for Safety (S4S). *WIT Transactions on The Built Environment*, 151:379 – 389, 2015.
- [36] Sebastian van de Hoef, Karl Henrik Johansson, and Dimos V. Dimarogonas. Fuel-Efficient En Route Formation of Truck Platoons. *CoRR*, abs/1704.08836, 2017.
- [37] Alessandro Tempia Calvino and Ludovic Apvrille. Direct model-checking of sysml models. In *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2021, Online Streaming, February 8-10, 2021*, pages 216–223. SCITEPRESS, 2021.
- [38] Carolyn L. Talcott, Vivek Nigam, Farhad Arbab, and Tobias Kappé. Formal Specification and Analysis of Robust Adaptive Distributed Cyber-Physical Systems. In *Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems - 16th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2016, Bertinoro, Italy, June 20-24, 2016, Advanced Lectures*, volume 9700 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 2016.
- [39] Yuri Gil Dantas, Antoaneta Kondeva, and Vivek Nigam. Less Manual Work for Safety Engineers: Towards an Automated Safety Reasoning with Safety Patterns. In *Proceedings 36th International Conference on Logic Programming (Technical Communications), ICLP Technical Communications 2020, (Technical Communications) UNICAL, Rende (CS), Italy, 18-24th September 2020*, volume 325 of *EPTCS*, pages 244–257, 2020.
- [40] Yuri Gil Dantas, Tiziano Munaro, Carmen Carlan, Vivek Nigam, Simon Barner, Shiqing Fan, Alexander Pretschner., Ulrich Schoepp, and Sergey Tverdyshev. A Model-based System Engineering Plugin for Safety Architecture Pattern Synthesis. In *Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development - MODELSWARD,*, pages 36–47. INSTICC, SciTePress, 2022.
- [41] Yuri Gil Dantas, Tiziano Munaro, Carmen Cârlan, Vivek Nigam, Simon Barner, Shiqing Fan, Alexander Pretschner, Ulrich Schöpp, and Sergey Tverdyshev. A Toolchain for Synthesizing and Validating Safety Architectures. *SN Comput. Sci.*, 4(4):335, 2023.
- [42] Yuri Gil Dantas, Simon Barner, Pei Ke, Vivek Nigam, and Ulrich Schöpp. Automating Vehicle SOA Threat Analysis Using a Model-Based Methodology. In *Proceedings of the 9th International Conference on Information Systems Security and Privacy, Lisbon, Portugal, February 22-24, 2023*, pages 180–191. SciTePress, 2023.
- [43] Yuri Gil Dantas and Ulrich Schöpp. SeCloud: Computer-Aided Support for Selecting Security Measures for Cloud Architectures. In *Proceedings of the 9th International*

- Conference on Information Systems Security and Privacy, Lisbon, Portugal, February 22-24, 2023*, pages 264–275. SciTePress, 2023.
- [44] Yuri Gil Dantas, Vivek Nigam, and Ulrich Schöpp. A Model-Based Systems Engineering Plugin for Cloud Security Architecture Design. *SN Comput. Sci.*, 5(5):553, 2024.
- [45] Yuri Gil Dantas and Vivek Nigam. Automating Safety and Security Co-design through Semantically Rich Architecture Patterns. *ACM Trans. Cyber Phys. Syst.*, 7(1):5:1–5:28, 2023.
- [46] Yuri Gil Dantas, Vivek Nigam, and Carolyn Talcott. A Formal Security Assessment Framework for Cooperative Adaptive Cruise Control. In *IEEE Vehicular Networking Conference (VNC)*, 2020.
- [47] INCOSE. Systems engineering vision 2020, 2007. Available at [http://www.ccose.org/media/upload/SEVision2020\\_20071003\\_v2\\_03.pdf](http://www.ccose.org/media/upload/SEVision2020_20071003_v2_03.pdf).
- [48] Ronald Brachman and Hector Levesque. *Knowledge Representation and Reasoning*. Elsevier, 2004.
- [49] fortiss GmbH. AutoFOCUS 2.21, 2022. Available at <https://www.fortiss.org/en/publications/software/autofocus-3>.
- [50] Michael Gelfond and Vladimir Lifschitz. Logic Programs with Classical Negation. In *Logic Programming, Proceedings of the Seventh International Conference, Jerusalem, Israel, June 18-20, 1990*, pages 579–597. MIT Press, 1990.
- [51] Potassco project. Clingo: A grounder and solver for logic programs <https://github.com/potassco/clingo>.
- [52] Ullrich Hustadt, Boris Motik, and Ulrike Sattler. Reasoning in Description Logics with a Concrete Domain in the Framework of Resolution. In *Proceedings of the 16th European Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 353–357. IOS Press, 2004.
- [53] Stephan Grimm. Knowledge Representation and Ontologies. In *Scientific Data Mining and Knowledge Discovery - Principles and Foundations*, pages 111–137. Springer, 2010.
- [54] Jeannette M. Wing. A Specifier’s Introduction to Formal Methods. *Computer*, 23(9):8–24, 1990.
- [55] Edmund M. Clarke, Orna Grumberg, Doron Peled, and Doron A. Peled. *Model Checking*. MIT Press, 1999.
- [56] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude: A High-Performance Logical Framework*. LNCS. Springer, 2007.

- [57] Peter Csaba Ölveczky. Teaching formal methods to undergraduate students using maude. In *Rewriting Logic and Its Applications - 14th International Workshop, WRLA@ETAPS 2022, Munich, Germany, April 2-3, 2022, Revised Selected Papers*, volume 13252 of *Lecture Notes in Computer Science*, pages 85–110. Springer, 2022.
- [58] Emmanuel Clement and Antoine Rauzy. A Fault Tree Assessment Software Fully Compliant with Open-PSA and using XFTA Fault Tree Engine. Available at <https://www.arbre-analyste.fr/en.html>, 2014.
- [59] Asim Abdulkhaleq. XSTAMPP (eXtensible STAMP Platform). Available at <https://github.com/SE-Stuttgart/XSTAMPP>, 2018.
- [60] Yuri Gil Dantas, Vivek Nigam, and Harald Ruess. Security Engineering for ISO 21434. *CoRR*, abs/2012.15080, 2020.
- [61] Christopher Preschern, Nermin Kajtazovic, and Christian Kreiner. Security Analysis of Safety Patterns. In *Proceedings of the 20th Conference on Pattern Languages of Programs, PLoP '13, USA, 2013*. The Hillside Group.
- [62] United Nations. UN Regulation No. 155 - Cyber security and cyber security management system. 2021.
- [63] Giedre Sabaliauskaite, Lin Shen Liew, and Jin CuiJin Cui. Integrating Autonomous Vehicle Safety and Security Analysis using STPA Method and the Six-Step Model. In *International Journal of Information Security*, pages 160–169, 2018.
- [64] Giedre Sabaliauskaite, Sridhar Adepu, and Aditya Mathur. A Six-Step Model for Safety and Security Analysis of Cyber-Physical Systems. In *Critical Information Infrastructures Security - 11th International Conference, CRITIS 2016, Paris, France, October 10-12, 2016, Revised Selected Papers*, volume 10242 of *Lecture Notes in Computer Science*, pages 189–200. Springer, 2016.
- [65] Georg Macher, Harald Sporer, Reinhard Berlach, Eric Armengaud, and Christian Kreiner. SAHARA: a security-aware hazard and risk analysis method. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*, pages 621–624. ACM, 2015.
- [66] David Förster, Claudia Loderhose, Thomas Bruckschlögl, and Franziska Wiemer. Safety goals in vehicle security analyses: a method to assess malicious attacks with safety impact. In *the 17th escar Europe - Embedded Security in Cars*, 2019.
- [67] Junjie Shen, Ningfei Wang, Ziwen Wan, Yunpeng Luo, Takami Sato, Zhisheng Hu, Xinyang Zhang, Shengjian Guo, Zhenyu Zhong, Kang Li, Ziming Zhao, Chunming Qiao, and Qi Alfred Chen. SoK: On the Semantic AI Security in Autonomous Driving. *CoRR*, abs/2203.05314, 2022.

- [68] Junjie Shen, Jun Yeon Won, Zeyuan Chen, and Qi Alfred Chen. Drift with Devil: Security of Multi-Sensor Fusion based Localization in High-Level Autonomous Driving under GPS Spoofing. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 931–948. USENIX Association, 2020.
- [69] Zhongyuan Hau, Kenneth T. Co, Soteris Demetriou, and Emil C. Lupu. Object removal attacks on lidar-based 3d object detectors. *CoRR*, abs/2102.03722, 2021.
- [70] Daniel Rezvani. Hacking Automotive Ethernet Cameras. Available at <https://argus-sec.com/blog/cyber-security-blog/hacking-automotive-ethernet-cameras/>.
- [71] Saurabh Jha, Shengkun Cui, Subho S. Banerjee, James Cyriac, Timothy Tsai, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. ML-Driven Malware that Targets AV Safety. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*, pages 113–124. IEEE, 2020.
- [72] Rony Komissarov and Avishai Wool. Spoofing Attacks Against Vehicular FMCW Radar. In *ASHES@CCS 2021: Proceedings of the 5th Workshop on Attacks and Solutions in Hardware Security, Virtual Event, Republic of Korea, 19 November 2021*, pages 91–97. ACM, 2021.
- [73] David Ke Hong, John Kloosterman, Yuqi Jin, Yulong Cao, Qi Alfred Chen, Scott A. Mahlke, and Z. Morley Mao. AVGuardian: Detecting and Mitigating Publish-Subscribe Overprivilege for Autonomous Vehicle Systems. In *IEEE European Symposium on Security and Privacy, EuroSecP 2020, Genoa, Italy, September 7-11, 2020*, pages 445–459. IEEE, 2020.
- [74] R. Hammett. Design by extrapolation: an evaluation of fault-tolerant avionics. In *20th DASC. 20th Digital Avionics Systems Conference (Cat. No.01CH37219)*, volume 1, pages 1C5/1–1C5/12 vol.1, 2001.
- [75] Massimo Baleani, Alberto Ferrari, Leonardo Mangeruca, Alberto L. Sangiovanni-Vincentelli, Maurizio Peri, and Saverio Pezzini. Fault-tolerant platforms for automotive safety-critical applications. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES 2003, San Jose, California, USA, October 30 - November 1, 2003*, pages 170–177. ACM, 2003.
- [76] Paul Hampton. Survey of Safety Architectural Patterns. In *Achieving Systems Safety - Proceedings of the Twentieth Safety-Critical Systems Symposium, SSS 2012, Bristol, UK, February 7-9, 2012*, pages 137–158. Springer, 2012.
- [77] Eduardo B. Fernandez. *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*. John Wiley & Sons, 2013.

- [78] Alexander van Den Berghe, Koen Yskout, and Wouter Joosen. Security patterns 2.0: towards security patterns based on security building blocks. In *Proceedings of the 1st International Workshop on Security Awareness from Design to Deployment, SEAD@ICSE 2018, Gothenburg, Sweden, May 27, 2018*, pages 45–48. ACM, 2018.