Automating Safety and Security Co-Design through Semantically-Rich Architecture Patterns

YURI GIL DANTAS, fortiss GmbH, Germany

VIVEK NIGAM, Federal University of Paraíba & Huawei Technologies Düsseldorf GmbH, Brazil & 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 patterns has consequences on security, e.g., the deployment of a safety pattern may lead to new threats. The other way around may also be possible, i.e., the deployment of a security pattern may lead to new failures. Safety and security co-design is, therefore, required to understand such consequences and trade-offs, in order to reach appropriate system designs. Currently, pattern descriptions, including their consequences, are described using natural language. Therefore, their deployment in system design is carried out manually, thus 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: • **Computer architectures** → Safety&Security; • **Logic** → Knowledge Representation and Reasoning.

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 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 can attack such communication channels to infiltrate vehicles and, e.g., disable safety functions thus reducing passenger safety [34] or even causing accidents [10].

Authors' addresses: Yuri Gil Dantas, fortiss GmbH, Munich, Germany, dantas@fortiss.org; Vivek Nigam, Federal University of Paraíba & Huawei Technologies Düsseldorf GmbH, João Pessoa & Düsseldorf, Brazil & Germany.

111

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 trade-offs 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 patterns include Watchdog and Monitor Actuator, and examples of security patterns include Firewall and Security Monitor.

Currently, however, 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 to other aspects [27]. Moreover, these consequences are context dependent. It may be that placing the same type of pattern in one location of the system architecture may have serious consequences, while placing the same pattern in another locations 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 patterns informally using natural language, a better approach is to provide more precise semantics by using Domain-Specific Languages (DSL).[1] *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.

This paper's main goal is to enable safety and security co-design automation by using semantically-rich architecture patterns. Our main contributions are as follows:

- **Domain-Specific Language (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 safety and security patterns template that contains semantic information provided by the proposed DSL. Due to space restrictions, we describe in the paper only four such patterns, two for safety and two for security. Notice that our machinery (described below) currently supports ten patterns [8]. The supported patterns are Acceptance Voting, Homogeneous Duplex, Heterogeneous Duplex,

---

[1]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.
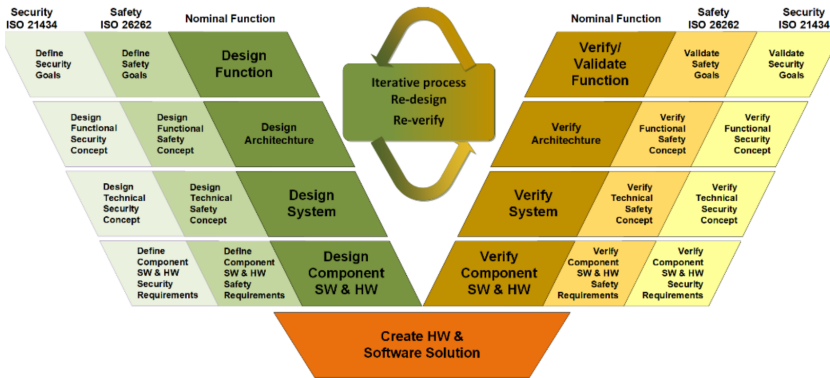
Fig. 1. Integrated V-Model for System, Safety [32].

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 pattern fails operational. We specify security reasoning principles based on formal intruder models, e.g., path reachability used to determine whether an attack can pose a threat to some sub-component from outside the system, and when a security 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 pattern may cause safety failures and when safety patterns can be targeted by intruders so 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, 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 paper by SafSecPat. SafSecPat is capable of automating several activities that are currently carried out manually. SafSecPat is publicly available in [8].

The remainder of this paper is organized as follows: Section 2 reviews basic safety and security concepts, the V-model used in vehicle system development, and templates used for describing patterns. Section 3 describes the running example taken from ISO 21434. It also illustrate 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 to 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 paper. 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 definition 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 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 *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 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 MCSes 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* are means to avoid service failures in the presence of faults. A *safety pattern* (described in further details in Section 2.1) is a system architecture solution that is known to provide some level of fault-tolerance.

Basic Security Concepts: The definition 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 involving a vehicle or vehicle function (e.g., an asset) and affecting a road user. An *attack* is an 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[2] 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 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 subject of safety and security analysis by providing details such as the item's preliminary architecture and operational conditions. Then safety and security analysis are 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, 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[3], Attack Trees [31] and path 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.

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, D, where QM

---

[2]The ISO 21434 define a path as a set and not as sequence. We will use here as a sequence.
[3]The threats considered by STRIDE are Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege.

has no safety criticality, while A,B,C,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 fail silent function shall necessarily lead to the loss of the functionality (e.g., shutdown 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. Moreover, as we will describe below with the architecture patterns, it is possible to effectively detect when a function is not longer functioning and switch to a safe mode and trigger appropriate measures, e.g., inform the driver.
- **Fail-Operational** is the strongest type of fault tolerance as it requires that a function operates 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, so to comply, for example, with the level of safety criticality established and security properties required. Typically, safety and security engineers make use architecture solutions, called architecture patterns.

Architecture patterns are abstract solutions to recurrent system problems such as safety and security.[4] 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 independent of implementation details. Their description make 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 patterns 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, 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 literature [1, 29].

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.

Table 1. Architecture Pattern Description Template

---

[4]Other measures like testing and established coding practices may be used in addition or instead of architecture patterns.

Since patterns are commonly used, and many times using different names, 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 for; 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 to other concerns, such as security, performance, reliability, 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, as well as the results of a safety analysis of the headlamp system.
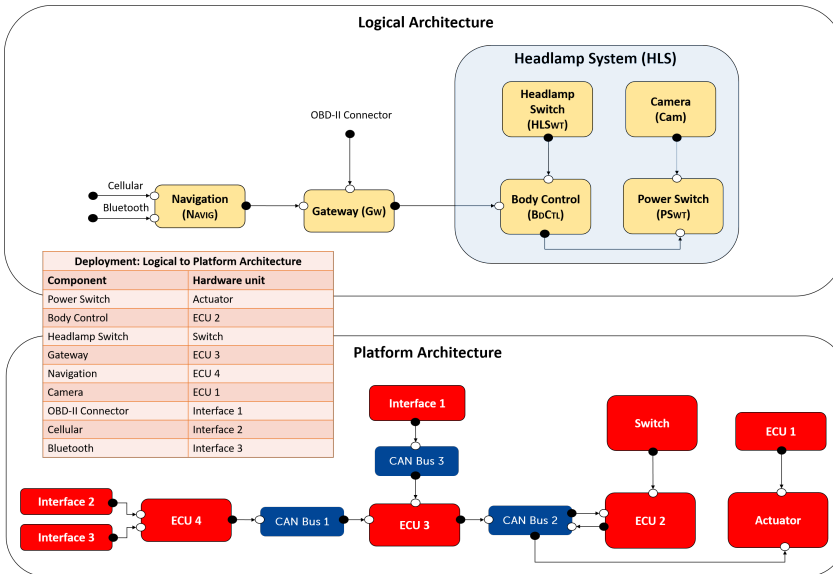

Fig. 2. Architecture of the headlamp system

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].

Hazard	Description	ASIL
HZ1	Headlamp turn off unintentionally during night driving.	C
HZ2	Unintended low beam of headlamp when no oncoming vehicle is detected.	A

Table 2. Identified hazard for the headlamp system

Figure 2 depicts both the logical and the platform 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., PwrSwt, 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.

A Camera (Cam) detects oncoming vehicles and sends signals to the Power Switch (PwrSwt). The Body Control (BdCtl) sends signals to PwrSwt. These signals are requests from the driver (coming from the Headlamp Switch – HlSwt) to turn the headlamp on or off. Note that the channel from BdCtl to PwrSwt 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 (Gw) that control access from other components located outside the headlamp system, e.g., Navigation (Navig). Gw receives signals from an OBD-II Connector (OBDConn). Navig has two interfaces, namely Cellular (Cell) and Bluetooth (Bt). Both Cell and Bt interfaces and OBDConn may access BdCtl to, e.g., carry out software updates. Note that in the platform architecture the channel from Navig to Gw is deployed to CAN Bus 1, the channel from Gw to BdCtl is deployed to CAN Bus 2, and the channel from OBDConn to Gw 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. 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.

- **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**.

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

Table 3. Safety goals to prevent the presence of identified hazards

Since all of these failures can lead independently to the hazard, the minimal cut sets are {**FL1**}, {**FL2**}, and {**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 1st failure of type erroneous. The system shall transition to a safe state always after the 2nd 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].[5] 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 can easily be carried out provided 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 can 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. Based on this threat model, the attack paths are enumerated.

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

---

[5]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 paper.

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.

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 Domain-Specific Language (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 paper 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 to grasp the developed SafSecPat.

4.1 Knowledge Bases and Answer-Set Programming

Knowledge Representation and Reasoning (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 $a_1 \vee \cdots \vee a_m \leftarrow \ell_1, \ldots, \ell_n$ or where $\ell, \ell_1, \ldots, \ell_n$ 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.[6]

Consider the program P_1 with the following two rules:

$$a \vee b \qquad c \leftarrow a$$

P_1 has two answer-sets $\{a, c\}$ and $\{b\}$. Intuitively, each answer-set is a minimal model of the logic program, i.e., 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 paper, we use the DLV notation writing :- for $\leftarrow$ and v for $\vee$, e.g., the

---

[6]We refer to [3] for the precise formal semantics of logic programs.

Fact	Description
cp(id)	id is a component in the system.
subcp(id$_1$,id$_2$)	id$_1$ is a sub-component of component id$_2$.
ch(id,id$_1$,id$_2$)	id is a logical channel connecting an output of component id$_1$ to an input of component id$_2$. Notice that it denotes a unidirectional connection.
if(id, [ch$_1$,…,ch$_n$])	id is an information flow following the channels in [ch$_1$,…,ch$_n$].
ecu(id)	id is an Electronic Computing Unit (ECU) that can run components.
can(id)	id is a Controller Area Network (CAN) to communicate between ECUs.
interface(id)	id is a hardware interface that allows the connection of external peripheral to components.
dep(id,id$_d$)	Component id is deployed (i.e., executed) in ECU id$_d$ or in interface id$_d$, or alternatively, the logical channel id is deployed in CAN id$_d$ to establish the communication between id's components.

Table 4. SafSecPat: SafSecPat: Language for (selected) architecture elements

program P_1 is written as a ∨ 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 Domain-Specific Language (DSL) for embedded systems. This DSL enables the specification of architecture elements, safety and security elements, and architecture patterns. These elements 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(cmpsa,cam,ps). ch(hsbd,hlSwt,bdCtl). ch(bcps,bdCtl,ps). ch(gwbc,gw,bdCtl).
ch(navgw,navig,gw). ch(obdgw,obdC,gw). ch(celnav,cell,navig). ch(btnav,bt,navig).
if(if1,[cmps]). if(if2,[hsbd,bcps]). if(if3,[obdgw,gwbc,bcps]).
if(if4,[celnav,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(bcpsa,bdCtl,ps) denotes the logical communication channel bcpsa between the Body Control and the Power Switch. The information flow if2 denotes data flows from channel hsbd to channel bcps.

A user may specify the hardware units of the 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.

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}.
$mcs(id_{msc},[id_{fl}])$	id_{msc} is a minimal cut set consisting of failure(s) id_{fl}.
$lmcs2hz([id_{msc}],id_{hz})$	id_{msc} 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 a 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 $pt_{sv} \in \{neg, maj, mod, sev\}$.
$reachI(id_{cp},id_{hw}, P)$	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).

Table 5. SAFSECPAT: Language for safety and security elements

```
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)` denotes 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(bdCtl,ecu2). dep(gw,ecu3). dep(navig,ecu4). dep(navgw,can1).
dep(gwbc,can2). dep(bcps,can2). dep(obdgw,can3). dep(cell,int1). dep(bt,int2).
dep(obdC,int3). dep(celnav,wl). dep(btnav,wl).
```

The fact `dep(bdCtl,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 Elements.
Our DSL consists of safety and security elements described in Table 5. By *safety elements*, we refer to safety goals, hazards, faults, failures, and minimal cut

sets. By *security elements*, we refer to potential threat and threat scenarios. [7] 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. This has a great impact on the precision of SafSecPat in recommending safety patterns, as, e.g., only a sub-set 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 1st failure has been detected. According to [25][18], 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 1st 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 such results in our DSL as follows:

```
hz(hz1,[hls,s3,e4,c3]). hz(hz2,[hls,s3,e3,c2]). ft(ft1,[bdCtl]). fl(fl1,[err]).
sg(sg1,[hz1,all1fail,never,all2fail]). sg(sg2,[hz2,never,most1fail,never]).
ft(ft2,[bcps]). ft2fl(ft1,fl1). fl(fl2,[loss]). ft2fl(ft2,fl2). ft(ft3,[cam]).
fl(fl3,[err]). ft2fl(ft3,fl3). mcs(mcs1,[fl1]). mcs(mcs2,[fl2]).
lmcs2hz([mcs1,mcs2],hz1). mcs(mcs3,[fl3]). lmcs2hz([mcs3],hz2).
```

The facts hz(hz1,[hls,s3,e4,c3]) and hz(hz2,[hls,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,[bdCtl]) 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(fl1,[err]) of type erroneous, and the fault ft2 triggers failure fl(fl2,[loss]) of type loss. The minimal cut set mcs1 consists of failure fl1 and mcs2 consists of failure fl2. Either minimal cut set mcs1 or mcs2 leads to hazard hz1 (as specified by the fact lmcs2hz([mcs1,mcs2],hz1)).

Security-wise, we expect that a user (e.g., a security engineer) provides all public elements as input to SafSecPat. The considered public elements for the headlamp system is 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 can be derived by reasoning rules, as demonstrated in Section 6.

---

[7]Our DSL enables the specification of further security elements such as damage scenario and risk determination as shown in [9]. We omit these security elements for the sake of presentation given that our focus is on safety and security co-design using architecture patterns.

Fact	Description
safetyPattern(id,[name, [cp], [inp],[int],[out]])	name is a safety pattern of ID id. This patter consists of a list of components (e.g., redundant components) cp. The last three parameters inp, int and out denote, respectively, the input, the internal and the output channels related to the pattern.
safetyIntent(name,[[fl_{tp}], asil, f_{op},f_{sl},f_{sf}])	name is a safety pattern suitable for avoiding failures of type fl_{tp}, where $fl_{tp} \in$ {err,loss}. ASIL asil denotes to which ASIL the pattern is suitable to be applied, where asil $\in$ {a,b,c,d}. Pattern name ensures the system to either fail operational (f_{op}), fail silent (f_{sl}), or fail safe (f_{sf}).
securityPattern(id,[name, [cp],[inp],[out]])	name is a security pattern of ID id. This patter 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.
securityIntent(name,[[th_{tp}]])	name is a safety pattern suitable for mitigating threats of type th_{tp}, where $th_{tp} \in$ {con, int, avl}.

Table 6. SafSecPat: Language for safety and security architecture patterns

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 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 includes 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.

5 SPECIFICATION OF SAFETY ARCHITECTURE PATTERNS

This section illustrates how we can use SafSecPat to provide semantically-rich description of safety 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 different design implementation in comparison to the components from the first pair.

	Description	SafSecPat Specification
Pattern name	Dual self-checking pair pattern with fail safe	NAME=dualSelfCheckingPairFS;
Structure		COMPONENT=[pr1,se2,fd1,pr2,se2,fd2]; INPUT_CH=[inp1,inp2,inp3,inp4]; INTERNAL_CH=[int1,int2,int3,int4,int5]; 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 transition the system to a safe state always after the 2nd failure has been detected. This pattern never fails silent.	TYPE_FAIL=[err]; ASIL=d; FAIL_OP=all1fail; FAIL_SILENT=never; 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.	TYPE_ASSUMPTION=are_independent; COMPONENT=[pr1,se1,pr2,se2];
	The the first and second pair shall be allocated to dedicated hardware units.	TYPE_ASSUMPTION=are_decoupled; COMPONENT=[pr1,se1,pr2,se2];
	The fault detector shall be verified.	TYPE_ASSUMPTION=are_verified; 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 **int**egrity if the failure associated with the primary component is of type erroneous.	COMPONENT=[fd1,fd2]; TYPE_THREAT=int;

Table 7. Dual self-checking pair pattern with fail safe

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 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 transition the system to a safe state. This pattern fails operational always after the 1st failure of type erroneous (i.e., failure on the first pair). It fails safe always after the 2nd 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. These assumptions will be created whenever this 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,_],_,_,_]).
```

5.2 Monitor-Actuator Pattern

The monitor-actuator pattern [1] consists of a primary component and a monitor. The monitor consumes the computations from the primary component and its original inputs such that it 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 1st failure 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.

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. This assumption will be created whenever a monitor-actuator pattern is instantiated.

```
assumption(monitorActuator,are_verified,[mon])
 :- safetyPattern(idpat,[monitorActuator,[_,mon],_,_,_]).
```

6 SPECIFICATION OF SECURITY ARCHITECTURE PATTERNS

As in the previous section, this section illustrates how we can use SAFSECPAT to provide semantically-rich description of security 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.

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 component. 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

	Description	SᴀꜰSᴇᴄPᴀᴛ Specification
Pattern name	Monitor-Actuator Pattern	`NAME=monitorActuator;`

Structure		`COMPONENT=[pr,mon];`
		`INPUT_CH=[inp1,inp2];`
		`INTERNAL_CH=[int1,shut];`
		`OUTPUT_CH=[out];`

Intent	This pattern is suitable for low critical-ity 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];`
		`ASIL=b;`
		`FAIL_OP=never;`
		`FAIL_SILENT=all1fail;`
		`FAIL_SAFE=never;`

Problem addressed	This pattern tolerates faults by avoiding failures of type erroneous.	
Assumptions	The monitor shall be verified.	`TYPE_ASSUMPTION=are_verified;` `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 **int**egrity if the failure associated with the primary component is of type erroneous.	`COMPONENT=[mon];` `TYPE_THREAT=int;`

Table 8. Monitor-Actuator pattern

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 SᴀꜰSᴇᴄPᴀᴛ. 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,[[ava,int]]).
```

The first assumption for the firewall pattern from Table 9 is specified as follows.

```
assumption(firewall,are_verified,[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 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].

	Description	SAFSECPAT Specification
Pattern name	Firewall	`NAME=firewall;`

Structure		`COMPONENT=[bus,fw,pr];`
		`INPUT_CH=[inp1,inp2];`
		`OUTPUT_CH=[out1,out2];`

Intent	This pattern intercepts, filters, and blocks incoming and outgoing messages in the network layer.	`TYPE_THREAT=[ava,int];`
Problem addressed	This pattern mitigates threats that violates the availability and integrity of the system.	
Assumptions	Firewall shall be verified	`TYPE_ASSUMPTION=are_verified;` `COMPONENT=[fw];`
	Security policies shall be specified.	`TYPE_ASSUMPTION=have_policies;` `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];` `TYPE_FAILURE=err;`

Table 9. Firewall pattern

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

Building on the DSL presented in Section 4, this section describes some 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 (a) a failure can be avoided, (b) a minimal cut set can be avoided, (c) a fault can be tolerated, (d) a hazard can be controlled, and (e) a safety goal can be satisfied. We introduce five new facts to specify safety reasoning principles for (a), (b), (c), (d) and (e). Note that four of the new facts receive attributes from the safety pattern intent as argument in order to make explicit how they have been

	Description	SᴀꜰSᴇᴄPᴀᴛ Specification
Pattern name	Security Monitor	`NAME=securityMonitor;`

Structure		`COMPONENT=[pr,mon];`
		`INPUT_CH=[int1,int2];`
		`INTERNAL_CH=[int1,shut];`
		`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	`TYPE_ASSUMPTION=are_verified;` `COMPONENT=[mon];`
	Security policies shall be specified.	`TYPE_ASSUMPTION=have_policies;` `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];` `TYPE_FAILURE=err;`

Table 10. Security monitor pattern

addressed by the pattern. These attributes consists 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 1st 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]) :- fl(IDFL,[TYPE]),
    ft(IDFT,[TARGET]), ft2fl(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. Both the rule for `tol` and for `avoidedMCS` are omitted here. Next, we specify a rule for hazard controllability.

```
ctl(IDHZ,ATTRSINTENT) :- hz(IDHZ,ATTRSHZ),
    lmcs2hz(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 leads 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 Patterns. We now introduce our reasoning rule for recommending which safety patterns could be used at which place of the system architecture to avoid failures provided as input information by the user. This is done by using the following disjunctive 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]])
      :- fl(IDFL,[FLTYPE]), ft(IDFT,[TARGET]), ft2fl(IDFT,IDFL),
      exploreSafPat(PAT), safetyIntent(PAT,[PATFLTYPE | ATTRSINTENT]),
      #member(FLTYPE,PATFLTYPE), getSafIntentASIL(PAT,PATASIL), mcs(IDMCS,FAILURES),
      #member(IDFL,FAILURES), lmcs2hz(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 lead 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 recommend 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 2 distinct patterns avoid the same failure).

7.1.2 Safety Pattern Recommendation for the Headlamp System. We now apply SafSecPat to the headlamp system described in Section 3 to automatically select which safety 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 (a) avoiding the failures leading to hazards HZ1 and HZ2, and (b) 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 1st failure, and it fails safe always after the 2nd 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 1st failure. The pattern assumptions generated by SafSecPat can help engineers to deploy the pattern-related components into the platform architecture.
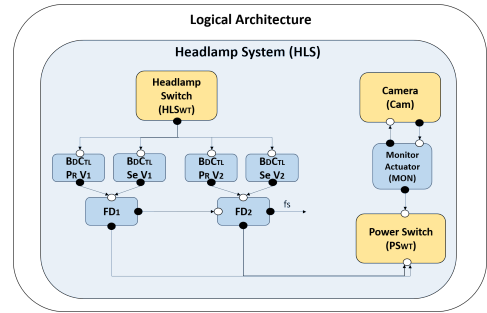

Fig. 3. Headlamp system with patterns

7.2 Security Reasoning

This paper 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 (a) a potential threat becomes a threat, and (b) a threat can be mitigated by a security pattern.

Before introducing these reasoning principles, we first introduce our threat model.

7.2.1 Threat Model. The threat model we assume is 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 to, i.e., information that appears in a channel reachable from a public interface that is not encrypted or encrypted with a key that she 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 hardware that is deployed in a 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 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
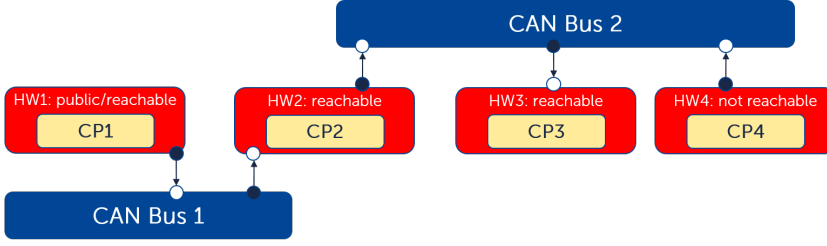
Fig. 4. Illustration of the intruder reachability

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 easily 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).
reachI(CH,CAN,[CAN|PATH]) :- writesToCan(CP,CAN), dep(CH,CAN),
      dep(CP,HWCP), reachI(CP,HWCP,PATH), not #member(CAN,PATH).
```

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 messages exchanged, the intruder model shown above does not take into account the fact that messages are encrypted or not nor does it take into account on how the messages exchanged 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 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 paper 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 on severity 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 and of severity sev. A failure of type loss that leads to a hazard of severity S2 is mapped to a pThreat of type ava 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]) :- fl(IDFL,[SAFTYPE]), ft(IDFT,[TARGET]),
  ft2fl(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 fl1 and hazard hz1 identified in Section 3. Failure fl1 that leads to hazard hz1 is mapped to the potential threat pt1. The potential threats becomes 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).

$$pThreat(pt1,[bdCtl,ecu2,err,sev]).$$
$$threat([pt1,[ecu2,can2,ecu3,can1,ecu4,int3],[bdCtl,ecu2,err,sev]).$$

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 for mitigating the type of threat violated by IDTH is placed in the architecture.

7.2.3 *Recommendation of Security Patterns.*

We introduce our reasoning rule for recommending which security patterns could be used at which place of the system architecture to mitigate threats provided as input information by the user or derived by safety elements. 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.[8] 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
xnsecurityPattern([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 Pattern Recommendation for the Headlamp System.*

We now apply SAFSECPAT to the headlamp system described in Section 3 to automatically select which security 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

---

[8]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".
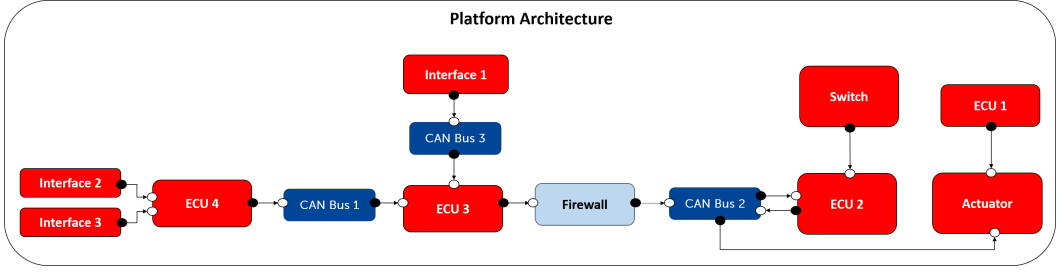
Fig. 5. Headlamp system with security pattern.

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]],[bdCtl,ecu2,int,sev]).
threat([pt2,[can2,ecu3,can3,int1]],[bcps,can2,ava,sev]).
```

Our goal is to provide security patterns suitable for mitigating these threats that violate the availability (ava) 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 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 paper proposes the use of KRR for safety and security co-analysis with architecture patters. The goal is to provide automated methods to reason about the consequences of safety patterns to security, and of security patterns to safety.

7.3.1 Security Consequences Caused by Safety 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]),
    fl(IDFL,[FAILTYPE]), ft2fl(IDFT,IDFL), dep(CP,CHECKER), lmcs2hz(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 public elements. 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 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 specifies 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],_,_,_]).
fl(IDFL,[err]) :- securityPattern(IDPAT,[firewall,[_,_,FW],_,_,_]),
      ft(IDPAT,[FW]), createID(IDPAT,firewall,IDFL).
ft2fl(IDPAT,IDFL) :- securityPattern(IDPAT,[firewall,[_,_,FW],_,_,_]),
      ft(IDPAT,[FW]), fl(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 security pattern triggers an identified hazard. Figure 6 illustrates a cascading failure caused by the deployment of a firewall.

The behavior of component CP2 depends on the signals sent by component CP1, and there is fault associated to CP2 that triggers failures leading to an identified hazard HZ. If the deployed Firewall erroneously blocks messages from CP1 then the failures on CP1 might occur. As a result, the failures from the Firewall might as well lead to hazard HZ.
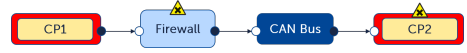

Fig. 6. Illustration of a cascading failure due to the deployment of a firewall

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 effecting the Body Control (e.g., when a user attempts to perform a software update via Bluetooth). 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 scenarios (1) or (2) are 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. 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* that selects a component of the architecture that a safety pattern shall

be added. (2) *Pattern selection* that selects a pattern from a pre-defined library of patterns, and (3) *Component substitution* that 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 enables 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 assuring 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 also hold. Relying on the instantiation of a pattern template that includes both assumptions and properties (similar to the template used by this paper), 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 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] that 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 do not address 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 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. In particular, we have proposed a Domain-Specific Language (DSL), called SafSecPat, that includes several safety and security concepts. We demonstrated its use in the description of several well-known patterns. Moreover, 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, 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 paper [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 elements (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 ten minutes to compute all solutions. However, given that the focus of SafSecPat is on development 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).
[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 2019, Munich, Germany, September 15-20, 2019*. 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 Carlan, Vivek Nigam, Simon Barner, Shiqing Fan, Alexander Pretschner., Ulrich Schoepp, 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,*. INSTICC, SciTePress, 36–47. https://doi.org/10.5220/0010831700003119
[8] Yuri Gil Dantas and Vivek Nigam. 2021. https://github.com/ygdantas/safsecpat. (2021).
[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)*.
[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* (2018).
[15] fortiss GmbH. 2020. *AutoFOCUS 2.19*. https://www.fortiss.org/en/publications/software/autofocus-3 Available at 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 DASC. 20th Digital Avionics Systems Conference (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. (2018). Available at https://www.iso.org/standard/43464.html.

[20] ISO/SAE AWI 21434. 2020. Road Vehicles - Cybersecurity engineering. (2020).
[21] John C. Knight. 2002. Safety critical systems: challenges and directions. In *ICSE*, 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 *(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. Available at https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/.