Less Manual Work for Safety Engineers: Towards an Automated Safety Reasoning with Safety Patterns (Application Paper)

Yuri Gil Dantas, Antoaneta Kondeva, and Vivek Nigam
fortiss GmbH, Germany
(*e-mail:* {dantas,kondeva,nigam}@fortiss.org)

Abstract

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

1 Introduction

The development of safety-critical systems, such as vehicles, aircraft and medical devices aims to achieve two goals: (1) to develop systems that cannot cause any harm, and (2) to convince regulatory bodies about the safeness of the system by demonstrating compliance to safety standards (iso 2018; arp 2018).

To achieve the first goal, safety engineers perform safety analyses to ensure that systems cannot cause any harm. For example, *Hazard Analysis* (iso 2018; ar4 1996) identifies the main hazards that shall be controlled. Other safety techniques, *e.g.*, FTA (ar4 1996), STPA (Leveson and Thomas 2018), FMEA (ar4 1996), HAZOP (Crawley and Tyler 2015), break down the identified main hazards into component hazards (a.k.a component failures), *i.e.*, faults that can trigger main hazards. Safety engineers commonly use *safety architectural patterns* (Preschern et al. 2013; Martin et al. 2020; Matos et al. 2014) to control the identified component hazards (or hazards for short) thus controlling the main hazards. To achieve the second goal, safety engineers shall develop a safety case (iso 2018; def 2007) for the system under development. The purpose of the safety case is to both (a) ensure that all hazards have been analyzed and (b) answer why a safety pattern has been deployed at a particular component to control which hazard.

Safety cases are often documented in textual form, or by models such as the Goal Structure

Notation (GSN) (gsn 2011). These models, however, have limited support for automated reasoning (Kondeva et al. 2019). It is not possible to automatically check whether safety arguments used in a safety case are correct, *i.e.*, check whether all hazards have been properly controlled by, *e.g.*, safety patterns. This is because the *safety reasoning* used to support system safety is implicitly written textually thus lacking the precise semantics to enable automation (Nigam et al. 2018). As a result, correctness checks are performed manually, possibly leading to human errors.

Our vision is to build an incremental development process for system safety and security assurance cases using automated methods that incorporate safety and security reasoning principles. This paper is the first step towards achieving this vision. We provide safety reasoning principles with safety patterns used during the definition of system architecture for embedded systems. We specify these principles using logic and logic programming as they are suitable frameworks for the specification of reasoning principles as knowledge bases and using them for automated reasoning (Baral 2010).

Our main contributions are threefold:

- **Domain-Specific Language (DSL):** We propose a DSL for safety reasoning with safety patterns. Our DSL includes (1) architectural elements, both functional components and logical communication channels; (2) safety hazards including guidewords used in typical analysis, *e.g.*, erroneous or loss of function; (3) a number of safety patterns including n-version programming, safety monitors, and watchdogs;
- **Reasoning Principles:** We specify key reasoning principles for determining when a hazard can be controlled or not, including reasoning principles used to decide when a safety pattern can be used to control a hazard. These reasoning principles are specified as Disjunctive Logic Programs (Eiter et al. 1997) based on the DSL proposed;
- **Automation:** We illustrate the increased automation enabled by the specified reasoning principles using the logic programming engine DLV (Leone et al. 2006). Our machinery enables two types of automated reasoning: *(1) Controllability:* which hazards can be controlled by the given deployed safety patterns and which hazards cannot be controlled. *(2) Safety Pattern Recommendation:* which safety patterns can be used and where exactly they should be deployed to control hazards that have not yet been controlled.

We validate our machinery[1] with two examples of safety-critical embedded systems taken from the automotive domain. The first example is an *Adaptive Cruise Control* system installed in a vehicle to adapt its speed in an automated fashion without crashing into objects in front and at the same time trying to maintain a given speed. The second example is a *Battery Management System* (Martin et al. 2020) responsible for ensuring that a vehicle battery is charged without risking it to explode by, *e.g.*, overheating. Our machinery infers a number of possible solutions involving different safety patterns that can be used to control identified hazards.

The remainder of the paper is structured as follows. Section 2 describes two motivating examples. Section 3 briefly describes basic notations of safety patterns, answer-set programming and disjuctive logic programs. Sections 4, 5, and 6 describe our main contributions. Section 7 illustrates the types of results that can be delivered by our machinery. After a discussion of the related work in Section 8, the paper is concluded in Section 9.

[1] All machinery needed to reproduce our results are publicly available: `https://github.com/ygdantas/safpat`

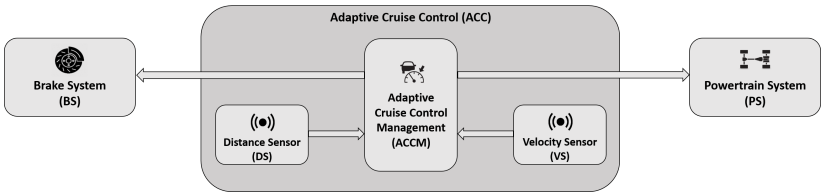


Fig. 1: Adaptive Cruise Control (ACC) Functional Architecture

2 Motivating Examples

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

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

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

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

H0$_{acc}$: The vehicle does not maintain a safe distance to any object in front.

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

- **H1**$_{acc}$**– Erroneous ACC:** ACC computes incorrect acceleration or braking values;
- **H2**$_{acc}$**– Loss of ACC:** ACC is not functioning.

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

- **H1.1**$_{acc}$**- Erroneous DS:** The DS computes an incorrect distance to the car in front;
- **H1.2**$_{acc}$**- Erroneous VS:** The VS computes an incorrect velocity;
- **H1.3**$_{acc}$**- Erroneous ACCM:** The ACCM computes wrong acceleration or braking values.

*Battery Management System (*BMS*).* We consider a simplified BMS responsible for controlling a rechargeable electric car battery (Martin et al. 2020). The BMS is a critical system as harm, *e.g.*, battery explosions, may occur if it does not compute the charging state of the battery correctly.

Figure 2 depicts the main functions composing the BMS. The charging interface (CI) represents the interface at the charging car station. This interface is triggered while recharging the battery (BAT) of the car. BMS receives relevant information (*e.g.*, voltage and temperature values) from BAT so that it can compute the charging state of BAT. Depending on the state of BAT,

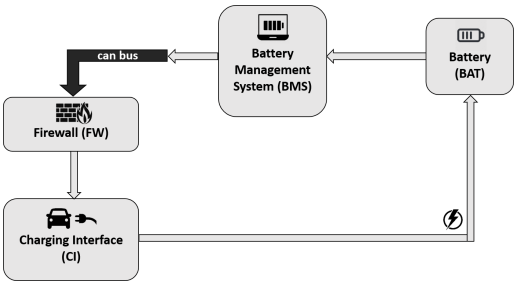


Fig. 2: Battery Management System (BMS) Functional Architecture

BMS sends signals of activation or deactivation of the external changer to CI. These signals are sent though a CAN bus. CI is considered the only function accessible by external users (e.g., drivers). To avoid that an intruder can access the CAN bus through CI, a firewall (FW) is placed between BMS and CI.[2] This decision, however, comes at a safety impact, as mentioned below.

The main hazard considered here is:

$\mathbf{H0}_{bms}$: The BAT is overcharged leading to its explosion.

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

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

The following three hazards may lead to $\mathbf{H1}_{bms}$. We use the word *omission* as hazard whenever a function is not provided when expected.

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

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

3 Preliminaries

This section reviews basic notions about safety patterns, ASP and disjuctive logic programs.

Safety Architectural Patterns. In the architectural level, a number of safety patterns are typically used for embedded system safety (Matos et al. 2014; Preschern et al. 2013). Examples of such patterns are Heterogeneous Duplex Redundancy (HDR), Triple Modular Redundancy (TMR), N-Version Programming (NProg), Safety Monitors (SafMon), and Watchdog (WD).

The goal of these patterns is to control some type of hazards provided some conditions are satisfied. WDs are used to detect when there is loss of function, thus controlling hazards associated with a *loss* of function. SafMons are used to check whether a function is computing correctly, thus controlling hazards associated with *erroneous* functions. HDR and TMR are used to control

[2] We refer the reader to (Martin et al. 2020) for more insights on why adding a FW between BMS and CI makes the system more secure.

hazards by increasing the redundancy of existing hardware, thus reducing the overall fault rate. They can also be used to increase the redundancy of paths in the system in case messages are lost or incorrectly computed. NProgs are used control hazards associated with possibly *erroneous* software functions by increasing the redundancy of such functions.

Answer-Set Programming and Disjunctive Logic Programs. We assume that the reader is familiar with Answer-Set Programming (ASP) and provide only a brief overview here. Let $\mathscr{K}$ be a set of propositional variables. A *default literal* is an atomic formula preceded by *not*. A propositional variable and a default literal are both *literals*. A rule r is an ordered pair $Head(r) \leftarrow Body(r)$, where $Head(r) = \ell$ is a literal and $Body(r) = \{\ell_1, \ldots, \ell_n\}$ is a set of literals. Such a rule is written as $\ell \leftarrow \ell_1, \ldots, \ell_n$. An *Answer-Set Program* (LP) is a set of rules. An interpretation M is an *answer set* of a LP P if $M' = least(P \cup \{not\_A \mid A \notin M\})$ and $M' = M \cup \{not\_A \mid A \notin M\}$, where least is the least model of the *definite logic program* obtained from the program P by replacing all occurrences of *not A* by a new atomic formula *not_A*.

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

$$a \leftarrow not\ b \qquad b \leftarrow not\ a \qquad c \leftarrow a$$

DLV is an engine implementing disjunctive logic programs (Eiter et al. 1997) based on ASP semantics (Gelfond and Lifschitz 1990). In particular, a rule may have disjunction in its head, *e.g.*, $a_1 \vee \cdots \vee a_m \leftarrow \ell_1, \ldots, \ell_n$, where a_i for $0 \leq i \leq m$ are atomic formulas. For example, consider the program P_1 with the two clauses $a \vee b$ and $c \leftarrow a$. It has the same two answer-sets as the program P. If a rule's head is empty, *i.e.*, $m = 0$, then it is a constraint. For example, if we add the clause $\leftarrow b$ to P_1, then the resulting program has only one answer-set $\{a,c\}$.

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

4 Basic Domain-Specific Language: Functional, Hardware and Safety Patterns

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

Example 4.1
The functional architecture depicted in Figure 1 is specified by the following atomic formulas, or facts, using the notation of the DLV prover (Leone et al. 2006):

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

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

Functional, Hardware and Safety Analysis	
Fact	**Denotation**
cp(id)	id is a function in the system.
subcp(id_1,id_2)	id_1 is a sub-function of the function id_2.
ch(id,id_1,id_2)	id is a logical channel connecting an output of the function id_1 to an input of the function id_2. Notice that it denotes a unidirectional connection.
if(id,$\overrightarrow{\mathsf{ch}}$)	id is an information flow following the channels in $\overrightarrow{\mathsf{ch}}$.
hw(id)	Function id is implemented as hardware, *e.g.*, circuit connected to sensors.
sw(id)	Function id is implemented as a software.
hz(id,id_c,tp,sv)	id is a hazard associated with the function id_c is of type tp, where $\mathsf{tp} \in \{\mathsf{err}, \mathsf{loss}, \mathsf{omission}, \mathsf{late}, \mathsf{early}\}$, and severity sv, where $\mathsf{sv} \in \{\mathsf{minor}, \mathsf{major}, \mathsf{fatal}, \mathsf{cat}\}$. err, loss, omission, late, and early denote, respectively, erroneous, loss of function, omission, late and early types of hazards (Stalhane 2015). minor, major, fatal, cat denotes, respectively, minor, major, fatal and catastrophic severity levels.
subHz(id_1,id_2)	id_1 is a hazard causing hazard id_2.

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

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

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

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

For example, the hazard **H1.3**$_{acc}$ (h13) is a sub-hazard of **H1**$_{acc}$ (h1).

Due to space limitations, we illustrate only the safMon pattern. The remaining patterns follow a similar reasoning. We refer the reader to (Prescher et al. 2013; Matos et al. 2014) for detailed description of these patterns.

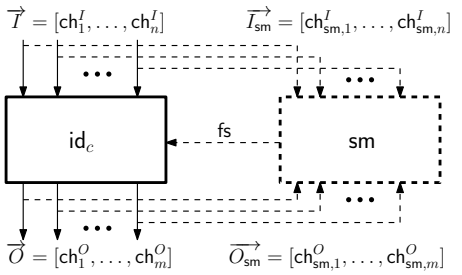


Fig. 3: Safety Monitor Pattern

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

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

Safety Architectural Patterns	
Fact	**Denotation**
HDR (id,id_c,I_c,$\mathsf{id}_{c\prime}$,I_{vt_1},I_{vt_2},vt, vt_{out},id_{out})	id is a duplex redundancy associated with the function id_c. I_c is a channel from id_c that might convey a fault message. $\mathsf{id}_{c\prime}$ is a function possibly id_c. vt is a voter that receives data from id_c and $\mathsf{id}_{c\prime}$ through channels I_{vt_1}, and I_{vt_2}, respectively. The result from vt is sent to id_{out} through channel vt_{out}.
TMR (id,id_c,I_c,$\mathsf{id}_{c\prime}$,$\mathsf{id}_{c\prime\prime}$,$I_{\mathsf{vt}_1}$,$I_{\mathsf{vt}_2}$, I_{vt_3},vt,vt_{out},id_{out})	id is a triple modular redundancy associated with the function id_c. I_c is a channel from id_c that might convey a fault message. $\mathsf{id}_{c\prime}$ and $\mathsf{id}_{c\prime\prime}$ are functions possibly id_c. vt is a voter that receives data from id_c, $\mathsf{id}_{c\prime}$ and $\mathsf{id}_{c\prime\prime}$ through channels I_{vt_1}, I_{vt_2}, and I_{vt_3}, respectively. The result from vt is sent to id_{out} through channel vt_{out}.
2Prog (id,id_c,$\overrightarrow{I_{\mathsf{id}_c}}$,$\overrightarrow{O_{\mathsf{id}_c}}$,$\mathsf{id}_{c\prime}$,$\overrightarrow{I_{\mathsf{vt}_1}}$,$\overrightarrow{I_{\mathsf{vt}_2}}$, $\overrightarrow{\mathsf{VT}}$,$\overrightarrow{\mathsf{VT}_{out}}$,$\overrightarrow{\mathsf{id}_{out}}$)	id is a 2-version programming associated with the function id_c (a.k.a. version 1). $\mathsf{id}_{c\prime}$ (a.k.a. version 2) is an identical function of id_c. The inputs to id_c and the outputs from id_c are sent through channels $\overrightarrow{I_{\mathsf{id}_c}}$ and $\overrightarrow{O_{\mathsf{id}_c}}$, respectively. $\overrightarrow{\mathsf{VT}}$ is a list of voters that receive data from id_c and $\mathsf{id}_{c\prime}$ through channels $\overrightarrow{I_{\mathsf{vt}_1}}$ and $\overrightarrow{I_{\mathsf{vt}_2}}$, respectively. The results from $\overrightarrow{\mathsf{VT}}$ are sent to their respective functions $\overrightarrow{\mathsf{VT}_{out}}$ through channels $\overrightarrow{\mathsf{id}_{out}}$.
safMon(id,id_c,$\overrightarrow{I}$,$\overrightarrow{O}$,fs,$\overrightarrow{I_{\mathsf{sm}}}$,$\overrightarrow{O_{\mathsf{sm}}}$,sm)	id is a safety monitor associated with the function id_c. It uses the list of input and output channels $\overrightarrow{I}$ and $\overrightarrow{O}$, respectively. The data of these channels are sent as input to sm through the list of channels $\overrightarrow{I_{\mathsf{sm}}}$ and $\overrightarrow{O_{\mathsf{sm}}}$. fs is a channel from sm to id_c which sends a fail-safe signal whenever some inconsistency is detected.
watchDog(id,id_c,fs,I_{wd},wd)	id is a watchdog associated with the function id_c. It receives liveness messages from id_c through channel I_{wd}. fs is a channel from wd to id_c which sends a fail-safe signal whenever some inconsistency w.r.t the expected messages is detected.

Table 2: SafPat: Language for Safety Architectural Patterns.

5 Safety Reasoning using DLV

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

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

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

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

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

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

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

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

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

Example 5.1
Consider the hazards and sub-hazards relations in Example 4.1. The hazards `hz(h1,acc,err,cat)` can be controlled if its three sub-hazards, `h11`, `h12` and `h13`, can be controlled.

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

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

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

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

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

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

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

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

HDR. The 2 (HDR) and 3 (TMR) Voter patterns can used for two different safety reasons: (1) to improve safety by hardware redundancy or (2) to improve safety by path redundancy. These are specified by the following rules, where `omission` is a type of error:

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

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

Remark: This paper specifically focuses on architectural principles. That is, we focus on the architecture components and how such components interact with other through channels. Encoding other reasoning principles like, *e.g.*, the way such components are implemented in practice (*i.e.*, their behavior), are left to future work.

6 Automated Pattern Recommendation

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

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

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

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

explore's arguments enable the control of the search space used by the system. For example, if explore(1,safMon), the system attempts to add at most one additional safety monitor to a given architecture. Multiple such facts can be used to recommend different patterns at the same time.

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

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

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

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

The rule above will attempt to place a safety monitor in any applicable location of the architecture. The following clause

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

limits the number of safety monitors that can be recommended to be at most `N`. Here `#count` is a DLV aggregate predicate returning the size of a symbolic set defined by its argument. The rule above returns the number of `CP` to which a new safety monitors are associated with.

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

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

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

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

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

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

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

7 Case Studies

This section illustrates the results of our automated safety reasoning for two case studies, namely Adaptive Cruise Control (ACC) and Battery Management System (BMS). We illustrate our re-

sults by depicting how the architectures of both ACC and BMS would appear on a layout when our machinery is used. The safety patterns suggested by our machinery are depicted as dark gray boxes, and the channels related (inputs or outputs) to such patterns are depicted as dashed arrows.

7.1 Adaptive Cruise Control (ACC)

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

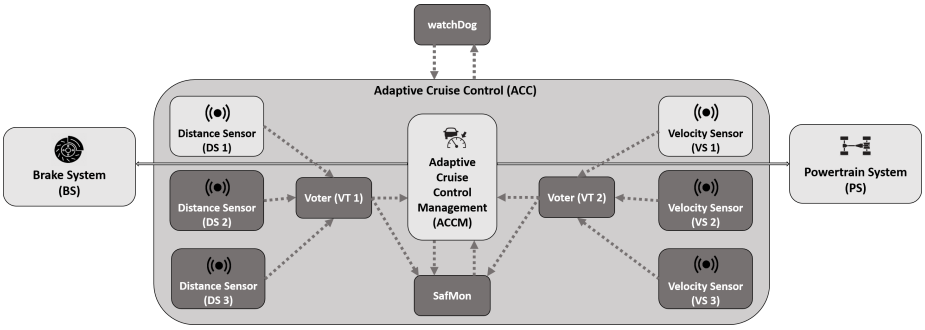


Fig. 4: ACC Functional Architecture with safMon, TMR and WD

We run our recommendation machinery to automatically identify what safety patterns could be used to control **H1**$_{acc}$, **H2**$_{acc}$, **H1.1**$_{acc}$, **H1.2**$_{acc}$, and **H1.3**$_{acc}$. Our machinery yielded *five complete solutions* (*i.e.*, architectures) for controlling these hazards. For the sake of space, we only show one of those solutions. The architecture of the chosen solution is depicted in Figure 4. The subset of our DLV specification for this solution is shown below. It contains the predicates for the recommended safety patterns and controllability.

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

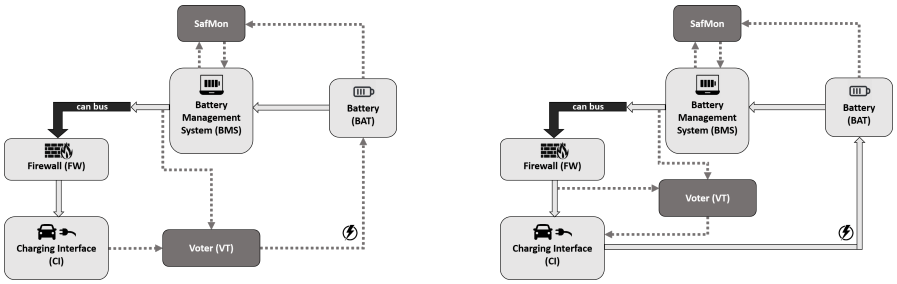
Our machinery recommended to use three safety patterns, *i.e.*, safMon, TMR, and watchDog, to control the identified hazards. The main difference w.r.t. the other solutions (omitted here) is nProg instead of safMon. To control the sub-hazards **H1.1**$_{acc}$ and **H1.2**$_{acc}$, our machinery recommended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

7.2 Battery Management System (BMS)

We identified an erroneous (**H1**$_{bms}$) hazard on CI, as described in Section 2. This erroneous hazard (**H1**$_{bms}$) is broken down into three sub-hazards, namely erroneous BMS (**H1.1**$_{bms}$), erroneous

CAN (**H1.2**$_{bms}$), and omission FW (**H1.3**$_{bms}$). Typically, hazards on CAN buses can be controlled by replacement only. Hence, we assume that **H1.2**$_{bms}$ has already been controlled.

Our recommendation machinery yielded *four complete solutions* (*i.e.*, architectures) to control **H1**$_{bms}$, **H1.1**$_{bms}$, and **H1.3**$_{bms}$. For the sake of space, we only show two of those solutions. The architecture of the chosen solutions are depicted in Figure 4. The DLV specification for those solutions is similar to the one presented in Section 7.1.



(a) Path redundancy for BMS and CI

(b) Path redundancy for BMS and FW

Fig. 5: Battery Management System Functional Architecture with safMon and HDR

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

8 Related Work

Failure Rates Computations. An important analysis for safety is the computation of failure rates of the system and its sub-systems as it is a requirement for safety-critical systems to have (very) low failure rates. The automation of this computation has been subject of some previous work (Helle 2012; ft+ ; Weber et al. 2012). In particular, for a given architecture and given sub-system fault rates, the failure rate of the system is computed. Our work on reasoning with safety patterns complements the work above as we consider the design of the architecture itself, which is part of the input used by the work above.

Safety Case Models. Goal Structure Notation (GSN) (gsn 2011) is a model for specifying safety cases. Safety cases are tree-like structures containing different types of nodes denoting, *e.g.*, Goals, Strategies, Contexts, Assumptions of a safety case. As the exact meaning of each node is specified textually (inside the node), models written in GSN enables little automation. There are, however, work that provide more structure to GSN models and others providing means for some automation (Cârlan et al. 2019). We describe some approaches below.

(Gleirscher and Cârlan 2017) proposes patterns encoding typical safety reasoning principles, such as those using FTA, FMEA, STPA. While these reasoning patterns provide some structure to GSN models, they suffer from the same automation limitations of GSN described above. On the one hand, our work complements this work by specifying reasoning principles based on safety patterns, which was not considered in (Gleirscher and Cârlan 2017). On the other hand, we believe that it is possible to encode some of the reasoning principles described in (Gleirscher and Cârlan 2017) and consider not only safety reasoning with patterns but the other types of reasoning described in (Gleirscher and Cârlan 2017).

(Duan et al. 2017; Dürrwang et al. 2017) propose automated quantitative evaluation methods for GSN models that associated to Goal nodes with values for belief, disbelief and uncertainty. It is not clear from this work how these values are related to the quality of safety argument. We believe that the encoding of our reasoning principles can profit from this work to make the relation between the quality of the safety argument and the belief values more explicit.

Safety Reasoning using Logic Programming. Logic programming has been used in the past for safety reasoning. For example, (Gómez et al. 2014) provides decision support for air traffic control systems by specifying landing criteria in complex landing situations by using Defeasible Logic Programming (DeLP). (S.A. et al. 2014) outlines a method for safety assessment of medical devices also based DeLP. An interesting work is presented in (Ardila et al. 2019) on the formalization of automotive standard requirements (iso 2018) to enable automatic reasoning about compliance with the standard. We take a similar approach to these work as we also use logic programming and engine to support safety engineers in the designing system architecture. However, we do not consider here reasoning with uncertain and incomplete knowledge as in the work above using DeLP. As described above, we are considering extending the type of safety reasoning encoded to also include uncertainty (Duan et al. 2017; Dürrwang et al. 2017). DeLP is a method we could consider for modeling such arguments.

9 Conclusion

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

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

The increased automation provided by our methods seems to support incremental methods for safety (and in the future security). It is possible to identify, *e.g.*, which hazards are no longer controlled whenever there is an incremental change to the system. We are currently investigating how to improve the proposed automated reasoning for this purpose.

Finally, we plan to integrate our machinery into the Model-Based Engineering Tool AutoFOCUS3 (af3). The goal is to enable safety engineers to use our automated reasoning with models

written in AutoFOCUS3. This will also enable the use of automated methods for building safety cases modeled in GSN (Cârlan et al. 2019).

References

AF3 – AutoFOCUS 3. More information at `https://af3.fortiss.org/`

Fault Tree Analysis – FT+. More information at `https://shorturl.at/pDLQ5`

Defence UK Ministry: Safety Management Requirements for Defence Systems UK Ministry of Defence.

1996. Standard ARP 4761: Guidelines and Methods for Conducting the Safety Assessment.

2011. *GSN Community Standard Version 1*. Available at `https://shorturl.at/AMRV4`

2018. Arp 4754a: Guidelines for Development of Civil Aircraft and Systems. Available at `https://www.sae.org/standards/content/arp4754a/`

2018. ISO 26262, Road vehicles Functional safety. Available at `https://shorturl.at/dhFW5`

ARDILA, J. P. C., GALLINA, B., AND GOVERNATORI, G. 2019. Lessons Learned while Formalizing ISO 26262 for Compliance Checking. In *TeReCom*.

BARAL, C. 2010. Knowledge Representation, Reasoning and Declarative Problem Solving. In *CUP*.

CÂRLAN, C., NIGAM, V., TSALIDIS, A., AND VOSS, S. 2019. ExplicitCase: Tool-Support for Creating and Maintaining Assurance Arguments Integrated with System Models. In *WoSoCer*.

CRAWLEY, F. AND TYLER, B., Eds. 2015. *HAZOP: Guide to Best Practice*.

DUAN, L., RAYADURGAM, S., HEIMDAHL, M. P. E., AYOUB, A., SOKOLSKY, O., AND LEE, I. 2017. Reasoning about Confidence and Uncertainty in Assurance Cases: A Survey. In *SERC*.

DÜRRWANG, J., BECKERS, K., AND KRIESTEN, R. 2017. A Lightweight Threat Analysis Approach Intertwining Safety and Security for the Automotive Domain. In *SAFECOMP*.

EITER, T., GOTTLOB, G., AND MANNILA, H. 1997. Disjunctive Datalog. *ACM Trans. Database Syst.*

GELFOND, M. AND LIFSCHITZ, V. 1990. Logic Programs with Classical Negation. In *ICLP*.

GLEIRSCHER, M. AND CÂRLAN, C. 2017. Arguing from Hazard Analysis in Safety Cases: A Modular Argument Pattern. In *HASE*.

GÓMEZ, S. A., GORON, A., AND GROZA, A. 2014. Assuring Safety in an Air Traffic Control System with Defeasible Logic Programming. In *ESWA*.

HELLE, P. 2012. Automatic SysML-based Safety Analysis. In *ACES-MB*.

KONDEVA, A., CARLAN, C., RUESS, H., AND NIGAM, V. 2019. On Computer-Aided Techniques for Supporting Safety and Security Co-Engineering. In *WoSoCer*.

LEONE, N., PFEIFER, G., FABER, W., EITER, T., GOTTLOB, G., PERRI, S., AND SCARCELLO, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Logic 7*, 499–562.

LEVESON, N. AND THOMAS, J. 2018. *STPA Handbook*.

MARTIN, H., MA, Z., SCHMITTNER, C., WINKLER, B., KRAMMER, M., SCHNEIDER, D., AND T. AMORIM, G. MACHER, C. K. 2020. Combined Automotive Safety and Security Pattern Engineering Approach. In *RESS*.

MATOS, H. L. V. D., DA CUNHA, A. M., AND DIAS, L. A. V. 2014. Using Design Patterns for Safety Assessment of Integrated Modular Avionics. In *DASC*.

NIGAM, V., PRETSCHNER, A., AND RUESS, H. 2018. Model-Based Safety and Security Engineering. White Paper, available at `https://arxiv.org/abs/1810.04866`.

PRESCHERN, C., KAJTAZOVIC, N., AND KREINER, C. 2013. Security Analysis of Safety Patterns. *PLoP*.

S.A., GOMEZ, A., GROZA, AND C.I., CHESNEVAR 2014. An Argumentative Approach to Assessing Safety in Medical Device Software Using Defeasible Logic Programming. In *MEDITECH*.

STALHANE, T. 2015. FMEA, HAZID, and Ontologies. In *Ont. Mod. in Phy. Ass. Int. Man.*.

WEBER, P., MEDINA-OLIVA, G., SIMON, C., AND IUNG, B. 2012. Overview on Bayesian Networks Applications for Dependability, Risk Analysis and Maintenance Areas. In *Eng. Appl. Artif. Intell.*