Security Modelling and Formal Verification of Survivability Properties: Application to Cyber-Physical Systems

S. Bernardi[a], U. Gentile[b], S. Marrone[c,*], J. Merseguer[a], R. Nardone[d]

[a] *Universidad de Zaragoza, Dpto. de Informática e Ingeniería de Sistemas, Zaragoza, Spain*
[b] *Digital Food Safety Department, Nestlè Research, Lausanne, Switzerland*
[c] *Università della Campania "Luigi Vanvitelli", Dip. di Matematica e Fisica, Caserta, Italy*
[d] *Università Mediterranea di Reggio Calabria, DIIES, Reggio Calabria, Italy*

Abstract

The modelling and verification of systems security is an open research topic whose complexity and importance needs, in our view, the use of formal and non-formal methods. This paper addresses the modelling of security using misuse cases and the automatic verification of survivability properties using model checking. The survivability of a system characterises its capacity to fulfil its mission (promptly) in the presence of attacks, failures, or accidents, as defined by Ellison. The original contributions of this paper are a methodology and its tool support, through a framework called `surreal`. The methodology starts from a misuse case specification enriched with UML profile annotations and obtains, as a by-product, a survivability assessment model (SAM). Using predefined queries the survivability properties are proved in the SAM. A total of fourteen properties have been formulated and also implemented in `surreal`, which encompasses tools to model the security specification, to create the SAM and to prove the properties. Finally, the paper validates the methodology and the framework using a cyber-physical system (CPS) case study, in the automotive field.

Keywords: Security specification, formal verification, survivability properties, UML, Cyber-physical systems (CPS)

1. Introduction

Some years ago, Cheng et al. [15] identified that becoming computing systems ever more pervasive, mobile and targets of security attacks, new challenges to security requirements engineering would be posed. Therefore, they advised that works on notations and methodologies for modelling and verifying high-level security policies would become strategic. More recently, Bures et al. [13] also identified as open yet the research topic on the need for verifying requirement specifications of cyber-physical systems (CPS) and declared its inherent complexity.

CPS are networked embedded systems used to monitor and control the physical world [72], for example, electrical power grids, oil and natural gas distribution, transportation systems or health-care devices. Undoubtedly, CPS security is of primary importance in the current networked world and understanding their vulnerabilities, attacks and protection mechanisms is a must for developing the underlying control software [34].

Among the list of challenges, identified by Cheng et al. [15] and Bures et al. [13], on requirements engineering for securing CPS, this work helps in the modelling of security requirements, early in the software life-cycle, and in the formal and automatic verification of system properties. Regarding the kind of properties, we mostly focus on system survivability ones. The survivability of a system can be defined as its capacity

*Contact author
Email address: `stefano.marrone@unicampania.it` (S. Marrone)

"to fulfil its mission on time, in the presence of attacks, failures, or accidents" [20], then preventing perpetual service degradations, outages or integrity leaks, for example.

Survivability, as defined in the original papers by Ellison et al. [20] and Knight et al. [38] embraces security and safety requirements, since it encompasses under the term *threats*, both attacks (usually named *threats*, in the security community) and accidental faults (often named *hazards*, in the safety community), and corresponding protection mechanisms (i.e., *survivability strategies*). This work considers misuse cases, introduced by Alexander [3] as follows: "Misuse cases – a form of use cases – help document negatives scenarios. Use and misuse cases, employed together, are valuable in threat and hazard analysis, system design, eliciting requirements, and generating test cases." Consequently, misuse cases are used for eliciting and specifying both security and safety requirements. In the context of CPS, we consider critical both types of requirements, and survivability (also referred to as *resilience*, as stated more recently in Goertzel et al. [31]) provides a framework for their modelling and analysis. For example, consider in the critical infrastructure domain, the well-known Stuxnet attack [40] – the first advanced persistent (APT) threat – to a CPS, that is the SCADA and PLC system of the nuclear plants of Iran in 2010, that provoked substantial damage to nuclear plants. The consequences of such damage could have been even more severe, also affecting people and the environment. This paper considers a smart car case study, that is a safety-critical CPS (a system failure may have catastrophic consequences on the user(s) and the environment), where safety requirements (expressed by ASIL – safety-integrity levels) can be affected by attacks.

The original contributions of this paper are a methodology and its tool support, through a framework called `surreal`. The methodology comprises different phases, significantly modelling and verification, and artefacts. For modelling, misuse cases are enriched with a UML[1] profile, then defining a security specification, where sequences of attacks and protections are inferred. By protections, we mean countermeasures, introduced to allow the system to recover from an attack. The UML profile [67] extension mechanism enables to tailor the language to different domains, in our case, the survivability domain, by introducing concepts such as survivability strategies or service modes. For verification, state-of-the-art model-checking techniques are used to prove predefined survivability properties on the security specification. The `surreal` framework offers support for all the methodology phases proposed in this paper.

The methodology and the framework, described in this work, contribute to the requirements engineering process by supporting the analysts in better eliciting and assessing security requirements, especially those related to system survivability. Concretely, the work contributes to:

- model threats to essential services, and the countermeasures needed to recover the system from degraded states;
- and to verify survivability properties by checking the modelled specification.

More specifically, this paper extends our previous work [8, 27] in many different aspects. First, it introduces model-checking for producing an assessment model automatically. Second, it proposes fourteen predetermined queries, ready to be used by the analyst, for verifying system survivability properties. Third, model checking is also used for carrying out such verification automatically. Fourth, we extend the UML profiles presented in [8, 27] to accommodate these new features, then improving the profiles modelling capabilities. Fifth, the current work provides tools that automate, for the analyst, the steps of the methodology. Last, this paper validates the approach with a case study in the CPS domain.

The structure of the paper is as follows. Section 2 recalls the background supporting this paper as well as some related works. Section 3 presents, at a glance, the methodology and tools that are part of the `surreal` framework, for the reader to catch the overall picture. Sections 4 and 5 describe the internals of each phase of the methodology. Section 6 elaborates a case study in the automotive field, which demonstrates the applicability of the methodology in the CPS context and the usefulness of the tool framework. Finally, Section 7 summarises the assumptions and conducts a threat to validity analysis, and Section 8 concludes the paper.

[1] Unified Modeling Language [67]

2. Background and related works

This section is devoted to review the background on survivability modelling and model-checking techniques (Sub-section 2.1) and the related works (Sub-section 2.2).

2.1. Background on survivability modelling and model-checking techniques

On survivability modelling. In previous works [8, 27], we proposed and implemented a UML profile[2] for specifying system survivability requirements. In particular, four main concepts are captured by our survivability profile [20, 38]:

- Essential services —representing system services that must survive despite threats materialisation. They are characterised by non-functional metrics (e.g., performance, integrity or availability) that define their health.
- Service modes — defining different Quality of Service (QoS) levels of the system according to combinations of essential services measured by their health, i.e., by a QoS index. For example, the system is in "fully operational" service mode when the availability of all of its essential services is greater than 90%.
- Threats — representing either activity carried out by attackers or materialisations of natural causes (e.g., blackouts) resulting in system failures. They may compromise essential services by degrading the system quality.
- Survivability strategies — resistance, recognition and recovery actions aimed to prevent/react against consequences of threats. They are countermeasures to threats that try to maintain or restore the health of essential services.

Considering a military command and control system, used as a running example in this paper[3], an *essential service* is, for instance, the provision – via GPS trackers – of up-to-date position awareness of military forces on a digital map. On the one hand, *threats* affecting this service can be either attacks or accidental faults: for example, respectively, a man-in-the-middle attack – that counterfeits the position of the enemy forces in the digital map – or unintentional destruction of the deployment platform where the essential service is running on. On the other hand, for each threat, different *survivability strategies* can be applied to mitigate it. In particular, the man-in-the-middle attack could be reduced by combining different types of strategies, such as implementing cryptographic protocols in GPS communication (*resistance*), anomaly detection techniques (*recognition*) and restoration of original geodata (*recovery*) after the attack has been detected. The accidental destruction of the deployment platform could be mitigated by the implementation of fault-tolerance mechanisms such as hardware and software redundancy and reconfiguration (*recovery*).

As shown in Figure 1, the survivability profile has two main packages, `Misuse case` and `SAM` (Survivability assessment model) extensions, and a package for types definitions. The `Misuse case` package extends UML use cases, and it is used to enrich misuse case specifications.

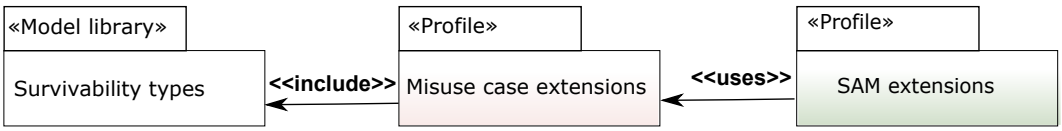

Figure 1: Survivability profile overview

In particular, the `Misuse case` package extends existing concepts proposed in the original misuse case notation [3] by including the survivability concepts from [20, 38], previously mentioned – i.e., essential

[2] A UML profile [42, 59] is a set of extensions (i.e., stereotypes and tags) that can be applied to UML model elements.
[3] The running example is introduced in Section 3.

services, service mode definition and survivability strategies – and by enabling the specification of QoS indices – e.g., availability metric. The `SAM` package is applied to UML state machines to specify system service modes, sequences of threats and survivability strategies.

Table 1 summarises the stereotypes of the profile used in this paper, where the last column highlights changes concerning the previous proposals [8, 27]. In particular, *tag type* means that the tag has a different meaning: in [27] misuse and recovery stereotypes have a *targetServiceMode* tag to specify the system service mode reached as a consequence of the stereotyped (misuse or recovery) use case, whereas here the *affects* tag is used to determine the QoS indices of the essential services affected by the stereotyped (misuse or recovery) use case. Similarly, in [8], state machine transitions can be annotated with a tag to specify the event that triggers the change of service mode (a misuse or a survivability strategy), whereas here the *path* tag of a *scenario* stereotyped transition is used to specify a sequence of misuse cases and survivability strategies that causes the change of a service mode. *Tag refinement* means that the *indices* tag has the same meaning as in our previous proposal [27], i.e., it is used to specify the QoS indices associated to an essential service, but it enables a finer-grained specification (i.e., the type of value domain, the value domain and the initial value). Appendix A presents the complete profile. The profile is now a component of the `surreal` framework that supports this paper.

It is worth noticing that the Survivability profile provides general concepts that can be applied to different domains, including cyber-physical systems (CPS). The application of such concepts to CPS will be illustrated, in this paper, with a running example of military command and control system and a case study of a smart car, in the automation domain.

Misuse case extensions package				
Stereotype	***Description***	***Tags (type)***	***Extended UML metaclass***	***Changes w.r.t. [8, 27]***
service	An essential service	indices (index)	Use case	tag refinement
misuse	A threat scenario	affects (affectConsequence)	Use case	tag type
threatens	A threat to a service		Dependency	
mitigates	A threat mitigation		Dependency	
service mode definition	Definition of the service modes	formula (String)	Constraint	new
recovery	A recovery strategy	affects (affectConsequence)	Use case	tag type
SAM extensions package				
Stereotype	***Description***	***Tags (type)***	***Extended UML metaclass***	***Changes w.r.t. [8, 27]***
mode	A service mode	severity	State	
scenario	A sequence of misuses and survivability strategies	path (MSactivation)	Transition	tag type

Table 1: UML profile extensions used in the approach

On model-checking techniques. Model-checking is a formal method that, given a finite-state model of a system and a formal property, systematically checks if the property is verified for each possible sequence of states in that model. If a violation of a property is detected, the model checker produces a counterexample that is a sequence of analysed states whose crossing lead to the violation. Model-checking can be automated and can be, in general, applied to both software and hardware systems for the verification of properties related to communication protocols, concurrent systems or even for safety-critical systems. In [4], model checking is used to analyse the completeness of requirement specification, the work in [29] proposes automated planning to compute sequences of actions able to reach a specified goal, while the work in [60] deals with the automatic generation of attack graphs in network security. Model-checking is suggested by the international standards (e.g., ISO 61508, ISO 26262) for the verification of safety-critical system specifications. In the safety-critical domains, Wang et al. [70] use a model checker to verify safety properties of the integrated modular avionics

(IMA) – a computing network involved in aircrafts software development – and Beneceretti et al. [7] proposes a framework based on model checking for the automatic system-level test case generation.

Model-checking can be even combined with learning-based techniques to obtain and verify properties from a black-box view of a system. In [21] model-learning combined with model checking, has been used to detect some flaws within the TCP protocols, by verifying properties on learned models of different clients and servers. Despite all the benefits introduced, the use of model checking comes with some drawbacks. First, the formal languages used to feed model checkers are mathematics-based and are often very complicated to fully master. Second, there is no guarantee that the counterexamples generated by a model checker are of a minimal length. To this aim, extra and more computation demanding techniques can be adopted, e.g., Bounded Model Checking [11].

2.2. Related works

In the following, we review the related works on the modelling of security and safety requirements and their verification with model checking techniques.

Security and safety requirements modelling. Hundreds of works can be found in the literature regarding the elicitation and modelling of security and safety requirements. The surveys [68, 54, 66] offer a good insight in this field. However, unlike our work, the greatest part of these works deal with safety-related approaches oriented to hazard identification. Among them, the following ones are of interest, although they do not use UML nor apply model checking techniques. In [65], the author proposes a method based on fault tree analysis (FTA) to derive requirements with the support of a state-based model. The Event-B formalism for control-systems is used in [45] and [49]; interestingly the former uses it to automate Failure Modes and Effects Analysis (FMEA) partially. In the area of safety management and safety-driven design, Leveson presents the STAMP/SPTA approach [43] to meet assurance goals in software projects among other fields. This approach is followed and applied in other papers [61, 22]. In the field of cyber-physical applications, the work in [48] presents a technique, to model interactions between components, that allows reasoning about timing behaviour.

The works in [17, 39, 71] are also in the field of safety-related approaches but closer to ours. In particular, Dörr et al. [17] propose a requirement elicitation process based on use case modelling. Koh et al. [39] use model checking, as our work, and FTA and combine such techniques to verify security requirements automatically. Yoo et al. [71] introduce a new formal method — NuSCR — to elicit safety-critical requirements and apply it to nuclear plants.

Security and safety requirements modelling using UML. The literature on modelling security and safety requirements, using UML is also large. In the following we only recall: a) some works highly cited in the literature, some of them have inspired our approach, and b) some UML profiles that have been the baseline for the profile presented in this paper. SecureUML [44] is a seminal work in modelling security based on UML and the model-driven paradigm. The approach presents a methodology for modelling access control that also offers support for specifying complex authorisation constraints. Then, SecureUML focusses on specifying role-based access control policies and requirements, while our approach is for the modelling of attacks and protections. The final goal of SecureUML is to automatically generate security infrastructures for access control while our approach aims to assess survivability properties of systems. The CORAS method[4] is oriented to model-driven risk analysis of changing systems [46], the CORAS language is used to support the analysis of security threats and risk scenarios in security risk analyses. UMLsec [36] allows to specify security information during the development of security-critical systems and provides tool-support for formal security verification according to the SVDT approach [32]. SVDT and its successor [28] allow for evaluating (already) verified alternatives against different requirements, including time-to-market and budget constraints. All these approaches are applied to software system design and IT security.

Regarding UML profiles, MARTE [51] (Modeling and Analysis of Real-Time and Embedded Systems) is an OMG standard mostly focussed on schedulability and performance. DAM [9] is a MARTE extension for

[4] http://coras.sourceforge.net/index.html

the modelling and analysis of dependable systems, while SecAM [55] extends DAM for security modelling of critical infrastructures, early in the system development life-cycle. CIP_VAM [69, 18] is a UML profile for vulnerability analysis and modelling in the field of critical infrastructure protection[5]. It is used in model-driven chains involving Bayesian networks and quantitative modelling, and it focuses on physical aspects modelling, integration with SecAM was proposed in [47]. Other approaches use SysML [23] instead of ad-hoc UML profiles, as in [57, 56], and others create specific profiles for SysML [12]. An application of the SysML language to a critical system for assessing repair/survivability strategies can be found in [10]. Finally, in [30] a UML profile for modelling functional safety requirements is proposed, the requirements are expressed in OCL and verified directly on the UML model.

Security and safety requirements verification using model checking. As stated at the beginning of the sub-section, different works exist having similar premises: the work [4] presents a tool, based on model checking, to complete the operational requirement specification according to the stakeholders' goals. However, this approach strongly relies on the state-based specification and forces the requirement engineer to define positive and negative scenarios each time the model checker verifies a property violation. Thus, at each iteration, there is the need to define such scenarios against the properties, using the considered temporal logic language.

In this work, we propose a framework that allows the engineer to model safety and security requirements in the same model, using an extended version of the use case diagram. The rationale behind our proposal is to relieve the engineer from the modelling of a precise state-based specification and the definition of the properties to be checked in the temporal logic language. Then, the framework leverages model transformations for the state-based representation of the specification to verify the properties, which are selected by the engineer from a list of properties expressed as English sentences. The results of the verification allow the engineer to make informed decisions about the completeness of the requirement specification.

Based on the STPA methodology, previously commented, the work [33] identifies and formally analyses safety and security requirements, but different from our work, it is not focussed on verifying survivability properties. Verification of safety requirements in large software systems using probabilistic model-checking is proposed in [14]. Unlike our work, the approach in [14] assumes the system already operational, and it is aimed at verifying, at runtime, the compliance with safety requirements. Another work, with some common points with our approach, is the seminal work in [25] that uses model-checking to test software implementations from requirements specifications. The main difference with this approach is that the methodology here proposed aims at verifying the specification instead of the resulting software artefact. Finally, the Formal Tropos [24] and Secure Tropos [50] approaches deserve to be mentioned. Formal Tropos is a language that enables the automatic verification of requirements using model-checking, although it is not explicitly devoted to security requirements. Secure Tropos is for the analysis of security requirements alongside functional ones. It drives system designers from the acquisition of requirements up to their verification. There exist two versions of Secure Tropos, one extends the i*-language and the other extending Tropos. Secure Tropos also offers a CASE tool [53].

In the light of the works above reviewed and considering the improvements, summarised in Section 1, that this paper offers concerning our previous works [8, 27], we can stress some conceptual differences with related works in the literature, as follows. First difference, we overcome the single-stepped attack and single-stepped recovery hypotheses assumed in [27], see Sub-section 4.2. Second, the proposed fourteen queries define a starting rich-full framework for guiding the analyst to select the requirements of interest to be verified in the system. Third, the use of model checking, for the automatic verification of the selected requirements, produces counterexamples that significantly helps the analyst, for example: 1) to find sequences of attacks and repairs, or 2) to find degradation paths, as well as recovery paths and strategies. All these improvements conform to a framework that empowers the analyst to automatically obtain an assessment model, that helps in many tasks concerning the automatic verification of system security properties.

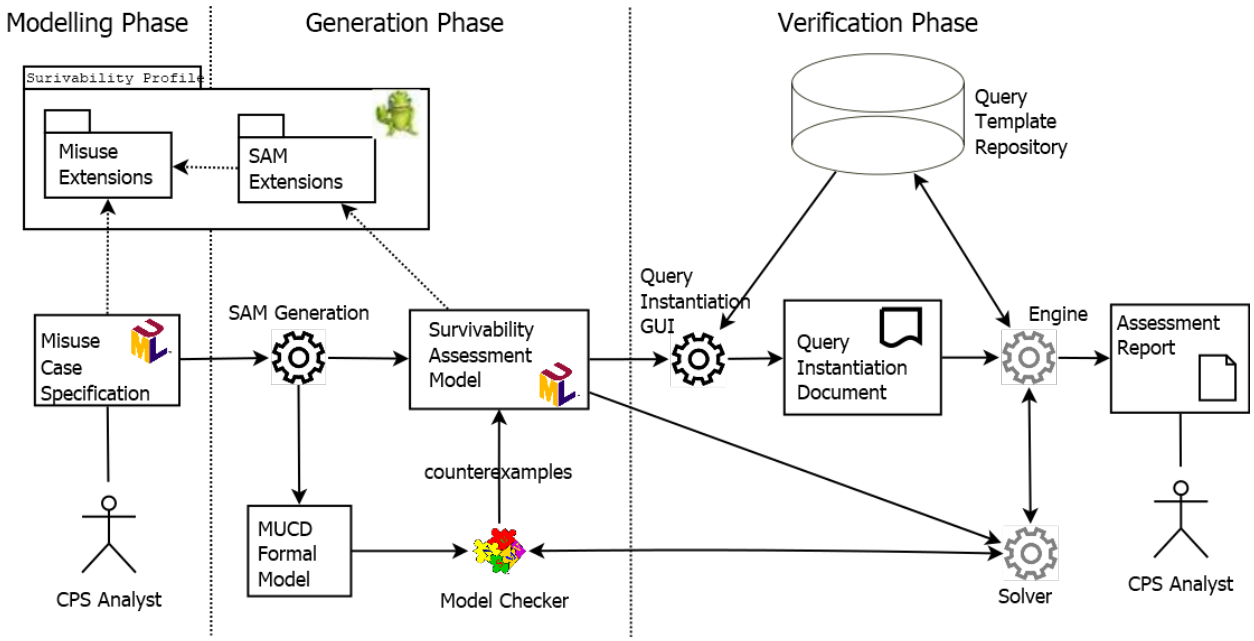

Figure 2: Methodology and tool framework overview

3. Methodology overview

Figure 2 presents the big picture of our methodology and related framework. The methodology is composed of three phases: modelling, generation and verification. Regarding tools, those depicted as black gears are used by the CPS analyst, while the grey ones are invoked transparently by the other tools.

During the modelling phase, the CPS analyst defines (functional and non-functional) system requirements building a UML misuse case diagram (MUCD) [3]. This specification is enriched by identifying essential services, threats, survivability strategies, and system service modes, which are annotated using the survivability profile recalled in the previous section.

During the generation phase, the aim is to create a `survivability assessment model` (SAM). The SAM is a UML state machine that represents the system service modes and the change of service modes caused by the occurrence of threats and the application of survivability strategies. The `SAM generation` tool accomplishes the task automatically, through several steps. In the first step, starting from the MUCD, the states of the SAM are created, they represent system service modes. Next, the tool transforms the MUCD into a Kripke model (`MUCD formal model`), that can be analysed by a model checker, e.g., NuSMV [16]. The results of the analysis, in particular the counterexamples provided by the model-checker, are then post-processed to add the transitions of the SAM and label them with sequences of events (threats and survivability strategies). Consequently, the SAM represents the system evolution throughout the different service modes using the possible sequences of threats occurrences and survivability strategies execution.

In our methodology, the verification phase deals with the verification of system survivability properties. The properties are specified as abstract queries and stored in a `query template repository` (QTR). The CPS analyst, using the `query instantiation GUI`, selects queries and instantiates them with actual elements of the MUCD (e.g., misuse cases) or of the SAM (i.e., service modes). Then, an `engine` is called, which downloads, for each instantiated query, a `solver` capable of executing it. Finally, the query is proved against the SAM by the solver and results are presented to the analyst in the form of an `assessment`

[5]CIP_VAM was developed within the European project METRIP `http://metrip.unicampus.it/`

`report`. Currently, the surreal framework allows assessing fourteen different survivability properties, that is the properties listed in Table 3. The rationale behind the choice of these properties is to provide a general support for the assessment of systems survivability, and this paper applies them in the CPS context. The support encompasses the analysis of the recoverability of service modes (*Security level* properties), and the analysis of the effect of threat occurrences and survivability strategies on the service modes (*Threat* and *Mitigation* properties). Moreover, the framework has been designed for being easily extensible regarding new survivability properties.

The assessment report allows the analyst to make informed decisions about the completeness of the requirement specification. For example, one property of interest to verify is the *strong reversibility*, that is the possibility to recover the system to a given service mode (property P1 in Table 3). This property does not hold when the specification omits possible survivability strategies mitigating one or more threats represented as misuse cases; in such a case, the analyst can decide to refine the MUCD by adding such strategies and repeat the generation and verification phases with the refined specification. Therefore, the modelling and verification activities, supported by the interactive methodology, are carried out by the CPS analyst in a cyclic manner, until a requirement specification that satisfies the properties of interest is found.

Motivation. In requirement engineering it is impossible to find a "silver bullet" and, in the case of cyber-physical systems, this task is worsened by the confluence of software, hardware, mutable operating environments and the human factor, since emerging behaviours are not rare but hard to predict. Hence, formal methods are just one of the techniques that can be used in such systems. They proved their effectiveness with many success stories, from Paris metro systems [6] to the Intel's practices for the design of CPU architectures using model checking [37]. Notwithstanding such techniques, new vulnerabilities are found even in those processors (e.g., the Spectre and Meltdown vulnerabilities). We strongly believe that the approach proposed in this paper, like other similar techniques, can not be as a one-size-fits-all tool for system hardening.

First, the proposed methodology is for eliciting security requirements, hence, it should be first used in the early stages of the system development to discover failure scenarios. The fulfilment of such requirements should be then assessed later in the lifecycle.

Second, the proposed approach does not exclude but creates synergy with other techniques as testing. As it has been demonstrated in [7] model checking combined with functional testing may be successful in industrial settings.

Finally, the proposed methodology must be embedded in the development process. As it is impossible to have detailed knowledge in the early stages of development, we think that it should be applied more than once during the system lifecycle.

The choice of focusing on a functional level description is because the most widespread methodologies dealing with safety, security and in-the-large, dependability assessment of a product/system during the whole duration of its lifecycle, start with some kind of Functional Hazard Assessment (FHA). This family of methodologies is in charge of eliciting, determining the proper level of safety (or other dependability attributes) for each component with the consequent definition of the appropriate design and validation processes. Since such processes start in the very early phases of the system lifecycle, when the architecture is not often defined yet, then functions that the system has to provide are the only known system assets. Since our approach supports such phases, a functional view of the system is a right starting point. To help this point of view, the most adopted international standards recommend FHA in the early phases across different domains (e.g., IEC 61508 [1], ISO 26262 [35], EN 50128 [2]).

Running example. Modern military command and control systems are actually systems of systems that incorporate fully-integrated modular cyber-physical systems such as personal combat displays, unmanned aerial systems and tactical mobility night vision devices to enhance the situational awareness and improve decision-making [19]. To support the methodology description a military command and control system [8] is used as a running example.. The system provides two basic essential services: messaging and map positioning. These services must survive despite the presence of faults or attacks, thus allowing the officers in charge to send timely their orders to subordinates and to achieve the situational awareness in the battlefield. In particular, we will address the following questions that indeed represent system survivability requirements:

- Is it always possible to recover to the service mode that provides the highest quality (the *best service mode*)?
- Let us suppose the system is offering the highest quality service and man-in-the-middle attacks occur that manipulate the information about the operations plan exchanged between the officers, what is the service quality provided by the system after the attacks?
- Let us consider a set of possible survivability strategies that can be used to improve the service quality in a degraded service mode. Which is the smallest subset that allows reaching the *best service mode*?

4. Modelling and generation phases

4.1. Modelling phase

In the modelling phase, the CPS analyst creates a MUCD enriched with a survivability specification. A MUCD is the result of a requirements elicitation process where four tasks can be emphasised: a) the elicitation of the essential services, which should *survive* despite the presence of threats; b) the vulnerability analysis (or threat modelling), where threats affecting essential services are identified; c) the definition of survivability strategies, that aim at eliminating or mitigating threats; and d) the definition of system service modes, which guarantee different levels of QoS, from the best one offered by the system, for all essential services, to the most degraded, but still acceptable. Specifically, system service modes are ranked according to the relevance of the QoS indices and their threshold values (i.e., QoS levels) associated to essential services: all system service modes but the one with the best QoS level are considered degraded service modes (or degraded states).

The specific methodology used to carry out the elicitation process has been already presented in [8] and it is here omitted. the emphasis is, indeed, on the artefacts produced by the process. The vulnerability analysis takes into account two hypotheses: (i) the threats (or misuses) are independent, they may occur concurrently, and (ii) they are carried out in a single step. The same holds for the recovery strategies which are considered as single-step actions to recover from a degraded state.

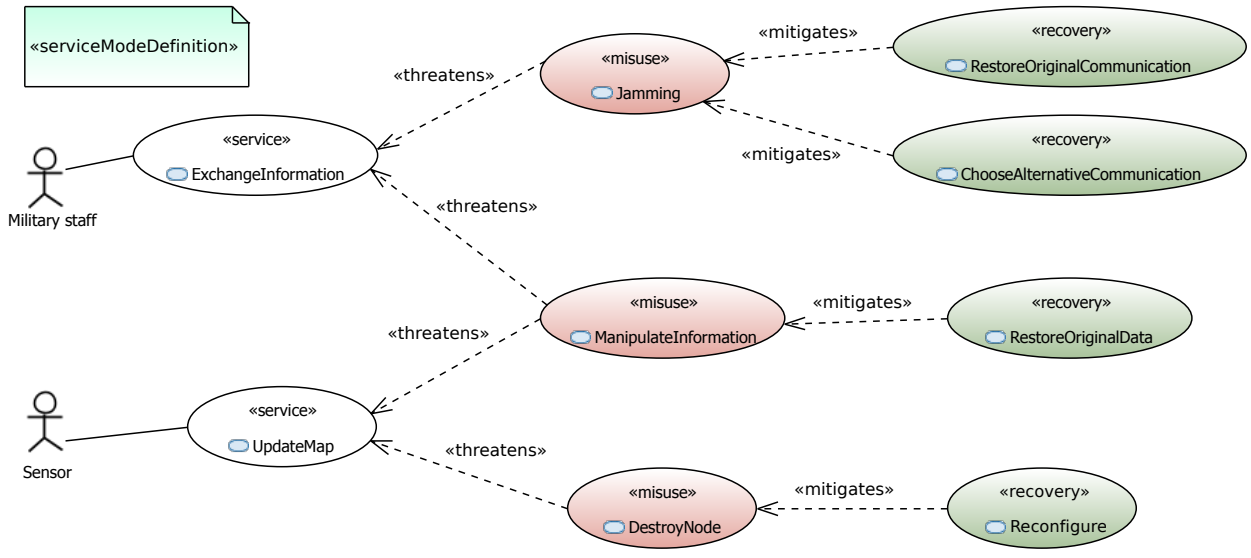

Figure 3: MUCD of the running example annotated with the survivability profile

Figure 3 shows an excerpt of the MUCD of the running example, where just two essential services are considered: i.e., *ExchangeInformation* – that is initiated by the military staff and includes different scenarios, such as the sending of reports, the request of supplies and the transmission of orders – and *UpdateMap* –

that is triggered by a sensor, like a GPS tracker, and it provides up to date position awareness of military forces on a digital map. A more comprehensive model can be found in [8], herein we refine the misuse case diagram applying the survivability profile, where for the sake of clarity, the tagged values associated to the stereotyped model elements are summarised in Table 2. In particular, essential services are characterised by two QoS indices, i.e., the availability and the integrity level, both express a percentage, thus they are defined over the interval `0..100`, and their initial values are set to the highest value. The initial values represent the optimistic situation where the system is not affected by threats.

There are three misuse cases in the diagram: *Jamming* represents an attack aimed at interrupting (or slowing down) the communication, it is usually carried out by sending interference signals; *ManipulateInformation* represents an attack that is aimed at manipulating the geodata used to update the digital map or the information exchanged by the military staff; and *DestroyNode* that models the destruction of a node, which can be either accidental or intentional. The misuse cases compromise the QoS of the essential services, in particular, each misuse case may affect one or more QoS indices and the degradation of a QoS index value is specified using the *affects* tagged-value (see in Table 2). For example, the *Jamming* misuse case affects the availability of the *ExchangeInformation* service, it may occur multiple times, and its occurrence decreases the initial availability value of 10%, whereas the *ManipulateInformation* misuse case affects the integrity level of the two essential services by halving its value. The *DestroyNode* misuse case affects both the availability and the integrity level of the *UpdateMap* essential service by setting their value to zero and 10, respectively.

Stereotype: service	***Tagged-values: indices (name,kind,values,initial)***
ExchangeInformation	(avail, integerInterval, 0..100, 100) (integLevel, integerInterval, 0..100, 100)
UpdateMap	(avail, integerInterval, 0..100, 100) (integLevel, integerInterval, 0..100, 100)
Stereotype: misuse	***Tagged-values: affects (index,set,inc,dec)***
Jamming	(avail, –, –, 10)
ManipulateInformation	(integLevel, 50, –, –)
DestroyNode	(avail, 0, –, –) (integLevel, 10, –, –)
Stereotype: recovery	***Tagged-values: affects (index,set,inc,dec)***
RestoreOriginalCommunication	(avail, 100, –, –)
ChooseAlternativeCommunication	(avail, –, 10, –)
RestoreOriginalData	(integLevel, 100, –, –)
Reconfigure	(avail, 100, –, –) (integLevel, 90, –, –)
Stereotype: serviceModeDefinition, tagged-values: formula	
(GS0,0, (ExchangeInformation.avail > 90) & (ExchangeInformation.integLevel > 60) & (UpdateMap.avail > 90) & (UpdateMap.integLevel > 60)) (GS1,1, (ExchangeInformation.avail > 80) & (ExchangeInformation.integLevel > 60) & (UpdateMap.avail > 80) & (Update map.integLevel > 60)) (GS2,2, (ExchangeInformation.avail > 50) & (ExchangeInformation.integLevel > 30) & (UpdateMap.avail > 50) & (UpdateMap.integLevel > 30)) (GS3,3,)	

Table 2: Tagged-values specification of the running example

For each misuse case, survivability strategies need to be specified to mitigate the effect of the misuse case on the QoS of the essential services. In the running example, only recovery strategies are modelled. In particular, two different strategies are included to mitigate a jamming attack, i.e., setting an alternative communication with lower bandwidth (*ChooseAlternativeCommunication*) and the restoration of the original communication (*RestoreOriginalCommunication*). The *ManipulateInformation* misuse case is mitigated by restoring the original geodata before the attack, and the destruction of a node (*DestroyNode*) is overcome through hardware redundancy and software reconfiguration. Similar to misuse cases, recovery strategies are

annotated with *affects* tagged-values to specify how they affect the QoS indices of the essential services. For example, a reconfiguration, after node destruction, improves both the availability and the integrity level by setting them to the initial value and 90, respectively; whereas the two alternative recovery strategies from a jamming attack increase the availability differently: a 100% availability is guaranteed with the restoration of the original communication and an increase of 10% (concerning the initial value) is obtained in case of choosing the alternative communication mean.

Finally, the last annotation included in the MUCD specifies the system service modes (see *serviceModeDefinition* in Table 2). We use an ad-hoc syntax that enables to define each service mode as a triplet: *(name, severity, QoSlevel)*, where *name* is the name of the service mode, *severity* is the severity level (the higher is the level more degraded the service mode is) and *QoSlevel* is a boolean expression that specifies the QoS level of the system in terms of the thresholds for the QoS indices associated to the essential services. In the running example, the thresholds for both the QoS indices are minimum values, and there are four service modes: *GS0* is the best service mode that guarantees at least a 90% availability and at least a 60% integrity level of the two essential services. The other service modes provide degraded services: in particular, *GS1* guarantees a lower threshold for the availability (i.e., 80%) concerning *GS0*, whereas in *GS2* both the availability and integrity thresholds are lower than *GS1*. Finally, *GS3* is the worst service mode, and it does not guarantee a QoS minimum threshold. Thus, the specification of the system service mode enables to divide the value domain space of the QoS indices into different regions, where each region is defined by the *QoSlevel* of a service mode: the Figure 4 shows a Venn's diagram representation of such regions.

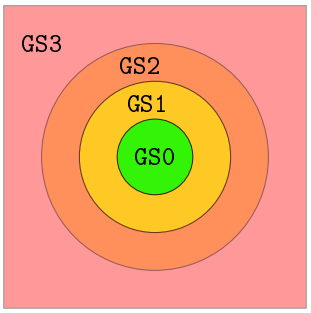

Figure 4: Venn's diagram representing system service modes

4.2. Generation phase

The work in [27] already dealt with the automatic generation of the SAM. In that paper, starting from a misuse cases diagram – that specifies the system essential services, the attacks and countermeasures – a SAM is produced by a model-to-model transformation. However, the approach has a limitation, it only considers single-stepped attacks/recoveries, i.e., the direct transition from a service mode to another is due to either a single attack occurrence or a single recovery execution. Although reasonable in some industrial contexts, this paper wants to overcome such hypothesis, then allowing to elicit complex attack-recovery sequences – that is sequences of multiple attack occurrences and multiple recovery executions that cause the direct transition from a service mode to another – which postulates the main motivation of this phase of the methodology.

In the generation phase, we use NuSMV [16] to produce the SAM. NuSMV is a powerful model-checking tool characterised by its simplicity in specifying both models and properties. To generate the SAM, the MUCD is first transformed into a Kripke model – MUCD formal model, in Figure 2 – and a first version of the SAM, which includes just the states representing the service modes. Next, the Kripke model is analysed by NuSMV and the counterexamples produced by the model checker are used to enrich the SAM with the state transitions.

MUCD-to-Kripke. The generation process from the MUCD to the Kripke model is depicted, at a high level of abstraction, in Figure 5 where a sample MUCD model is represented on the left and a scheme of the Kripke model is on the right. In the Figure, the directed arrows show the mapping between the model

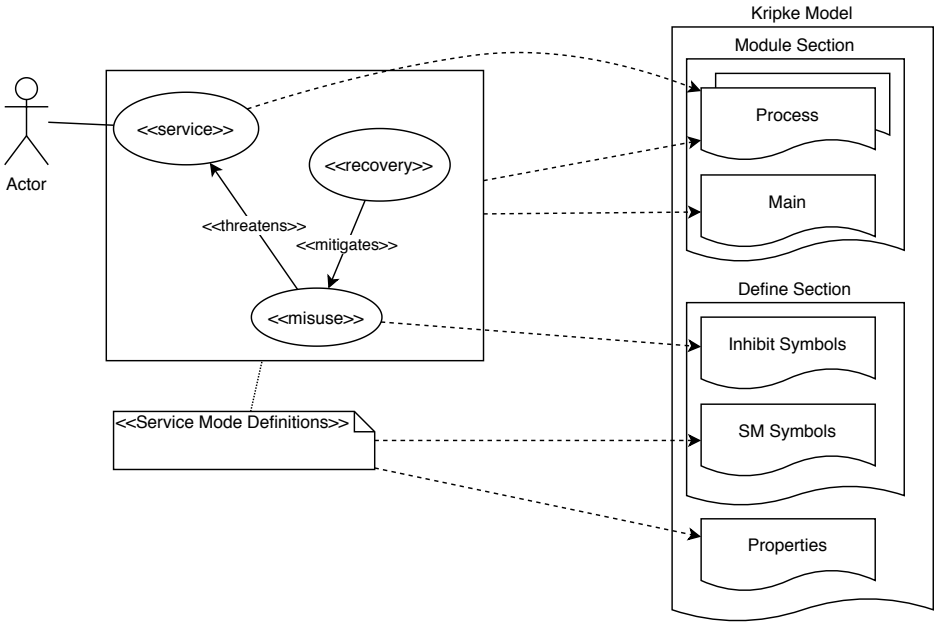

Figure 5: MUCD-to-Kripke: overview

elements of the MUCD and the three main sections of the Kripke model, that is: *Module Section*, *Define Section* and *Properties*.

The *Module Section* contains the description of the behaviour of the processes determining the evolution of the system: this description is apportioned among as many modules as the number of the *«service»* use cases, in the MUCD, and one main module. The *Define Section* contains the definition of symbols. In particular, there are two sets of symbols: the *Inhibit Symbols* – used in the main module – to permit the activation of one or more attacks, and the SM Symbols – used in the *Properties* section – that capture into boolean variables the system service modes. In the end, the *Properties* section reports a list of CTL formulas expressing the possibility to pass from a service mode to another.

In the following, we describe the generation process in details with the help of the running example of Figure 3 and the Listing 1, that reports an excerpt of the Kripke model automatically generated from Figure 3. The complete Kripke model can be found in the Appendix C.

Listing 1: MUCD Kripke model structure

```
-- Process modules
MODULE ExchangeInformation(p_Jamming,p_ChooseAlternativeCommunication,p_RestoreOriginalCommunication,
p_ManipulateInformation,p_RestoreOriginalData)
VAR
avail: 0..100;
integLevel: 0..100;
ASSIGN
init(avail) := 100;
init(integLevel) := 100;
next(avail) := case
(p_Jamming = TRUE) & (p_ChooseAlternativeCommunication = KO)
& (avail >= (10 + 0)): avail - 10;
(p_ChooseAlternativeCommunication = OK)
& (avail <= (100 - 10)): avail + 10;
(p_Jamming = TRUE) & (p_RestoreOriginalCommunication = KO)
& (avail >= (10 + 0)): avail - 10;
(p_RestoreOriginalCommunication = OK) & (avail < 100): 100;
TRUE: avail;
```

```
esac;
next(integLevel) := case
(p_ManipulateInformation = TRUE) & (p_RestoreOriginalData = KO)
& (integLevel > 50): 50;
(p_RestoreOriginalData = OK) & (integLevel < 100): 100;
TRUE: integLevel;
esac;

...
-- Main module
MODULE main
VAR
Jamming: boolean;
ManipulateInformation: boolean;
DestroyNode: boolean;
RestoreOriginalCommunication: {ENABLED, OK, KO};
ChooseAlternativeCommunication: {ENABLED, OK, KO};
RestoreOriginalData: {ENABLED, OK, KO};
Reconfigure: {ENABLED, OK, KO};
proc_ExchangeInformation: ExchangeInformation(Jamming,ChooseAlternativeCommunication,
RestoreOriginalCommunication,ManipulateInformation,RestoreOriginalData);
proc_UpdateMap: UpdateMap(DestroyNode,Reconfigure,ManipulateInformation,
RestoreOriginalData);
ASSIGN
init(Jamming) := FALSE;
next(Jamming) := case
(Jamming_inhibitor = TRUE): FALSE;
(Jamming_inhibitor = FALSE): {TRUE, FALSE};
esac;
init(ManipulateInformation) := FALSE;
next(ManipulateInformation) := case
(ManipulateInformation_inhibitor = TRUE): FALSE;
(ManipulateInformation_inhibitor = FALSE): {TRUE, FALSE};
esac;
init(DestroyNode) := FALSE;
next(DestroyNode) := case
(DestroyNode_inhibitor = TRUE): FALSE;
(DestroyNode_inhibitor = FALSE): {TRUE, FALSE};
esac;
init(RestoreOriginalCommunication) := KO;
init(ChooseAlternativeCommunication) := KO;
init(RestoreOriginalData) := KO;
init(Reconfigure) := KO;
next(RestoreOriginalCommunication) := case
(RestoreOriginalCommunication = KO) & ((Jamming = TRUE)): ENABLED;
(RestoreOriginalCommunication = ENABLED): {ENABLED, OK};
(RestoreOriginalCommunication = OK): KO;
TRUE: RestoreOriginalCommunication;
esac;
next(ChooseAlternativeCommunication) := case
(ChooseAlternativeCommunication = KO) & ((Jamming = TRUE)): ENABLED;
(ChooseAlternativeCommunication = ENABLED): {ENABLED, OK};
(ChooseAlternativeCommunication = OK): KO;
TRUE: ChooseAlternativeCommunication;
esac;
next(RestoreOriginalData) := case
(RestoreOriginalData = KO) & ((ManipulateInformation = TRUE)): ENABLED;
(RestoreOriginalData = ENABLED): {ENABLED, OK};
(RestoreOriginalData = OK): KO;
TRUE: RestoreOriginalData;
esac;
next(Reconfigure) := case
(Reconfigure = KO) & ((DestroyNode = TRUE)): ENABLED;
(Reconfigure = ENABLED): {ENABLED, OK};
(Reconfigure = OK): KO;
TRUE: Reconfigure;
esac;
-- Inhibit Symbols
DEFINE
Jamming_inhibitor := FALSE;
ManipulateInformation_inhibitor := FALSE;
DestroyNode_inhibitor := FALSE;

-- SM Symbols
DEFINE
GSO := (proc_ExchangeInformation.avail > 90)
& (proc_ExchangeInformation.integLevel > 60) & (proc_UpdateMap.avail > 90)
```

```
& (proc_UpdateMap.integLevel > 60);
GS1 := !(GS0) & (proc_ExchangeInformation.avail > 80)
& (proc_ExchangeInformation.integLevel > 60) & (proc_UpdateMap.avail > 80)
& (proc_UpdateMap.integLevel > 60);
GS2 := !(GS0 | GS1) & (proc_ExchangeInformation.avail > 50)
& (proc_ExchangeInformation.integLevel > 30) & (proc_UpdateMap.avail > 50)
& (proc_UpdateMap.integLevel > 30);
GS3 := !(GS0 | GS1 | GS2);

-- Properties
CTLSPEC AG (GS0 -> AX(!GS1))
...
CTLSPEC AG (GS3 -> AX(!GS2))
```

First, there are as many modules as use cases stereotyped *«service»* in the MUCD. The parameters of a module are the names of the attacks and recovery actions related to the use case, i.e., the misuses that threat the use case and the recoveries that mitigate the misuses. Each module is then responsible for determining the evolution of the QoS indices specified in the *«service»* as a response to the values specified for the attacks and recoveries.

The second part is the main module that instantiates all the attacks and recoveries in the MUCD model with the following behaviour: attacks are represented by boolean variables (i.e., `TRUE` if the attack is launched, otherwise `FALSE`); recoveries are represented by three-valued variables (i.e., `KO` if the recovery is not active, `ENABLED` if a triggering attack has been launched, but the recovery is not executed, yet, and `OK` if the recovery is executed). The body of the main module correlates the evolution of the attacks (`KO` $\longrightarrow$ `OK`) and of the recoveries (`KO` $\longrightarrow$ `ENABLED` $\longrightarrow$ `OK`). Furthermore, the main module instantiates the attack-related Kripke modules passing the attack/recovery variables to the corresponding use case modules as actual parameters. The usage of NuSMV's modules for the modelling of the behaviour of the *«service»* use cases is not motivated by the need of instantiating these modules more than once in the main module; but rather by choice of respecting a modular approach and easing the generation process. Moreover, the main module instantiates the process modules by passing as actual parameters the misuse and recovery variables since misuse and recoveries have global scope (i.e., they must be seen from all the service processes).

The third part of the NuSMV model is made of two `DEFINE` sections that are related to the definition of: (1) attack inhibitor variables, used to inhibit one or more attack occurrences in fine-grained analyses (see Section 5); (2) service mode variables used to understand if the system is in one service mode or another.

Concerning the latter, there are as many boolean variables as service modes, which are defined according to the *QoSlevel* boolean expressions in the tagged values of the *«serviceModeDefinition»* stereotype. The *severity* values of the service modes define a total ordering relation of the service modes that is translated into an expression by taking into account the precedence between the variables themselves. As an example, if there are two service modes, `Gx` and `Gy` with severity of `Gy` greater than the severity of `Gx`, such service modes are translated into two variables as in Listing 2:

Listing 2: Defining auxiliary variables

```
Gx := expr_x;
Gy := (!Gx) & expr_y;
```

where `expr_x` and `expr_y` are, respectively, the *QoSlevel* boolean expressions associated to the service modes `Gx` and `Gy`.

Both these groups of symbols are introduced for technical reasons. They simplify, respectively: 1) the switching between the MUCD formal model used in the generation and the verification phases (see Section 5 for further details); 2) the generation of the properties to check since, without defining such symbols, the properties should report the whole expressions with QoS indices.

The last part defines the properties to check. There is one property per transition in the SAM, hence, if we have n service modes, there will be $n*(n-1)$ transitions and properties to check. Each property computes the sequence of events that brings from a service mode `Gx` to a service mode `Gy`. In order to compute such sequence, we need to negate it in the form of a CTL expression — i.e., it is always true that starting from `Gx`, all the next steps present `!Gy`, where the conditions `Gx` and `Gy` are the truth of the variables as defined above. The CTL formula for checking the "`Gx-to-Gy`" property is then expressed as in the Listing 3:

Listing 3: Defining CTL formula for the `Gx-to-Gy` property

```
CTLSPEC AG (Gx -> AX(!Gy))
```

Counterexamples-to-SAM. The SAM is a UML state machine, where the states represent the system service modes, and the transitions allow the system to evolve through service modes. The statuses of the SAM are directly generated from the MUCD, considering the *serviceModeDefinition*, whereas the transitions between states are added from the results provided by NuSMV from the checking of the Kripke model.

In particular, for each service mode in the *serviceModeDefinition*, a state is generated and stereotyped as *«mode»*; each state is annotated with the *severity* tagged value, representing the severity of the service mode. The transitions between states are generated by considering the list of counterexamples that are produced by NuSMV with the checking of the CTL formulas, for the `Gx-to-Gy` properties, defined in the Kripke model. In the case that the `Gx-to-Gy` property is considered true by the model-checker, no counterexample is found and thus there is no feasible sequence of attacks and/or repairs between the two service modes `Gx` and `Gy`. Otherwise, the model checker produces a detailed description of the steps from `Gx` to `Gy`.

Such a description is parsed according to an EBNF grammar. By constructing a proper parser and semantic analyser, the attacks/recoveries contained in the counterexample are filtered and then used to annotate the transition in the SAM, with the *path* tagged value of the *«scenario»* stereotype[6]. A path is a sequence of misuse cases/recovery strategies states that cause the change of service mode, where each path item is specified by a triplet *(service,value,step)*: *service* is the name of the misuse case/recovery strategy, *value* is its state and *step* is the global system state (see Tables A.7 and A.8 of the Appendix A).

The Listing 4 reports an excerpt of the NuSMV output related to our running example, whereas Figure 6 shows the generated SAM in its graphical form.

Listing 4: Sample of a counterexample

```
-- specification AG (GS3 -> AX !GS2) is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 12.1 <-
Jamming = FALSE
ManipulateInformation = FALSE
DestroyNode = FALSE
RestoreOriginalCommunication = KO
ChooseAlternativeCommunication = KO
RestoreOriginalData = KO
Reconfigure = KO
proc_ExchangeInformation.avail = 100
proc_ExchangeInformation.integLevel = 100
proc_UpdateMap.avail = 100
proc_UpdateMap.integLevel = 100
DestroyNode_inhibitor = FALSE
ManipulateInformation_inhibitor = FALSE
Jamming_inhibitor = FALSE
GS3 = FALSE
GS2 = FALSE
GS1 = FALSE
GS0 = TRUE
-> State: 12.2 <-
Jamming = TRUE
DestroyNode = TRUE
-> State: 12.3 <-
Jamming = FALSE
DestroyNode = FALSE
RestoreOriginalCommunication = ENABLED
ChooseAlternativeCommunication = ENABLED
Reconfigure = ENABLED
proc_ExchangeInformation.integLevel = 100
proc_ExchangeInformation.avail = 90
proc_UpdateMap.avail = 0
proc_UpdateMap.integLevel = 10
```

[6]The filter consists in cutting the sequence between the relevant service modes (e.g., from `Gx` to `Gy`) also purging the sequence from all the model variables evolution not related to attacks and/or recoveries.

```
GS3 = TRUE
GS0 = FALSE
-> State: 12.4 <-
ManipulateInformation = TRUE
RestoreOriginalCommunication = OK
ChooseAlternativeCommunication = OK
Reconfigure = OK
-> State: 12.5 <-
ManipulateInformation = FALSE
RestoreOriginalCommunication = KO
ChooseAlternativeCommunication = KO
RestoreOriginalData = ENABLED
Reconfigure = KO
proc_ExchangeInformation.avail = 100
proc_ExchangeInformation.integLevel = 50
proc_UpdateMap.avail = 100
proc_UpdateMap.integLevel = 90
GS3 = FALSE
GS2 = TRUE
```

The screenshot, in Figure 6 on the right, shows the property panel of the Eclipse-Papyrus tool, with the *path* value (not complete) associated to the transition *T_GS3_GS2* – from the service mode *GS3* to the service mode *GS2*, with lower severity. The complete path has been manually added in the note symbol attached to the transition. In particular, the path discovered by the model-checker represents the situation where both a *Jamming* and a *DestroyNode* has occurred and the corresponding recovery actions become enabled (step 0); in the next step (step 1), a *ManipulateInformation* attack is launched and, in the meanwhile, the recovery actions for the attacks previously occurred are executed; finally, recovery actions are deactivated (step 2). The effect of the recovery actions is to re-establish the 100% availability of the two essential services and increase the integrity level of *updateMap*; however, the attack in step 1 affects the integrity level of the *ExchangeInformation* which remains equal to 50%. Thus, the reached service mode *GS2* is better than *GS3*, but it is still degraded.

Observe that, the paths found by the model-checker maybe not realistic in the context of the system under analysis. For instance, according to the approach discussed above both the recovery strategies *RestoreOriginalCommunication* and *ChooseAlternativeCommunication* are executed, whereas it seems straightforward that the execution of just the former is sufficient to improve the availability of the essential services. In the verification phase, the CPS analyst can perform a fine-grained analysis to check whether both are necessary, or just one of them is sufficient to improve the QoS indices. The framework is also open for fine-grained automatic analysis that are future works for this paper.

5. Verification phase

5.1. Properties and the query template repository

Once the system services, threats, strategies and service modes have been specified, and the survivability assessment model (SAM) automatically generated, then the system is ready for verification purposes. To this end, we have collected a set of survivability properties. Although large, the set is not exclusive but expandable. Most of these requirements belong to the survivability analysis field since they test properties related to system recovery. Table 3 presents the properties that are expressed as queries that can be proved against the SAM by a solver. Each property has a unique identifier (first column), a name (third column) and specifies a query template (fourth column) that will be eventually instantiated to the SAM. The query is expressed in natural language (English), and it is characterised by input parameters (fifth column) that may represent either service modes, misuse cases or recovery strategies. Depending on the property to be assessed, a different type of result will be returned by the solver (sixth column) that is a boolean value (i.e., true/false), a service mode or a scenario (i.e., a sequence of misuse cases or recovery strategies). The complete set of properties conforms what we call the query template repository (QTR) in the `surreal` framework.

Table 3 shows the properties arranged according to their *kind* (second column), which guides the interests of the analyst in the verification phase. In particular, the *Security level* properties focus the analysis on the recoverability of service modes, the *Threat* properties allow analysing the effect of threat occurrences on the

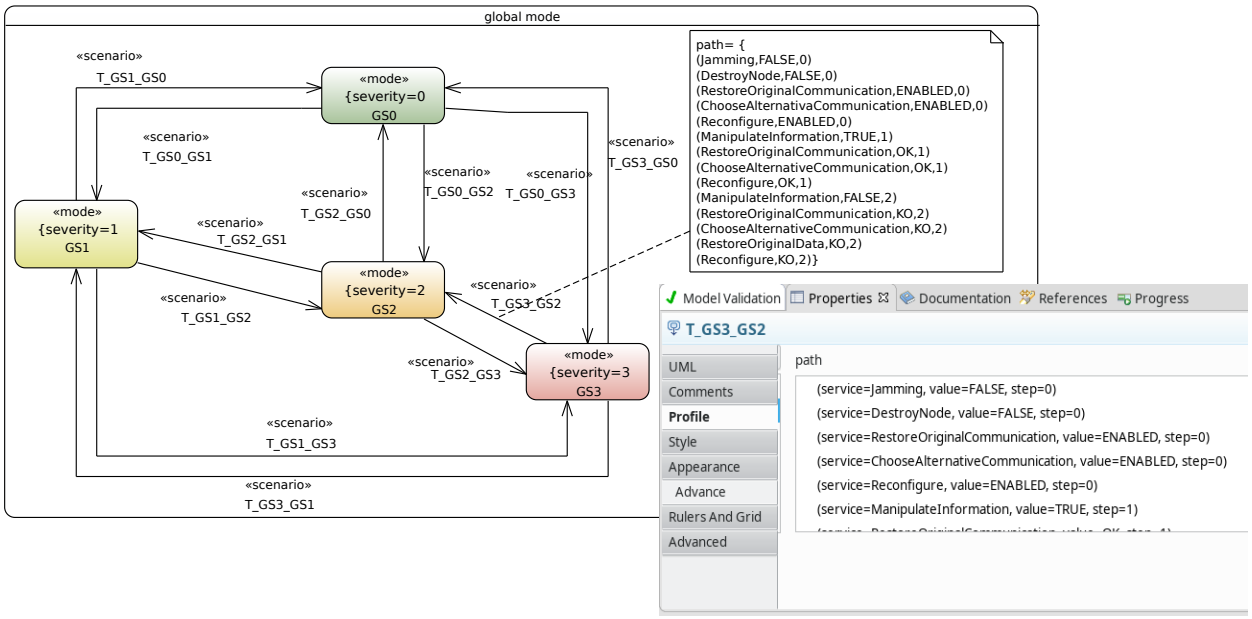

Figure 6: SAM of the running example

service modes and the *Mitigation* properties help the analyst in deciding on the survivability strategies to be developed in the system. Appendix B formalises all the properties implemented by the surreal framework and listed in Table 3.

The properties in the running example. Let us recall and interpret the three questions initially posed in Section 3:

1. Is it always possible to recover to the service mode that provides the highest quality (the *best service mode*)?
2. Let us suppose the system is offering the highest quality service and attacks that manipulate information occur, what is the service quality provided by the system after the attacks?
3. Let us consider a set of possible survivability strategies that can be used to improve the service quality in a degraded service mode. Which is the smallest subset that allows reaching the *best service mode*?

The first question can be pinpointed to *P1* and can be answered by instantiating the service mode *GS0* (see Figure 6) to the $\langle SMode \rangle$ input parameter of the query template. The second question can be addressed by instantiating *P8*, i.e., considering the *ManipulateInformation* misuse case (see Figure 3). Finally, the last question can be answered by instantiating three times *P14*, one for each degraded service mode, i.e., *GS1*, *GS2* and *GS3*.

ID	Kind	Name	Query template	Parameters type	Result type
P1	Security level	Reversibility	It is always possible to recover to the $\langle SMode \rangle$	$\langle SMode \rangle$ (a service mode)	boolean
P2	Security level	Strong reversibility	It is always possible to recover to the $\langle SMode \rangle$ without further degradation	$\langle SMode \rangle$ (a service mode)	boolean
P3	Security level	Recoverability	It is always possible to recover to $\langle DegradedMode \rangle$ from $\langle WorseDegradedMode \rangle$	$\langle DegradedMode \rangle$ (a service mode), $\langle WorseDegradedMode \rangle$ (a worse service mode)	boolean
P4	Security level	Strong recoverability	It is always possible to recover to $\langle DegradedMode \rangle$ from $\langle WorseDegradedMode \rangle$ without degradation	$\langle DegradedMode \rangle$ (a service mode), $\langle WorseDegradedMode \rangle$ (a worse service mode)	boolean
P5	Threat	Threat consequence (single occurrence)	Does a single occurrence of $\langle Misuse \rangle$ provoke a system degradation?	$\langle Misuse \rangle$ (a misuse case)	boolean
P6	Threat	Threat consequence (multiple occurrence)	Does (multiple) occurrence of $\langle Misuse \rangle$ provoke a system degradation?	$\langle Misuse \rangle$ (a misuse case)	boolean
P7	Threat	Security level threat impact (single occurrence)	Given the *best service mode*, which is the service mode reached by a single occurrence of $\langle Misuse \rangle$	$\langle Misuse \rangle$ (a misuse case)	$\langle SMode \rangle$ or $\emptyset$
P8	Threat	Security level threat impact (multiple occurrence)	Given the *best service mode*, which is the service mode reached by (multiple) occurrence of $\langle Misuse \rangle$	$\langle Misuse \rangle$ (a misuse case)	$\langle SMode \rangle$ or $\emptyset$
P9	Threat	Threat scenario	Given the *best service mode*, which is the smallest set of misuses that leads to $\langle DegradedMode \rangle$?	$\langle DegradedMode \rangle$ (a service mode)	$\langle Scenario \rangle$ or $\emptyset$
P10	Mitigation	Recovery feasibility	The strategy $\langle Recovery \rangle$ is feasible	$\langle Recovery \rangle$ (a recovery strategy)	boolean
P11	Mitigation	Multiple recovery	The strategies $\langle Recovery_1 \rangle, \ldots, \langle Recovery_N \rangle$ are always needed together	$\langle Recovery \rangle^*$ (a sequence of recovery strategies)	boolean
P12	Mitigation	Recovery mutual exclusion	The strategies $\langle Recovery_1 \rangle, \ldots, \langle Recovery_N \rangle$ are never carried out together	$\langle Recovery \rangle^*$ (a sequence of recovery strategies)	boolean
P13	Threat/ Mitigation	Threat/recovery effectiveness	Is the strategy $\langle Recovery \rangle$ effective to mitigate the threat $\langle Misuse \rangle$?	$\langle Recovery \rangle$ (a recovery strategy), $\langle Misuse \rangle$ (a misuse case)	boolean
P14	Mitigation	Best set of strategies in a service mode	Among the feasible strategies in $\langle DegradedMode \rangle$, which is the smallest set of strategies that leads to the *best service mode*?	$\langle DegradedMode \rangle$ (a service mode)	$\langle Scenario \rangle$ or $\emptyset$

Table 3: Survivability properties supported by the surreal framework

5.2. The surreal framework

As shown in Figure 2, for the verification phase the surreal framework encompasses several tools and documents, as follows.

Repository. The QTR is currently implemented as a JSON file deployed on a web server. The file completely describes each property and includes an extra line to indicate the URL of the *solver* for the query: the Listing 5 reports an excerpt of it, where only the template of property *P8* is shown.

Listing 5: Excerpt of the QTR (JSON implementation)

```
{
"queries":[
...
{"id":"P8",
"kind":"Threat",
"name":"Security level threat impact (multiple occurrence)",
"description":"Given the best service mode, which is the service mode reached by (multiple)"
"occurrence of <M>",
"paramlist"[
{"name":"M",
"stereotype":"Misuse"}
],
"result":"ServiceMode",
"solver":"http://localhost:8081/MultipleThreatImpactSolver.jar"},
...
]
}
```

Query instantiation GUI. The CPS analyst is now in charge of choosing the queries of interest and of specifying the parameters for binding — i.e., selecting the actual model elements according to the parameter type list. The query instantiation GUI guides the analyst in accomplishing this task for producing the query instantiation document (QID). Figure 7 depicts a snapshot of the GUI when executed for our running example analysis.

Engine and solvers. The engine is the core of the query analysis process. It is in charge to read the proper solvers in the QTR and call them based on the instantiated queries in the QID. More in details, the engine asks the QTR for the solver, retrieves the URL and dynamically loads it in the JVM[7] to enable the solution of the instantiated query according to the binding specified in the QID. As inputs, the solver receives the SAM and an instantiated query (i.e., a single element of the QID). The solution algorithms of each solver are different one from another, but three categories have been identified:

- `Type A`: some solvers simply explore the SAM (e.g., in understanding which are the essential services potentially recovered by a recovery action);
- `Type B`: others rely on the exhaustive state space exploration capability of the SAM by the model checker;
- `Type C`: others can combine the two approaches above in effective and efficient solution algorithms. As an example of this class, Algorithm 1 sketches a pseudo-java solution for the `solve` method of the "Best set of strategies in a service mode (P14)" solver. This solution method uses the first approach (`Type A`) to analyse the SAM model by searching for a sequence of transitions from the queried service mode to the best one (lines 2-5). In the case of negative response, there is no recovery strategy, and hence the function ends returning an empty scenario. Otherwise, a recovery strategy is generated by considering as actions the paths associated with the transitions of the sequence.

  However, this set may be not minimal[8]. Therefore, the second approach is applied (`Type B`) and the algorithm (lines 7-12) computes the smallest set of recovery actions by re-analysing the model with Bounded Model Checking (BMC) technique. More in detail:

[7] Java Virtual Machine [52]

[8] The set is not minimal because traditional model checkers do not guarantee the minimal length of the computed counterexamples: the application of further techniques is due (e.g., [26]).

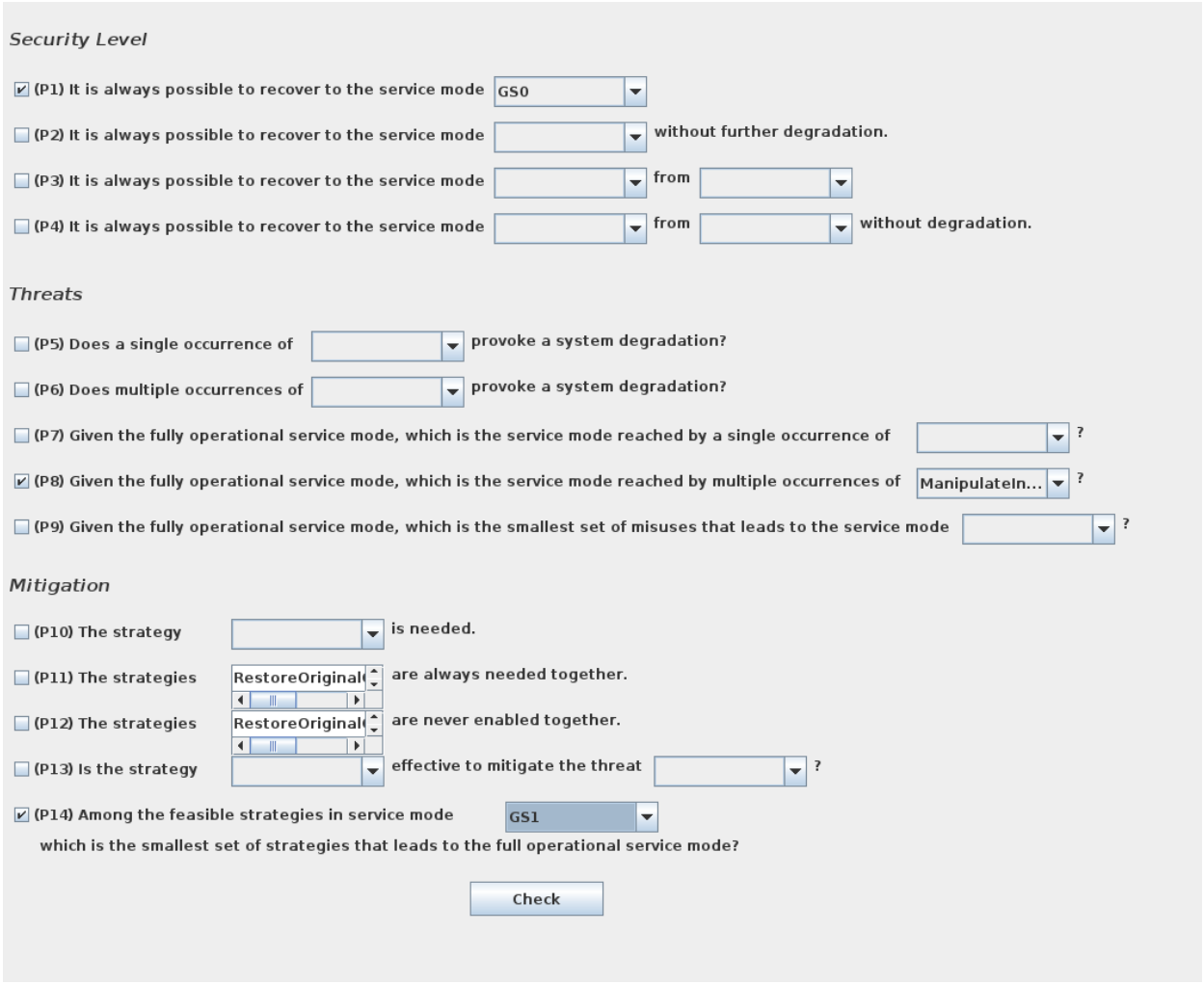

Figure 7: Query instantiation GUI executed on the running example

- @line 7: the MUCD formal model is generated using a new transformation component which is built based on the one developed in the Generation phase. These components differ in the number of properties to check (i.e., in P14 query template the model checker is asked to compute the path from `degradedModeName` to `bestModeName`);
- @line 8: the MUCD formal model is analysed. In this case, the difference is just in the command line for launching NuSMV. The **-bmc** flag is just added (inside the called method);
- @lines 9-10: they create and invoke the proper method of the specific query template post-processor to separate the requested answer among the other counterexamples;
- @lines 11-12: at these lines, the transition from `degradedModeName` to `bestModeName` is extracted, parsed, cleaned and added to the returning Scenario.

It is worth noticing that the Query Template solvers of `Type B` and `Type C` require to analyse the MUCD formal model again; the *MUCD-to-Kripke* transformation, described in Sub-section 4.2, is engineered to reuse most of the Kripke model automatically obtained in the generation phase. However, in the verification phase, the model checker is not used to generate the labelled transitions of the SAM, but to verify a property

on the SAM. Hence, the *Properties* section of the Kripke model is different from the one generated before. Furthermore, while the *Process Section* is the same, few differences are present in the *Define Section* in case the property deals with one single attack (e.g., P8). In this case, the evolution of the original MUCD formal model is different because all the attacks can not fire, but the one that is the subject of the analysis. To this aim, the Inhibit Symbols of the *Define Section* are changed setting to `TRUE` all of them but the one related to the attack that is free to fire.

Finally, and according to Figure 2, the called solver returns the results for the engine to generate a textual report. All the reports related to all the queries in the QID are collected by the engine that returns them to the user in terms of an assessment report (AR). Figure 8 depicts a snapshot of such a GUI for our running example analysis. Thus, the feedback provided to the analysis are the following answers:

- `P1(SM=GS0):true`. It is always possible to recover to the best service mode.
- `P8(M=ManipulateInformation):[GS2]`. When manipulation information attacks occur in the best service mode, then the system degrades to service mode *GS2*.
- `P14(S=GS1):@0:(P)RestoreOriginalCommunication->OK`. The best service mode can be restored from the degraded service *GS1* by carrying out the only *RestoreOriginalCommunication* strategy.

Algorithm 1 The *Best set of strategies in a service mode* solution process

```
1: procedure RESULT: SOLVE(query: QueryInstantiation, model: SAMHandler)
2:     retval = new ScenarioResult()
3:     degradedModeName = query.getBindingEntry("S")
4:     bestModeName = model.getBestServiceMode()
5:     reachable = model.existingNotDegradingPath(degradedMode,bestMode)
6:     if reachable == True then
7:         P14Transformation t = new P14Transformation(model,degradedModeName,bestModeName)
8:         String report = t.execute()
9:         Postprocessor pp = new P14Processor()
10:        pp.buildEvolution(report)
11:        Transition extended = pp.getTransition(degradedModeName,bestModeName)
12:        retval.load(extended)
       return retval
```

On the extensibility of the solvers. The solution already presented for the solvers enables easy extensibility of the tool by allowing developers to define their query templates and related solvers. As supported by the UML class diagram in Figure 9, the implementation of a solver is limited to the classes in the `solver.specific` package. More in detail, a solver should use and/or extend only some classes in the `surreal.engine` and `surreal.samgen` packages of the surreal framework, as depicted by the class hierarchy.

The main classes that a solver developer must implement are mainly related to: (1) the core of its solving algorithm — `SpecificSolvers`'s `solve` method —, (2) the usage of the `SAMHandler` containing the services able to query the SAM enhanced model (`Type A` and `Type C` solvers), (3) the implementation of both `SpecificTransformation` and `SpecificPostProcessor` in `Type B` and `Type C`. Solvers which respectively extend the classes of the surreal framework — `Transformation` is in charge of generating the SAM while `PostProcessor` parses the results of the model checker execution. Finally, as explained previously, each solver returns the results, concretely an object of the class `Result`, for the engine to generate a textual report as in Figure 8.

6. The case study

This section describes the application of the approach to a more complex case study in order to exhibit its potentialities in real contexts. The considered system is a smart car in the domain of intelligent

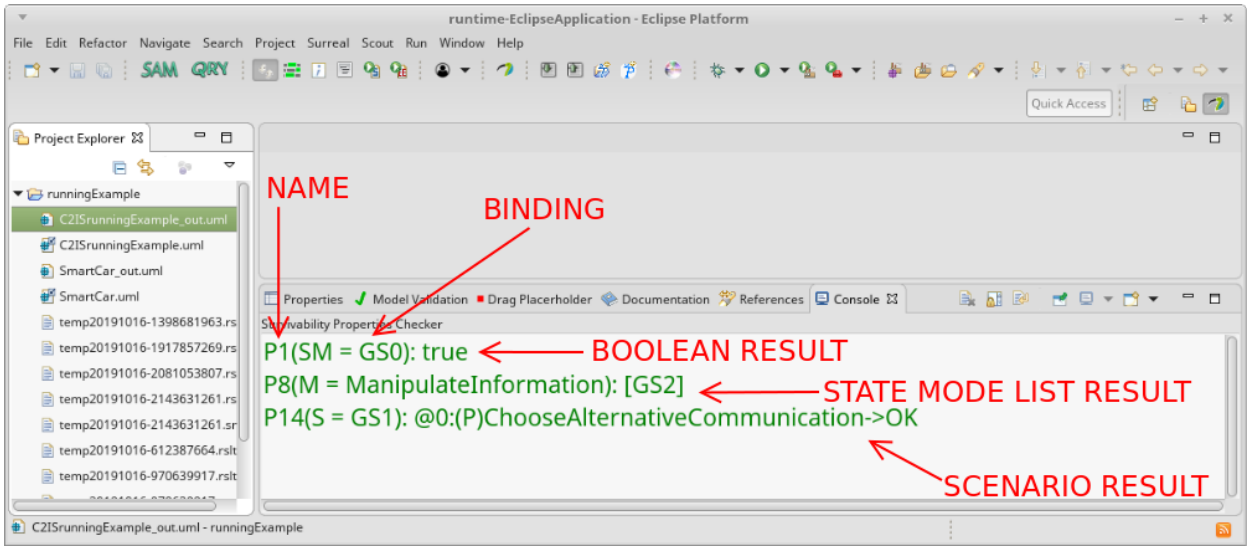

Figure 8: The running example - Assessment Report

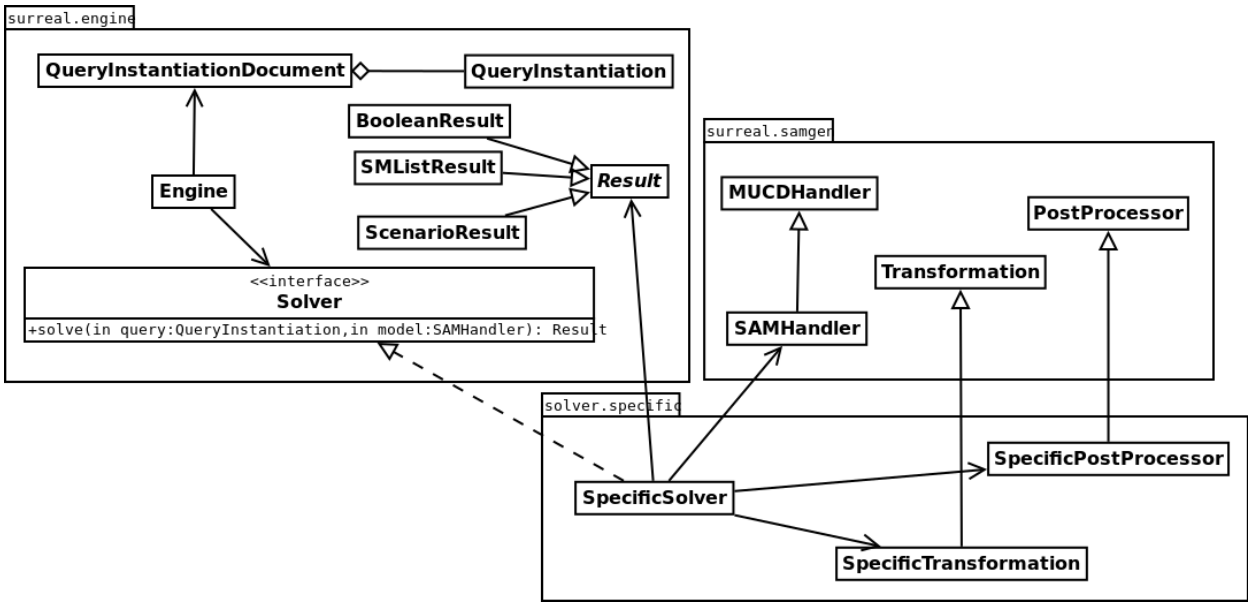

Figure 9: Class diagram for extending solvers in the surreal framework

transportation systems. The case study has been extensively described in [62], where readers can find the complete description of the system and a survey of known attacks against it.

Modern smart cars are equipped with many sensors, actuators and a set of control units (Electronic Control Units - ECUs) that are able to manage both mechanical and electrical components, such as braking, transmission, airbags, infotainment, emergency call and adaptive cruise control. These components allow the introduction of innovative smart control and assisting subsystems, such as Autonomous Driving Systems (ADS), Adaptive Cruise Control (ACC), collision avoidance and emergency vehicle notification systems. All these systems rely on data collected by on-board sensors that automatically generate novel control actions

to maintain speed and safety distance, to immediately brake the car, to alert the driver with messages transmitted by emergency vehicles and so on. Smart cars also integrate an Internet connection (mainly through a data SIM card) for infotainment services to enable an in-car WiFi connection.

A single bus (Controlled Area Network - CAN) – introduced in the '80s to reduce the car wiring costs and to share information among the different subsystems – connects all the devices with the ECU. Furthermore, wireless technologies enable communication with other vehicles (known as Vehicle-to-Vehicle communications) and with the traffic infrastructures (Vehicle-to-Infrastructure communications). A physical connection (On-Board Diagnostics Socket - ODB) is also present to provide physical access to the whole system. Typically, all these subsystems and components do not include any security mechanism making the smart car vulnerable to various types of attacks.

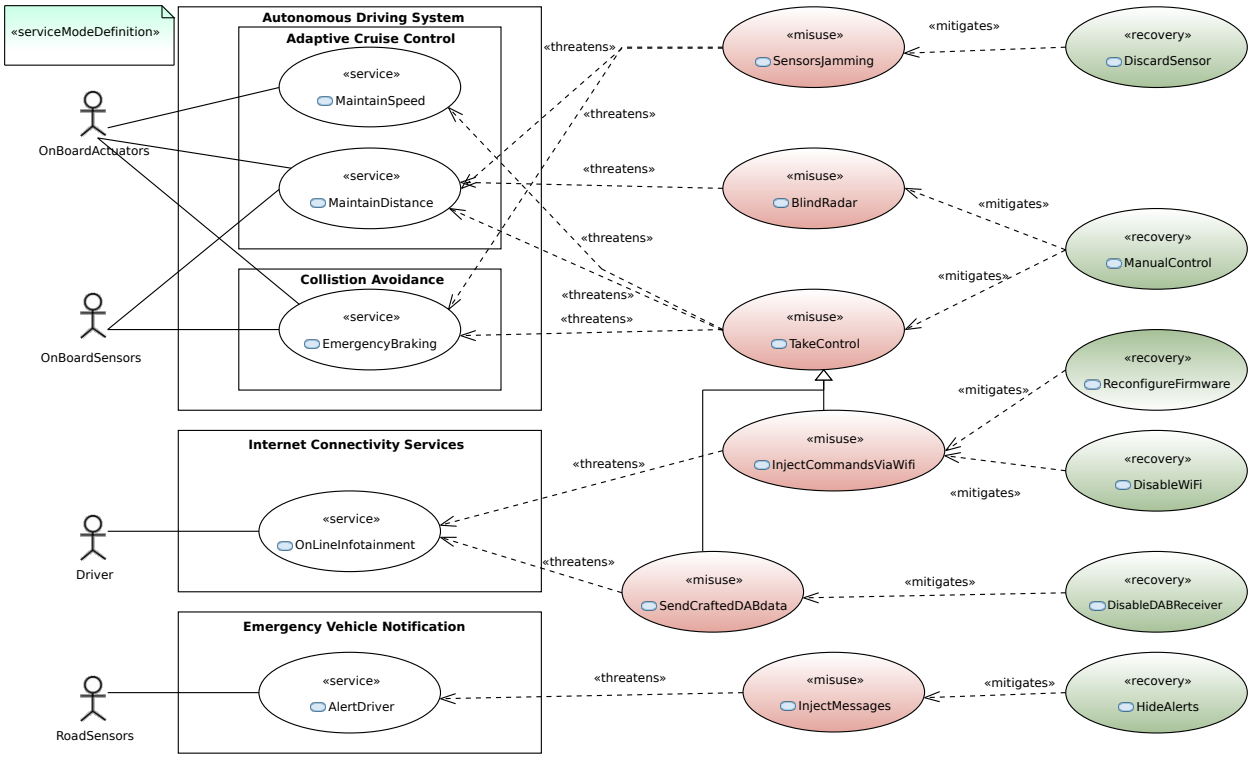

Figure 10: Misuse case of the case study.

Misuse case diagram. The misuse case diagram, shown in Figure 10, represents a set of subsystems of the smart car at a high level of abstraction. As said previously, the depicted diagram is a UML use case diagram where mainly the use cases are annotated as `service`, `misuse` or `recovery` to identify respectively essential services, threat scenarios and recovery strategies. The ADS consists of two subsystems: the ACC and the collision avoidance subsystems. The former is in charge of providing two essential services *MaintainSpeed* and *MaintainDistance*, whereas the latter provides the *EmergencyBraking*. The first two services are needed by the ACC to regulate the speed of the car to automatically keep a minimum distance from the preceding vehicles. Both these services rely on the information provided by the on-board sensors, and specifically either on a radar, or on a laser detector, or on a camera, and are able to brake the car when it is approaching a slower vehicle, and then accelerates when the traffic condition permits it. These services have been modelled as UML use cases annotated with the stereotype `service`. The third service is needed to activate the emergency braking of the car (with the maximum breaking strength) when an obstacle is detected in the proximity of the car to avoid collisions. The other two sub-systems i.e., Internet connectivity services and

Emergency vehicle notification, provide respectively the *OnLineInfotainment* and the *AlertDriver* essential services. The former is initiated by the driver and it allows him/her to easily control all those systems such as GPS navigation system, radio, music playing and smartphone integration using simple and intuitive commands. The latter is waked up by road sensors and it shows alerts about the presence of emergency vehicles with the rights of way in the proximity of the car. Also these two services are use cases of the misuse case diagram, annotated as `service`. Furthermore, the diagram in Figure 10 contains six misuse cases (use cases annotated with the stereotype `misuse`) which can threaten the essential services, and six recovery strategies which can be applied to recover the system (use cases annotated with the stereotype `recovery`). Each misuse case and recovery will be described in the following of this paragraph. The dependencies among use cases are also annotated with the stereotype `threatens` to model the relationships between misuse cases and services, and with the stereotype `mitigates` to model the relationships between recoveries and misuses.

According to the ISO 26262 – Functional Safety for Road Vehicles [35] – the essential services have to be classified considering the Automotive Safety Integrity Levels (ASILs). There are four levels in the ASIL classification: from ASIL A to ASIL D, where ASIL A represents the lowest requirement on the service whereas ASIL D is the highest. In addition, ASIL QM means that there are not safety requirements associated to the service. As reported in Table 4, the services considered in this case study have all an impact on safety. In particular, the *indices* tagged-values associated to the essential services correspond to the highest value of ASIL (i.e., 100), indicating the highest safety requirement, where the ASIL enumeration domain $\{QM, A, B, C, D\}$ has been mapped to an integer interval `0..100` as follows: $QM = 0, A = [1, 25], B = [26, 50], C = [51, 75], D = [76, 100]$. Moreover, the availability of the *On-LineInfotainment* and *AlertDriver* essential services is also considered. This index, in fact, is expressed as an integer interval `0..6`, which specifies the number of "nines" after the comma, i.e., $0 = 99\%, 1 = 99.9\%, ..., 6 = 99.999999\%$. Just to interpret the availability values, considering a mission time of 1 year, the corresponding downtimes are about $30s$ (avail=4), $3s$ (avail=5) and $315ms$ (avail=6). Thus, the availability initial values assigned to the essential services are the highest ones.

There are six misuse cases in the diagram in Figure 10 (represented by the use cases annotated with the stereotype `misuse`). *SensorsJamming* represents an attack aimed at slowing down (or interrupting) the distance measurement by means of interference signals, thus it threatens the maintenance of the distance from the preceding vehicle and the capability of activating the emergency braking. *BlindRadar* also threatens the *MaintainDistance* service as the previous attack, but it is aimed to prevent the correct measurement of the distance from the preceding vehicle. Each time one of these two attacks are launched, they decrease the initial ASIL level of the affected services by 20%. *TakeControl* represents an attack aiming at taking the control of the ADS system, e.g., managing speed and distance from the outside. A successful attack of this type completely reduces the ASIL of the affected services to 0. *InjectCommandsViaWifi* and *SendCraftedDABdata* are special cases of *TakeControl*, in the sense that they also aim to take control of the ADS of the car but they act, indirectly, by exploiting vulnerabilities of the *OnLineInfotainment* service. The former represents an attack conducted by a nearby adversary who wants to take control of the ADS of the car by injecting malicious commands through the in-car WiFi; the latter models an attack conducted by an adversary who creates a fake radio station and sends crafted Digital Audio Broadcasting (DAB) signals to compromise the on-line infotainment of the smart cars in the range. Both these attacks reduce the ASIL of the affected services to 50% and decrease the availability of the *OnLineInfotainment* service of 1 nine. Finally, *InjectMessages* represents an attack to the emergency vehicle notification system, where an adversary injects messages in the traffic control system of high-traffic roads. This attack reduces the ASIL of the *AlertDrive* service to 50% and decreases its availability of 1 nine.

In the diagram depicted in Figure 10, six survivability strategies – modelled as use cases annotated with the stereotype `recovery` – are defined to mitigate the effects of the misuse cases and recover the system. *DiscardSensor* is introduced to mitigate the sensor jamming attack, and its effect on the QoS indices of the services affected by the misuse case is to increase their ASIL of 20%. An extreme recovery strategy consists in giving the manual control to the driver (*ManualControl*), which can help in case of adversaries who take the control of the ADS of the car or blind radar, but with a negative impact on the availability of the *OnLineInfotainment* service, reducing it to 0. To contrast the injection of commands via Wi-Fi, applicable strategies are to reconfigure the firmware (*ReconfigureFirmware*) or to disable the in-car

Stereotype: service	***Tagged-values: indices (name,kind,values,initial)***
MaintainSpeed	(ASIL,integerInterval, 0..100,100)
MaintainDistance	(ASIL,integerInterval, 0..100,100)
EmergencyBraking	(ASIL,integerInterval, 0..100,100)
On-LineInfotainment	(ASIL,integerInterval, 0..100,100) (avail, integerInterval, 0..6, 6)
AlertDriver	(ASIL,integerInterval, 0..100,100) (avail, integerInterval, 0..6, 6)
Stereotype: misuse	***Tagged-values: affects (index,set,inc,dec)***
SensorsJamming	(ASIL, –, –, 20)
BlindRadar	(ASIL, –, –, 20)
TakeControl	(ASIL, 0, –, –)
InjectCommandsViaWiFi	(ASIL, 50, –, –) (avail, –, –, 1)
SendCraftedDABData	(ASIL, 50, –, –) (avail, –, –, 1)
InjectMessages	(ASIL, 50, –, –) (avail, –, –, 1)
Stereotype: recovery	***Tagged-values: affects (index,set,inc,dec)***
DiscardSensor	(ASIL, –,20,–)
ManualControl	(avail, 0, –, –)
ReconfigureFirmware	(ASIL, 100, –,–) (avail, –, 1, –)
DisableWiFi	(ASIL, 100, –, –)
DisableDABReceiver	(ASIL, 100, –, –)
HideAlerts	(avail, 0, –, –)
Stereotype: serviceModeDefinition, tagged-values: formula	
(Optimal,0,(MaintainSpeed.ASIL > 75) & (MaintainDistance.ASIL > 75) & (EmergencyBraking.ASIL > 75) & (AlertDriver.ASIL > 75) & (AlertDriver.avail > 5) & (On-LineInfotaintment.ASIL > 75) & (On-LineInfotaintment.avail >5)) (DegradedICandEN,1,(MaintainSpeed.ASIL > 75) & (MaintainDistance.ASIL > 75) & (EmergencyBraking.ASIL > 75) & (AlertDriver.ASIL > 50) & (AlertDriver.avail > 4) & (On-LineInfotaintment.ASIL > 50) & (On-LineInfotaintment.avail >4)) (DegradedASDSafety,2,(MaintainSpeed.ASIL > 50) & (MaintainDistance.ASIL > 50) & (EmergencyBraking.ASIL > 50) & (AlertDriver.ASIL > 75) & (AlertDriver.avail > 5) & (On-LineInfotaintment.ASIL > 75) & (On-LineInfotaintment.avail >5)) (Degraded,3,(MaintainSpeed.ASIL > 50) & (MaintainDistance.ASIL > 50) & (EmergencyBraking.ASIL > 50) & (AlertDriver.ASIL > 50) & (AlertDriver.avail > 4) & (On-LineInfotaintment.ASIL > 50) & (On-LineInfotaintment.avail >4)) (VeryDegraded,4,(MaintainSpeed.ASIL > 25) & (MaintainDistance.ASIL > 25) & (EmergencyBraking.ASIL > 25) & (AlertDriver.ASIL > 25) & (AlertDriver.avail > 4) & (On-LineInfotaintment.ASIL > 25) & (On-LineInfotaintment.avail >4)) (WorstDegradation, 5)	

Table 4: Tagged-values specification of the case study.

WiFi (*DisableWiFi*). In both these cases, the strategies improve the ASIL of the services that is reset to 100%, while the availability of the *OnLineInfotainment* service increases by 1 nine only in case of firmware reconfiguration. Instead, if the adversary tries to crack the DAB signal, it is possible to disable the DAB receiver (*DisableDABReceiver*), which increases the ASIL to 100%. At last, to recovery the system from fake messages injected against the emergency vehicle notification system, it is possible to hide the alerts (*HideAlerts*) but, as for the manual control, the availability of the alert service goes to 0.

At the bottom of Table 4, we reported the definition of the six system service modes which vary from the *Optimal* to the *WorstDegradation*. In particular, the *Optimal* service mode guarantees ASIL D (i.e., greater than 75) for all the essential services and an availability of at least 6 nines for the *OnLineInfotainment* and

AlertDriver services. Starting from the best service mode, two distinct degradations are possible: degradation of Internet Connectivity and/or of the emergency vehicle notification services (*DegradedICandEN*), and degradation of ADS services (*DegradedASDSafety*). The intersection of both these degradations is a different service mode, named *Degraded*. A last acceptable degraded service mode is *VeryDegraded*, where essential services guarantee a minimum ASIL B, and the *OnLineInfotainment* and *AlertDriver* services guarantee an availability of at least 5 nines. The last service mode, *WorstDegradation*, does not guarantee minimum thresholds of the QoS indices. The inclusion relationship between the six services modes is represented by the Venn's diagram shown in Figure 11.

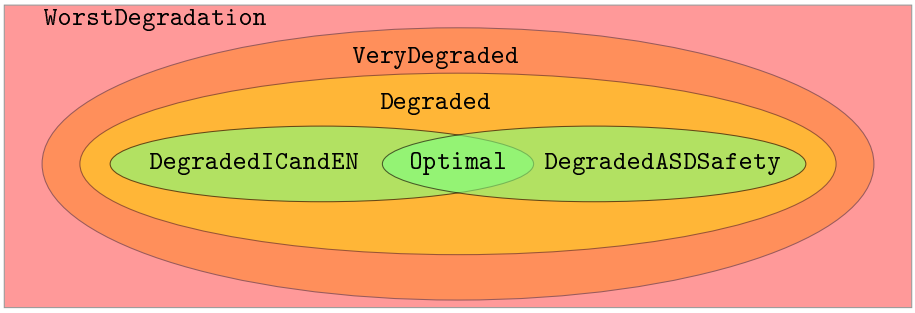

Figure 11: Venn's diagram representing smart car service modes

The SAM model. Figure 12 depicts the SAM model that has been automatically generated from the misuse case diagram of Figure 10 by the `SAM generation` tool. The generated state machine has 6 states, corresponding to the service modes previously described, and 24 transitions, corresponding to the possibilities of the system to pass through the different service modes.

For sake of space, we do not report the NuSMV model of the case study; in the following paragraph, we describe the counterexamples to SAM, which are automatically generated to verify some properties of interest.

System verification. The model previously described has been verified against the three properties P1 (Reversibility), P8 (Security level threat impact) and P14 (Best set of strategies in a service mode). Excluding the trivial case of the service mode *WorstDegradation*, we obtained that the reversibility is guaranteed for all the service modes but *VeryDegraded*. In fact, the reader can verify that all the corresponding states except *VeryDegraded* in the SAM model (depicted in Figure 12) have incoming transitions from all the states with higher severity.

More complex results have been found for the remaining two properties. In fact, the analysis of the security level threat impact (P8) highlights that multiple occurrences of TakeControl lead the system in the state *WorstDegradation*. Instead, each of the remaining misuse cases sets ASIL of services to 50 and decreases their availability one at a time up to unacceptable levels.

At last, the analysis of the best set of strategies in a service mode (P14) highlights that the recovery *ReconfigureFirmware* is the shortest way — alone or in combination to other countermeasures — to recover the system from the states *DegradedICandEN*, *Degraded*, *VeryDegraded* and *WorstDegradation*. In fact, this recovery rises up the ASIL to 100, which is the level required by the best service mode.

In the last part of this paragraph, we report the execution times of the conducted analysis. All the results have been obtained working on a laptop equipped with Intel(R) Core(TM) i7-2677M CPU @ 1.80GHz and 4 Gbyte of RAM. The SAM generation has been performed in 25.32 seconds. The analysis of P1 was almost instantaneous (since it is of `Type A`). Concerning P8 and P14 properties, Table 5 reports the minimum, maximum and average time. The difference between the execution times of these two analysis is due to the need in P14 query to find the shortest counterexample, that we implement with the BMC analysis — generally more complex and time-consuming than other kinds of analysis.

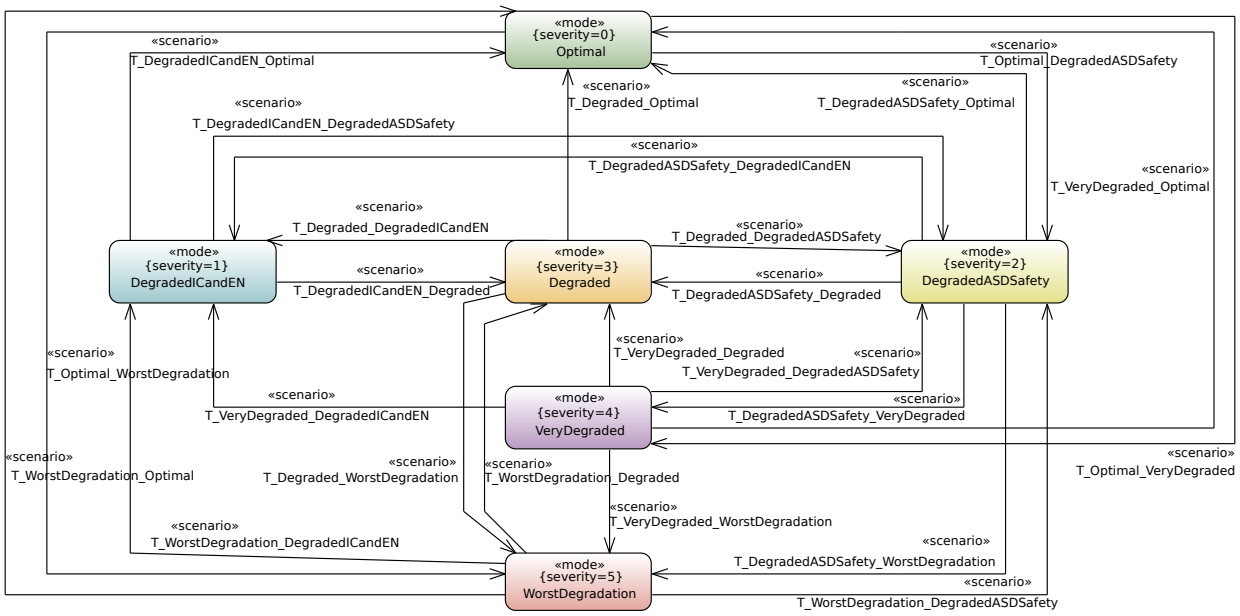

Figure 12: SAM of smart car (automatically generated).

property	*execution time [sec]*		
	min	**avg**	**max**
P8	0.01	0.05	0.07
P14	26.89	36.21	51.04

Table 5: Execution times

These values demonstrate the applicability of the proposed approach also on complex real-world case studies, as the smart car could be. Up to now, the tool analyses each query separately: future tool optimization actions would explore batch executions of properties on the same formal model (to exploit single generation of the state space) and/or parallel executions.

7. Assumptions and threats to validity

The presented methodology relies, in our view, on realistic assumptions discussed throughout the paper and here summarized. Mainly, the application of the `surreal` framework is enabled by the usage of profiled UML models representing both, use cases and misuse cases, but also service modes of the overall system need to be known and well-defined. These assumptions can be easily satisfied in realistic applications when domain experts analyse the survivability from such an elicitation of possible misuse cases. Starting from this model, the framework is able to conduct automatic verification, so giving answers to common and important properties in the survivability field. Summarizing, the main hypothesis supporting the methodology are the following:

HP1: misuses may occur concurrently and are independent;

HP2: misuses are considered as carried out in a single step;

HP3: recovery strategies are independent;

HP4: recovery strategies are considered as single-step actions and are able to recover from a degraded state;

HP5: the QoS levels of each system status, as well as the impact of the attacks and the improvement due to the mitigation strategies, are based on the requirement engineer domain knowledge.

When using `surreal`, we have observed that the framework is able to cope with the increasing size of the models in realistic scenarios. In fact, the tree-like structure offered by the Eclipse plugin helped to the scalability of the approach. Moreover, when the number of misuses increases, the model can be organized in packages for a better visualization and management. In any case, `surreal` leverages, for modelling, state of the art Eclipse tools and scales according to them. If some threat exists for these tools it will apply to `surreal` as well. The application of the methodology and framework to the proposed case study has shown us the effective management of a complex real-world application and its validation. In fact, the manual modelling of the case study has not been a hard task and the framework well supported this activity. So, we are also confident that more complex case studies can also be managed. Even if we have investigated fourteen properties, the `surreal` framework can be extended with additional ones and corresponding solvers. The latter can be carried out by downloading the source code of the framework, freely available on the GitHub repository of the project, and developing a custom package, following the instructions given in Section 5. According to our opinion, such development of a new solver is not a challenging task for a medium experienced Java developer. At last, it is important to remark that the real impact of attacks and recovery strategies during system service may not correspond, one by one, to those estimated by the approach. In fact, estimated values can be affected by contextual factors that may not be completely captured by the model initially. This threat is common to all modelling approaches, especially in the initial stages of the life-cycle. In fact, there is extensive research on modelling under uncertainties [5]. However, as the development process progresses, the engineer is equipped with better knowledge and tools, such as system prototypes, that help to gain insights for calibrating the model, then obtaining more accurate estimations. In particular, for threat elicitation, discrepancies can derive from many sources, even from a wrong threat analysis conducted by experts. Fortunately, in our proposal, the automation capabilities provided by `surreal`, both at the modelling and solution levels, make feasible to re-apply the approach several times in short time-span. This is useful for updating the UML models, as well as the estimations, as such knowledge is acquired.

8. Discussion and conclusion

The relevant contribution of this work, in our view, is the capability of verifying survivability properties. Moreover, the verification is automatic, as well as the generation of the model where they are proved, i.e., the SAM. The SAM is a by-product of the security specification developed by the analyst, mostly in terms of misuse cases.

Currently, we have conceptually investigated fourteen properties, all of them implemented in the framework. As explained in Sub-section 5.2, the set of properties is extensible, also in terms of tool implementation - Figure 9 - which confers great potential to the approach. The properties are expressed abstractly, in terms of security concepts, e.g., threats, which means that they need to be interpreted in the problem domain by an expert, e.g., the CPS analyst in the case study. Such abstraction level makes more robust the proposal since it is not bounded to a particular problem domain or application, but can address CPS at large. However, it posed the challenge of formalising each property, as explained in Appendix B.

The methodological and theoretical contexts proposed by the work have indeed been made practicable. The `surreal` framework has been developed for the Eclipse platform [63] and applying the model-driven [64, 58] paradigm for the `SAM generation tool`. The rest of the tools, `query instantiation GUI`, `engine` and `solvers` have been developed in Java [52] language and they are also integrated in the framework, which can be freely dowloaded from `https://github.com/stefanomarrone/surreal`.

Concerning the surreal architecture, the design choice to implement a lightweight analysis framework with just the `Solver` interface has the following advantages: a) it gives the freedom to the user to implement his/her specific solvers with the preferred technology; b) an open architecture supports extensibility of the framework with third-party contributions, of new solvers; c) ad-hoc solvers, focussing on a single Query

Template, are more manageable to design, test and load at run-time in the engine during the verification phase.

The `surreal` framework and the overall approach have been validated through a CPS case study in the automotive domain. The system is exposed to attacks that threaten the safety of the passengers. The case study makes it clear at least: a) the complexity of survivability specifications in CPS, as manifested in the specifications in Table 4 and in the complex definition of the transitions in Figure 12, b) the important role of the CPS analyst for understanding the problem domain, for example, when s/he needs to classify essential services considering the ASILs, and c) the need for verifying security properties. Although we decided not to present all the details of the system verification, we assess three interesting properties, and among others, we conclude that for different service modes the system could not reach an optimal mode, i.e., the safety is not guaranteed.

As future work, we want to test the tool framework in various case studies and with new properties at hand. Other research efforts will be put on upgrading the methodology and tools by considering quantitative aspects. Services, misuses and recovery actions can also be annotated with quantitative information capturing the probability of their occurrence and/or success. Furthermore, we plan to integrate this quantitative information into the MUCD formal model, and use quantitative model checking frameworks (e.g., PRISM [41]). The final aim is to provide to the CPS analyst not only information about the possible sequence of events but also to estimate the probability of their occurrence, to boost his/her decisional power.

Appendix A. Survivability profile

The survivability profile, see Figure 1, is structured in three separate packages:

- *Misuse case extensions*: it includes stereotypes to specify threats/attacks and protections in UML use case diagrams, as a result of threats modelling and survivability analysis of the system. Table A.6 lists all the stereotypes of this package.
- *Survivability Assessment Model (SAM) extensions*: it includes stereotypes to specify service modes and changes of service modes in UML state machine diagrams. This package depends on the previous one. Table A.7 lists all the stereotypes of this package.
- *Survivability types*: it includes a set of datatypes/enumeration used to define the previous stereotypes. Table A.8 lists all types in detail.

Table A.6: Misuse case extensions.

Stereotype/Tag	*Extension*	*Description*
misuse	–	(*Generalization: serviceMS*) A misuse case represents an use case from the point of view of an hostile actor.
Tag		
affects		It is a set of consequences on the services threatened by the misuse case. Each consequence is expressed in terms of the (negative) impact on the value of a QoS index.
successProb		It is the probability of succeeding.
attackDelay		It is the mean time between the attack launching and the intrusion occurrence.
misuser	Actor	A misuser is an hostile actor: it can be an attacker, an unaware user who uses the system in the wrong way or the environment that hinders the sytem being in operation.
mitigates	Dependency	It is a direct relationship between a strategy that aim at mitigating a misuse case and the misuse case.

Continued on next page

Table A.6 – continued from previous page

resistance	–	(*Generalization: strategy*) It is a strategy aimed at repealing an attack or masking an accidental fault [20].
recovery *Tag* affects MTTR	–	(*Generalization: strategy*) It is a strategy aimed at restoring the service after an intrusion or failure [20]. It is a set of consequence on the services that were threatened by the misuse cases mitigated by the strategy. Each consequence is expressed in terms of (positive) impack on the value of a QoS index. (Mean Time To Recover) It is the time to undergo recovery.
recognition	–	(*Generalization: strategy*) It is a strategy aimed at detecting an attack/fault and evaluating the damage [20].
service *Tag* indices	Use Case	It is an essential service provided by the system that must survive even when it is infiltrated, compromised or crashed [20]. It is a set of Quality of Service (QoS) requirements associated to the service. Each QoS requirement is expressed in terms of a performance, dependability or security index.
serviceModeDefinition *Tag* formula	Constraint	It is a specification of the global service modes of the sytem. It is a set of global service modes. Each service mode is a logical expression that defines the QoS requirements of the system in terms of minimum/maximum acceptable values for the QoS indexs associated to the essential services.
serviceSM	Use Case	It is an abstract stereotype that may represent either a misuse case or a survivability strategy.
strategy *Tag* successsProb	Use Case	(*Generalization: serviceSM*) It is an abstract stereotype that represents a survivability strategy. Probability of succeeding.
threatens	Dependency	It is a direct relationship between a misuse case that threatens an essential service and the service.

Table A.7: Survivability Assessment Model extensions.

Stereotype/Tag	*Extension*	*Description*
mode *Tag* severity	State	A global service mode. The severity level of the service mode: the higher is the level the more degraded mode is.
scenario *Tag* path	Transition	The system changes from a global service mode to another global service mode. It is a sequence of misuse cases/recovery strategies that causes the change of a service mode.

Table A.8: Survivability types.

Datatype/Attribute	*Description*
MSActivation	It is a misuse/strategy activation.
Attribute	
service	The misuse case or the survivability strategy.
value	The state value associated to the service.
step	The step number, representing a state of the global system, that includes this service value.
affectConsequence	It is the consequence on a QoS index.
Attribute	
index	The name of the QoS index.
set	The value set to the QoS index.
inc	The increment to the current value of the QoS index (positive consequence due to a recovery strategy).
dec	The decrement to the current value of the QoS index (negative consequence due to a misuse case).
duration	Mean duration.
Attribute	
value	Time value.
unit	Time unit.
index	QoS index.
Attribute	
name	The name of the QoS index.
kind	The type of value domain.
values	The value domain.
initial	The initial value.
indexKind	Index value domain
integerInterval	An integer interval.
enum	Enumeration.

Appendix B. Formalisation of the properties

This appendix provides a formalisation of the survivability properties that the `surreal` framework offers currently. We realised the need for formalising the properties early while studying their application even to simple examples, and definitively while implementing them. In fact, some properties where being interpreted by each researcher slightly different. However, it took discussions to find a better way of carrying out the formalisation. Finally, we based the formalisation on two definitions: a) the SAM (survivability assessment model), i.e., a state machine, given in Definition 1; and b) the service mode reachability, given in Definition 2.

Definition 1. *A state machine is a tuple* $\mathcal{SM} = \langle S, T, \mathcal{C}, [\rangle, \pi, \omega \rangle$*, where:*

- S *is the set of service modes.*
- T *is the set of transitions representing the changes of service mode.*
- $\mathcal{C} = \mathcal{T} \cup \mathcal{R}$ *is the set of (mis)use cases, which is partitioned into the set of threats* $\mathcal{T}$ *(misuse cases), and the set of recovery strategies* $\mathcal{R}$ *(strategy use cases).*
- *The service mode change function* $[\rangle : T \to S \times S$ *associates to each transition* $t \in T$ *a pair of service modes* (s, s')*, where* s *is the leaving service mode and* s' *is the entering service mode. The change from service mode* s *to service mode* s' *is denoted by:* $s[t\rangle s'$*.*

- *The priority function $\pi : S \to \mathbb{N}$ assigns a natural number to each service mode: the lower is the priority, the more degraded is the service mode. The priority function π defines a total ordering of the service modes and $s_0 \in S$ such that $\pi(s_0) = max_{s \in S}\pi(s)$ is the best service mode.*
- *Let $\mathcal{O} = \mathcal{T} \times \{TRUE, FALSE\} \bigcup \mathcal{R} \times \{OK, ENABLED, KO\}$ be the set of the possible threat/recovery strategy states, the function $\omega : T \to \mathcal{O}^n$ assigns to each transition $t \in T$, where $s[t\rangle s'$, a sequence of threat/recovery strategy occurrences $(\tau_1, \ldots, \tau_n)$, where $\tau_i \in \mathcal{O}$, that causes the change of service mode from s to s'. We denote by $|(\tau, \omega(t))|$, the number of occurrences of $\tau \in \mathcal{O}$ in the sequence occurrence $\omega(t)$ of transition $t \in T$.*

Definition 2. *Let $\mathcal{SM} = \langle S, T, \mathcal{C}, [\rangle, \pi, \omega\rangle$ be a state machine, a service mode $s' \in S$ is reachable from a service mode $s \in S$ if there exists a sequence of transitions $\sigma \equiv (t_0, t_1, \ldots, t_n)$, where $t_i \in T, i = 0, \ldots, n$, that leads from s to s', i.e.:*

$$s[t_0\rangle s_0[t_1\rangle s_1[\ldots\rangle s_{n-1}[t_n\rangle s' \equiv s[\sigma\rangle s'.$$

In the following, each query template or property, listed in Table 3, is formally defined using the notation just introduced.

Property 1 (Reversibility). Let $\mathcal{SM} = \langle S, T, \mathcal{C}, [\rangle, \pi, \omega\rangle$ be a state machine and $x \in S$ a service mode. Then, it is always possible to recover to x iff $\forall s \in \mathcal{SM} \setminus \{x\}$ such that $\pi(s) < \pi(x), \exists \sigma \equiv (t_0, t_1, \ldots, t_n) : s[\sigma\rangle x$.

Property 2 (Strong reversibility). Let $\mathcal{SM} = \langle S, T, \mathcal{C}, [\rangle, \pi, \omega\rangle$ be a state machine and $x \in S$ a service mode. Then, it is always possible to recover to x without further degradation iff $\forall s \in \mathcal{SM} \setminus \{x\}$ such that $\pi(s) < \pi(x), \exists \sigma \equiv (t_0, t_1, \ldots, t_n) : s[\sigma\rangle x \equiv s[t_0\rangle s_0[t_1\rangle s_1[\ldots\rangle s_{n-1}[t_n\rangle x$ and $\forall i = 0, \ldots, n-1 : \pi(s) < \pi(s_i)$.

Property 3 (Recoverability). Let $\mathcal{SM} = \langle S, T, \mathcal{C}, [\rangle, \pi, \omega\rangle$ be a state machine, $s, s' \in S$ two service modes, where $\pi(s') < \pi(s)$. Then, it is always possible to recover to s from s' iff $\exists \sigma \equiv (t_0, t_1, \ldots, t_n) : s[\sigma\rangle s'$.

Property 4 (Strong recoverability). Let $\mathcal{SM} = \langle S, T, \mathcal{C}, [\rangle, \pi, \omega\rangle$ be a state machine and $s, s' \in S$ two service modes, where $\pi(s') < \pi(s)$. Then, it is always possible to recover to s from s' without further degradation iff $\exists \sigma \equiv (t_0, t_1, \ldots, t_n) : s'[\sigma\rangle s \equiv s'[t_0\rangle s_0[t_1\rangle s_1[\ldots\rangle s_{n-1}[t_n\rangle s$ and $\forall i = 0, \ldots, n-1 : \pi(s') < \pi(s_i)$.

Property 5 (Threat consequence – single occurrence). Let $\mathcal{SM} = \langle S, T, \mathcal{C}, [\rangle, \pi, \omega\rangle$ be a state machine and $\tau \in \mathcal{T}$ a threat. Then, a single occurrence of $\tau \equiv (\tau, TRUE)$ provokes a system degradation iff $\exists t \in T : s_i[t\rangle s_j$ such that:

1. $\pi(s_j) < \pi(s_i)$,
2. $|(\tau, \omega(t))| = 1$, and
3. $\forall \tau' \in \mathcal{T} \setminus \{\tau\} : (\tau', TRUE) \notin \omega(t)$.

Property 6 (Threat consequence – multiple occurrence). Let $\mathcal{SM} = \langle S, T, \mathcal{C}, [\rangle, \pi, \omega\rangle$ be a state machine and $\tau \in \mathcal{T}$ a threat. Then, a multiple occurrence of $\tau \equiv (\tau, TRUE)$ provokes a system degradation iff $\exists t \in T : s_i[t\rangle s_j$ such that:

1. $\pi(s_j) < \pi(s_i)$,
2. $|(\tau, \omega(t))| \geq 1$, and
3. $\forall \tau' \in \mathcal{T} \setminus \{\tau\} : (\tau', TRUE) \notin \omega(t)$.

Property 7 (Security level threat impact – single occurrence). Let $\mathcal{SM} = \langle S, T, \mathcal{C}, [\rangle, \pi, \omega\rangle$ be a state machine, $s_0 \in S$ the best service mode and $\tau \in \mathcal{T}$ a threat. Let us denote by $target(s_0) = \{s \in \mathcal{SM} \mid \exists t \in T : s_0[t\rangle s\}$, the set of service modes that can be reached directly from s_0. Then, the set $\hat{S} \subseteq target(s_0)$:

$$\hat{S} = \{s \mid \exists t \in T : s_0[t\rangle s,\ \exists\, (\tau, TRUE) \in \omega(t) : |((\tau, TRUE), \omega(t))| = 1,\ \nexists\, \tau' \in \mathcal{T} \setminus \{\tau\} : (\tau', TRUE) \in \omega(t)\}$$

is the set of service modes reached by a single occurrence of the threat $\tau \in \mathcal{T}$ from the best service mode s_0.

Property 8 (Security level threat impact – multiple occurrence). Let $\mathcal{SM} = \langle S, T, \mathcal{C}, [\rangle, \pi, \omega\rangle$ be a state machine, $s_0 \in S$ the best service mode and $\tau \in \mathcal{T}$ a threat. Let us denote by $target(s_0) = \{s \in \mathcal{SM} \mid \exists t \in T : s_0[t\rangle s\}$, the set of service modes that can be reached directly from s_0. Then, the set $\hat{S} \subseteq target(s_0)$:

$$\hat{S} = \{s \mid \exists t \in T : s_0[t\rangle s,\ \exists\, (\tau, TRUE) \in \omega(t) : |((\tau, TRUE), \omega(t))| \geq 1,\ \nexists\, \tau' \in \mathcal{T} \setminus \{\tau\} : (\tau', TRUE) \in \omega(t)\}$$

is the set of service modes reached by multiple occurrences of the threat $\tau \in \mathcal{T}$ from the best service mode s_0.

Property 9 (Threat scenario). Let $\mathcal{SM} = \langle S, T, \mathcal{C}, [\rangle, \pi, \omega\rangle$ be a state machine, $s_0 \in S$ the best service mode and $s \in S, s \neq s_0$ a service mode. Given a sequence of transitions $\sigma \equiv (t_0, \ldots, t_n)$ that leads from s_0 to s, i.e., $s_0[\sigma\rangle s$, the set $\mathcal{T}(\sigma) = \{\tau \in \mathcal{T} : \exists t_i \in \sigma, (\tau, TRUE) \in \omega(t_i)\}$ contains the threats that cause the service degradation to s from the best service s_0. Then, the smallest set of threats $\mathcal{T}^*$ that leads to s from s_0 satisfies the equality:

$$|\mathcal{T}^*| = \min_{\sigma : s_0[\sigma\rangle s} \{|\mathcal{T}(\sigma)|\}$$

Property 10 (Recovery feasibility). Let $\mathcal{SM} = \langle S, T, \mathcal{C}, [\rangle, \pi, \omega\rangle$ be a state machine and $\rho \in \mathcal{R}$ a recovery strategy. Then, ρ is feasible iff $\exists t \in T : s[t\rangle s'$ where $\pi(s) < \pi(s')$ such that $\exists (\rho, OK) \in \omega(t)$.

Property 11 (Multiple recovery). Let $\mathcal{SM} = \langle S, T, \mathcal{C}, [\rangle, \pi, \omega\rangle$ be a state machine and $\rho_1, \ldots, \rho_n \in \mathcal{R}$, n recovery strategies. Then, the n survivability strategies are always needed together iff $\forall t \in T : s[t\rangle s'$ where $\pi(s) < \pi(s')$ such that $\exists i \in \{1, \ldots, n\} :\ (\rho_i, OK) \in \omega(t) \implies \forall j \in \{1, \ldots, n\} \setminus \{i\} : (\rho_j, OK) \in \omega(t)$.

Property 12 (Recovery mutual exclusion). Let $\mathcal{SM} = \langle S, T, \mathcal{C}, [\rangle, \pi, \omega\rangle$ be a state machine and $\rho_1, \ldots, \rho_n \in \mathcal{R}$, a subset of recovery strategies. Given a transition $t \in T : s[t\rangle s'$, let us denote by $R(t) = \{\rho \in \mathcal{R} : (\rho, OK) \in \omega(t)\}$, the set of recovery strategies that causes the change of service mode from s to s'. Then, the subset of recovery strategies are never carried out together iff:

$$\nexists s \in S : \{\rho_1, \ldots, \rho_n\} \subseteq \bigcup_{t \in T : s[t\rangle s'} R(t).$$

Property 13 (Threat/recovery effectiveness). Let $\mathcal{SM} = \langle S, T, \mathcal{C}, [\rangle, \pi, \omega\rangle$ be a state machine, $\tau \in \mathcal{T}$ a threat and $\rho \in \mathcal{R}$ a recovery strategy. Then, the strategy ρ is effective to mitigate the threat τ iff

$$\forall s \in S, \forall t \in T : s[t\rangle s', (\tau, TRUE) \in \omega(t), \nexists (\tau', TRUE) \in \omega(t), \tau' \neq \tau \implies$$

$$\exists t' \in T : s'[t'\rangle s'', (r, OK) \in \omega(t'), \nexists (r', OK) \in \omega(t'), r' \neq r$$

where $\pi(s') < \pi(s'')$.

Property 14 (Best set of strategies in a service mode). Let $\mathcal{SM} = \langle S, T, \mathcal{C}, [\rangle, \pi, \omega\rangle$ be a state machine, $s_0 \in S$ the best service mode and $s_n \in S, s_n \neq s_0$ a service mode. Given a sequence of transitions $\sigma \equiv (t_n, \ldots, t_1)$ $(t_i \in T, i = 1, \ldots, n)$ that leads s_n to s_0, i.e., $s_n[t_n\rangle s_{n-1}[\ldots\rangle s_1[t_1\rangle s_0 \equiv s_n[\sigma > s_0$, where $\pi(s_i) < \pi(s_{i-1}), i = n, \ldots, 1$, let $\mathcal{R}(\sigma) = \{\rho \in \mathcal{R} \mid \exists t_i \in \sigma : (\rho, OK) \in \omega(t_i)\}$ be the set of recovery strategies that occurred in σ. Then, the smallest set of strategies $\mathcal{R}^*$ that leads to the best service mode satisfies the equality:

$$|\mathcal{R}^*| = \min_{\sigma : s[\sigma\rangle s_0} \{|\mathcal{R}(\sigma)|\}$$

where $|X|$ is the cardinality of the set X.

Appendix C. Kripke model of the running example

This appendix includes the complete Kripke model of the running example.

```
-- Process modules

MODULE ExchangeInformation(p_Jamming,p_ChooseAlternativeCommunication,p_RestoreOriginalCommunication,
                           p_ManipulateInformation,p_RestoreOriginalData)
VAR
        avail: 0..100;
        integLevel: 0..100;
ASSIGN
        init(avail) := 100;
        init(integLevel) := 100;
        next(avail) := case
         (p_Jamming = TRUE) & (p_ChooseAlternativeCommunication = KO) & (avail >= (10 + 0)): avail - 10;
         (p_ChooseAlternativeCommunication = OK) & (avail <= (100 - 10)): avail + 10;
         (p_Jamming = TRUE) & (p_RestoreOriginalCommunication = KO) & (avail >= (10 + 0)): avail - 10;
         (p_RestoreOriginalCommunication = OK) & (avail < 100): 100;
         TRUE: avail;
        esac;
        next(integLevel) := case
         (p_ManipulateInformation = TRUE) & (p_RestoreOriginalData = KO) & (integLevel > 50): 50;
         (p_RestoreOriginalData = OK) & (integLevel < 100): 100;
         TRUE: integLevel;
        esac;

MODULE UpdateMap(p_DestroyNode,p_Reconfigure,p_ManipulateInformation,p_RestoreOriginalData)
VAR
        avail: 0..100;
        integLevel: 0..100;
ASSIGN
        init(avail) := 100;
        init(integLevel) := 100;
        next(avail) := case
         (p_DestroyNode = TRUE) & (p_Reconfigure = KO) & (avail > 0): 0;
         (p_Reconfigure = OK) & (avail < 100): 100;
         TRUE: avail;
        esac;
        next(integLevel) := case
         (p_DestroyNode = TRUE) & (p_Reconfigure = KO) & (integLevel > 10): 10;
         (p_Reconfigure = OK) & (integLevel < 90): 90;
         (p_ManipulateInformation = TRUE) & (p_RestoreOriginalData = KO) & (integLevel > 50): 50;
         (p_RestoreOriginalData = OK) & (integLevel < 100): 100;
         TRUE: integLevel;
        esac;

-- Main module
MODULE main
VAR
        Jamming: boolean;
        ManipulateInformation: boolean;
        DestroyNode: boolean;
        RestoreOriginalCommunication: {ENABLED, OK, KO};
        ChooseAlternativeCommunication: {ENABLED, OK, KO};
        RestoreOriginalData: {ENABLED, OK, KO};
        Reconfigure: {ENABLED, OK, KO};
        proc_ExchangeInformation: ExchangeInformation(Jamming,ChooseAlternativeCommunication,
                                  RestoreOriginalCommunication, ManipulateInformation,RestoreOriginalData);
        proc_UpdateMap: UpdateMap(DestroyNode,Reconfigure,ManipulateInformation,RestoreOriginalData);
ASSIGN
        init(Jamming) := FALSE;
        next(Jamming) := case
                (Jamming_inhibitor = TRUE): FALSE;
                (Jamming_inhibitor = FALSE): {TRUE, FALSE};
        esac;
        init(ManipulateInformation) := FALSE;
        next(ManipulateInformation) := case
                (ManipulateInformation_inhibitor = TRUE): FALSE;
                (ManipulateInformation_inhibitor = FALSE): {TRUE, FALSE};
        esac;
        init(DestroyNode) := FALSE;
        next(DestroyNode) := case
                (DestroyNode_inhibitor = TRUE): FALSE;
                (DestroyNode_inhibitor = FALSE): {TRUE, FALSE};
        esac;
        init(RestoreOriginalCommunication) := KO;
```

```
        init(ChooseAlternativeCommunication) := KO;
        init(RestoreOriginalData) := KO;
        init(Reconfigure) := KO;
        next(RestoreOriginalCommunication) := case
                (RestoreOriginalCommunication = KO) & ((Jamming = TRUE)): ENABLED;
                (RestoreOriginalCommunication = ENABLED): {ENABLED, OK};
                (RestoreOriginalCommunication = OK): KO;
                TRUE: RestoreOriginalCommunication;
        esac;
        next(ChooseAlternativeCommunication) := case
                (ChooseAlternativeCommunication = KO) & ((Jamming = TRUE)): ENABLED;
                (ChooseAlternativeCommunication = ENABLED): {ENABLED, OK};
                (ChooseAlternativeCommunication = OK): KO;
                TRUE: ChooseAlternativeCommunication;
        esac;
        next(RestoreOriginalData) := case
                (RestoreOriginalData = KO) & ((ManipulateInformation = TRUE)): ENABLED;
                (RestoreOriginalData = ENABLED): {ENABLED, OK};
                (RestoreOriginalData = OK): KO;
                TRUE: RestoreOriginalData;
        esac;
        next(Reconfigure) := case
                (Reconfigure = KO) & ((DestroyNode = TRUE)): ENABLED;
                (Reconfigure = ENABLED): {ENABLED, OK};
                (Reconfigure = OK): KO;
                TRUE: Reconfigure;
        esac;

-- Inhibit Symbols
DEFINE
        Jamming_inhibitor := FALSE;
        ManipulateInformation_inhibitor := FALSE;
        DestroyNode_inhibitor := FALSE;

-- SM Symbols
DEFINE
        GS0 := (proc_ExchangeInformation.avail > 90) & (proc_ExchangeInformation.integLevel > 60) &
               (proc_UpdateMap.avail > 90) & (proc_UpdateMap.integLevel > 60);
        GS1 := !(GS0) & (proc_ExchangeInformation.avail > 80) &
               (proc_ExchangeInformation.integLevel > 60) & (proc_UpdateMap.avail > 80) &
               (proc_UpdateMap.integLevel > 60);
        GS2 := !(GS0 | GS1) & (proc_ExchangeInformation.avail > 50) &
               (proc_ExchangeInformation.integLevel > 30) & (proc_UpdateMap.avail > 50) &
               (proc_UpdateMap.integLevel > 30);
        GS3 := !(GS0 | GS1 | GS2);

-- Properties
CTLSPEC AG (GS0 -> AX(!GS1))
CTLSPEC AG (GS0 -> AX(!GS2))
CTLSPEC AG (GS0 -> AX(!GS3))
CTLSPEC AG (GS1 -> AX(!GS0))
CTLSPEC AG (GS1 -> AX(!GS2))
CTLSPEC AG (GS1 -> AX(!GS3))
CTLSPEC AG (GS2 -> AX(!GS0))
CTLSPEC AG (GS2 -> AX(!GS1))
CTLSPEC AG (GS2 -> AX(!GS3))
CTLSPEC AG (GS3 -> AX(!GS0))
CTLSPEC AG (GS3 -> AX(!GS1))
CTLSPEC AG (GS3 -> AX(!GS2))
```

Acknowledgments

This research was supported by the Spanish Ministry of Science, Innovation and Universities [ref. Medrese-RTI2018-098543-B-I00]. The author U. Gentile thanks the European Organization for Nuclear Research (CERN) where he started the activities described in the paper. Finally, the authors want to thank the reviewers and editors for their invaluable help to improve the paper.

References

[1] IEC 61508: Functional Safety of Electrical/electronic/programmable Electronic Safety Related Systems (1998).

[2] Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems, CENELEC, 2011.

[3] I. Alexander, Misuse cases: use cases with hostile intent, IEEE Software 20 (2003) 58–66.
[4] D. Alrajeh, J. Kramer, A. Russo, S. Uchitel, Elaborating Requirements Using Model Checking and Inductive Learning, IEEE Transactions on Software Engineering 39 (2013) 361–383.
[5] Ayyub, Bilal and Klir, George, Uncertainty Modeling and Analysis in Engineering and the Sciences, Chapman & Hall, Taylor and Francis Group, 2006.
[6] P. Behm, P. Benoit, A. Faivre, J.M. Meynadier, Météor: A successful application of B in a large project, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 1708 (1999) 369–387.
[7] M. Benerecetti, R.D. Guglielmo, U. Gentile, S. Marrone, N. Mazzocca, R. Nardone, A. Peron, L. Velardi, V. Vittorini, Dynamic state machines for modelling railway control systems, Science of Computer Programming 133 (2017) 116–153.
[8] S. Bernardi, L. Dranca, J. Merseguer, A model-driven approach to survivability requirement assessment for critical systems, Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability 230 (2016) 485–501.
[9] S. Bernardi, J. Merseguer, D.C. Petriu, Model-Driven Dependability Assessment of Software Systems, Springer Berlin Heidelberg, 2013.
[10] M. Biagi, L. Carnevali, F. Tarani, E. Vicario, Model-based quantitative evaluation of repair procedures in gas distribution networks, ACM Trans. Cyber-Phys. Syst. 3 (2018) 19:1–19:26.
[11] A. Biere, A. Cimatti, E. Clarke, O. Strichman, Y. Zhu, Bounded Model Checking, Advances in Computers 58 (2003) 117–148.
[12] G. Biggs, T. Sakamoto, T. Kotoku, A profile and tool for modelling safety information with design information in SysML, Software & Systems Modeling 15 (2016) 147–178.
[13] T. Bures, D. Weyns, B. Schmer, E. Tovar, E. Boden, T. Gabor, I. Gerostathopoulos, P. Gupta, E. Kang, A. Knauss, P. Patel, A. Rashid, I. Ruchkin, R. Sukkerd, C. Tsigkanos, Software engineering for smart cyber-physical systems: Challenges and promising solutions, SIGSOFT Softw. Eng. Notes 42 (2017) 19–24.
[14] R. Calinescu, S. Kikuchi, K. Johnson, Compositional reverification of probabilistic safety properties for large-scale complex it systems, in: R. Calinescu, D. Garlan (Eds.), Large-Scale Complex IT Systems. Development, Operation and Management, Springer, Berlin, Heidelberg, 2012, pp. 303–329.
[15] B. Cheng, J. Atlee, Research directions in requirements engineering, FoSE 2007: Future of Software Engineering, pp. 285–303.
[16] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, NuSMV 2: An open source tool for symbolic model checking, Lecture Notes in Computer Science 2404 (2002) 359–364.
[17] J. Dörr, D. Kerkow, A.V. Knethen, B. Paech, Eliciting efficiency requirements with use cases, in: In Proceedings of the International Workshop on Requirements Engineering: Foundations of Software Quality (REFSQ'2003).
[18] A. Drago, S. Marrone, N. Mazzocca, R. Nardone, A. Tedesco, V. Vittorini, A model-driven approach for vulnerability evaluation of modern physical protection systems, Software and Systems Modeling 18 (2019) 523–556.
[19] D.A. Eisenberg, D.L. Alderson, M. Kitsak, A. Ganin, I. Linkov, Network foundation for command and control (c2) systems: Literature review, IEEE Access 6 (2018) 68782–68794.
[20] R.J. Ellison, R.C. Linger, T.A. Longstaff, N.R. Mead, Survivable network system analysis: A case study, IEEE Software 16 (1999) 70–77.
[21] P. Fiterău-Broştean, R. Janssen, F. Vaandrager, Combining Model Learning and Model Checking to Analyze TCP Implementations, in: A. Chaudhuri, Swaratand Farzan (Ed.), Computer Aided Verification, Springer International Publishing, Cham, 2016, pp. 454–471.
[22] I. Friedberg, K. McLaughlin, P. Smith, D. Laverty, S. Sezer, Stpa-safesec: Safety and security analysis for cyber-physical systems, Journal of Information Security and Applications 34 (2017) 183 – 196.
[23] S. Friedenthal, A. Moore, R. Steiner, A Practical Guide to SysML: Systems Modeling Language, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
[24] A. Fuxman, M. Pistore, J. Mylopoulos, P. Traverso, Model checking early requirements specifications in Tropos, in: Proceedings Fifth IEEE International Symposium on Requirements Engineering, 2001, pp. 174–181.
[25] A. Gargantini, C. Heitmeyer, Using model checking to generate tests from requirements specifications, SIGSOFT Softw. Eng. Notes 24 (1999) 146–162.
[26] P. Gastin, P. Moro, Minimal Counterexample Generation for SPIN, in: D. Bošnački, S. Edelkamp (Eds.), Model Checking Software, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 24–38.
[27] U. Gentile, S. Bernardi, S. Marrone, J. Merseguer, V. Vittorini, A model driven approach for assessing survivability requirements of critical infrastructures, Journal of High Speed Networks 23 (2017) 175–186.
[28] G. Georg, K. Anastasakis, B. Bordbar, S.H. Houmb, I. Ray, M. Toahchoodee, Verification and trade-off analysis of security properties in UML system models, IEEE Trans. Software Eng. 36 (2010) 338–356.
[29] M. Ghallab, D. Nau, P. Traverso, Automated Planning: Theory and Practice, 2004.
[30] M. Gharib, P. Lollini, A. Ceccarelli, A. Bondavalli, Engineering functional safety requirements for automotive systems: A cyber-physical-social approach, in: D. Yu, V. Nguyen, C. Jiang (Eds.), 19th IEEE International Symposium on High Assurance Systems Engineering, HASE 2019, Hangzhou, China, January 3-5, 2019, IEEE, 2019, pp. 74–81.
[31] K. Goertzel, L. Feldman, Software Survivability: Where Safety and Security Converge, in: AIAA Infotech@Aerospace Conference, 6-9 April, 2009.
[32] S.H. Houmb, G. Georg, S.H. St, F. Collins, R. France, An integrated security verification and security solution design trade-off analysis, in: Integrating Security and Software Engineering: Advances and Future Visions, IDEA Group Publishing, 2007, pp. 190–219.
[33] G. Howard, M. Butler, J. Colley, V. Sassone, Formal analysis of safety and security requirements of critical systems

supported by an extended stpa methodology, in: 2017 IEEE European Symposium on Security and Privacy Workshops (EuroS PW), pp. 174–180.
[34] A. Humayed, J. Lin, F. Li, B. Luo, Cyber-physical systems security – a survey, IEEE Internet of Things Journal 4 (2017) 1802–1831.
[35] ISO 26262, Road vehicles - Functional safety, ISO, 2011.
[36] J. Jürjens, UMLsec: Extending UML for secure systems development, in: Proceedings of the 5th International Conference on The Unified Modeling Language, UML '02, Springer-Verlag, London, UK, 2002, pp. 412–425.
[37] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, A. Naik, Replacing testing with formal verification in intel® CoreTM i7 processor execution engine validation, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 5643 LNCS (2009) 414–429.
[38] J.C. Knight, E.A. Strunk, Achieving critical system survivability through software architectures, in: R. de Lemos, C. Gacek, A. Romanovsky (Eds.), Architecting Dependable Systems II, Springer, Berlin, Heidelberg, 2004, pp. 51–78.
[39] K. Koh, P. Seong, SMV model-based safety analysis of software requirements, Reliability Engineering and System Safety 94 (2009) 320–331.
[40] Kushnet, D., The real story of Stuxnet, IEEE Spectrum (2013).
[41] M. Kwiatkowska, G. Norman, D. Parker, PRISM 4.0: Verification of Probabilistic Real-time Systems, in: G. Gopalakrishnan, S. Qadeer (Eds.), Proc. 23rd International Conference on Computer Aided Verification (CAV'11), volume 6806 of *LNCS*, Springer, 2011, pp. 585–591.
[42] F. Lagarde, H. Espinoza, F. Terrier, S. Gérard, Improving UML profile design practices by leveraging conceptual domain models, in: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), Atlanta (USA), ACM, 2007, pp. 445–448.
[43] N. Leveson, N. Dulac, Safety and risk-driven design in complex systems-of-systems, in: A Collection of Technical Papers - 1st Space Exploration Conference: Continuing the Voyage of Discovery, 2005, volume 1, pp. 584–608.
[44] T. Lodderstedt, D. Basin, J. Doser, SecureUML: A UML-Based Modeling Language for Model-Driven Security, in: l.J. Jézéque, H. Hussmann, S. Cook (Eds.), The Unified Modeling Language. UML 2002, volume 2460 of *Lecture Notes in Computer Science*, Springer, Berlin, Heidelberg, 2002.
[45] I. Lopatkin, A. Iliasov, A. Romanovsky, Y. Prokhorova, E. Troubitsyna, Patterns for representing FMEA in formal specification of control systems, Proceedings of IEEE International Symposium on High Assurance Systems Engineering, 2011, pp. 146–151.
[46] M.S. Lund, B. Solhaug, K. Stølen, Foundations of security analysis and design VI, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 231–274.
[47] S. Marrone, R. Rodríguez, R. Nardone, F. Flammini, V. Vittorini, On synergies of cyber and physical security modelling in vulnerability assessment of railway systems, Computers and Electrical Engineering 47 (2015) 275–285.
[48] A. Masrur, M. Kit, V. Matěna, T. Bures, W. Hardt, Component-based design of cyber-physical applications with safety-critical requirements, Microprocessors and Microsystems 42 (2016).
[49] D. Méry, N. Singh, Analyzing requirements using environment modelling, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 9185 (2015) 345–357.
[50] H. Mouratidis, Secure Software Systems Engineering: The Secure Tropos Approach (Invited Paper), JSW 6 (2011).
[51] OMG-MARTE, UML Profile for MARTE: Modeling and Analysis of Real-time Embedded Systems, OMG, 2011. Version 1.1, formal/11-06-02.
[52] Oracle, Website, 2019. URL: `https://www.oracle.com/technetwork/java/index.html`.
[53] M. Pavlidis, S. Islam, H. Mouratidis, A CASE Tool to Support Automated Modelling and Analysis of Security Requirements, Based on Secure Tropos, in: S. Nurcan (Ed.), IS Olympics: Information Systems in a Diverse World, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 95–109.
[54] M. Raja Ramesh, C. Satyananda Reddy, A survey on security requirement elicitation methods: Classification, merits and demerits, International Journal of Applied Engineering Research 11 (2016) 64–70.
[55] R.J. Rodríguez, J. Merseguer, S. Bernardi, Modelling security of critical infrastructures: A survivability assessment, The Computer Journal (2014).
[56] Y. Roudier, M. Idrees, L. Apvrille, Towards the model-driven engineering of security requirements for embedded systems, 2013 3rd International Workshop on Model-Driven Requirements Engineering, MoDRE 2013 - Proceedings, pp. 55–64.
[57] S. Scholz, K. Thramboulidis, Integration of model-based engineering with system safety analysis, International Journal of Industrial and Systems Engineering 15 (2013) 193–215.
[58] B. Selic, The pragmatics of model-driven development, IEEE Software 20 (2003) 19–25.
[59] B. Selic, A Systematic Approach to Domain-Specific Language Design Using UML, in: Tenth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2007), 7-9 May 2007, Santorini Island, Greece, IEEE Computer Society, 2007, pp. 2–9.
[60] O. Sheyner, J. Haines, S. Jha, R. Lippmann, J. Wing, Automated generation and analysis of attack graphs, Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy (2002) 273–284.
[61] J. Song, H. Zhao, X. Li, Y. Yang, C. Liu, H. Li, A new software failure analysis method based on the system reliability modeling, Proceedings of 2019 IEEE 8th Joint International Information Technology and Artificial Intelligence Conference, ITAIC 2019, pp. 1143–1149.
[62] I. Stellios, P. Kotzanikolaou, M. Psarakis, C. Alcaraz, J. Lopez, A survey of iot-enabled cyberattacks: Assessing attack paths to critical infrastructures and services, IEEE Communications Surveys & Tutorials 20 (2018) 3453–3495.
[63] The Eclipse Foundation, Website, 2019. URL: `http://www.eclipse.org/oxygen/`.

[64] The Object Management Group (OMG), Model-Driven Architecture Specification and Standardisation, Technical Report, 2018. URL: http://www.omg.org/mda/.
[65] E. Troubitsyna, Elicitation and specification of safety requirements, 3rd International Conference on Systems, ICONS 2008, pp. 202–207.
[66] S. Ullah, M. Iqbal, A.M. Khan, A survey on issues in non-functional requirements elicitation, in: International Conference on Computer Networks and Information Technology, 2011, pp. 333–340.
[67] UML2, Unified Modeling Language: Infrastructure, 2017. Version 2.5.1, OMG document: formal/2017-12-05.
[68] J. Vilela, J. Castro, L. Martins, T. Gorschek, Integration between requirements engineering and safety analysis: A systematic literature review, Journal of Systems and Software 125 (2017) 68–92.
[69] V. Vittorini, S. Marrone, N. Mazzocca, R. Nardone, A. Drago, A model-driven process for physical protection system design and vulnerability evaluation, Topics in Safety, Risk, Reliability and Quality 27 (2015) 143–169.
[70] H. Wang, D. Zhong, T. Zhao, Avionics system failure analysis and verification based on model checking, Engineering Failure Analysis 105 (2019) 373–385.
[71] J. Yoo, T. Kim, S. Cha, J.S. Lee, H. Son, A formal software requirements specification method for digital nuclear plant protection systems, Journal of Systems and Software 74 (2005) 73–83.
[72] Y. Zacchia Lun, A. D'Innocenzo, F. Smarra, I. Malavolta, M. Benedetto, State of the art of cyber-physical systems security: an automatic control perspective, Journal of Systems and Software 149 (2018).