SoREn, How Dynamic Software Update Tools Can Help Cybersecurity Systems to Improve Monitoring and Actions

Sébastien Martinez, Christophe Gransart, Olivier Stienne, Virginie Deniau, Philippe Bon

SoREn, How Dynamic Software Update Tools can help Cybersecurity Systems to improve Monitoring and Actions

Sébastien Martinez, Christophe Gransart, Olivier Stienne, Virginie Deniau and Philippe Bon
(Université Gustave Eiffel, IFSTTAR, France
name.surname@univ-eiffel.fr)

Abstract: Because stopping a service to apply updates raises issues, Dynamic Software Updating studies the application of updates on programs without disrupting the services they provide. This is acheived using specific mechanisms operating updating tasks such as the modification of the program state. To acheive transparency, Dynamic Software Updating systems use pre-selected and pre-configured mechanisms. Developers provide patches that are transparently converted to dynamic updates. The cost of such transparency is often that applied patches cannot modify the general semantic of the updated program. Allowing dynamic modification of the general semantic of a running program is rarely considered.

In the context of protection of communications between moving vehicles and uncontrolled infrastructure, SoREn (Security REconfigurable Engine) is designed to be dynamically reconfigurable. Its semantics can transparently be modified at runtime to change the security policy it enforces. Administrators can supply new policies to trigger a reconfiguration, without developing new components. This paper details and discusses the design of SoREn, its meta-model linked to cybersecurity business concepts and its automatic reconfiguration calculator allowing transparent application of reconfigurations.

Key Words: dynamic software updating, dynamic reconfiguration, quiescence, security
Category: D.2.7, D.2.10

1 Introduction

Today's world expects software systems to be available at every moment, whether the system provides critical services like airport traffic control management or whether its downtime would cause user discomfort like a computer operating system forcing a reboot for updating. Updating running software systems becomes a critical issue as it requires the system to be restarted, causing downtime and loss of state as well as financiary losses. As a consequence, fixing errors and patching vulnerabilities is often postponed, leaving software unprotected. Dealing with service disruption induced by regular updating processes requires cautious planning of maintenance operations. They are often planned during known periods of low usage, imply the duplication of services and rotation of the services to be patched and rebooted. Dynamic Software Updating (DSU) addresses this issue and focuses on methods for applying updates without disrupting provided services. DSU systems embed mechanisms for updating running applications (*e.g.* transform their state, replace obsolete code with its newest version) [Giuffrida and Tanenbaum, 2010]. Many of these mechanisms are detailed in the literature, each having specific needs and properties. A given mechanism may be well suited for a specific type of application or update while being incompatible with other types of application or update. Each DSU system embeds a few mechanisms well suited to its targets (*i.e.* programming language, type of application, type of update) and use them for every update. This approach allows transparent usage of dynamic updating as little developer interaction is required to apply dynamic patches. However it usually restricts possibilities for updates, often to the point that updates can only fix errors and

security flaws without modifying the general semantic of the original program. When discussing dynamic updates, transparency usually means that a patch developed through usual software maintenance processes is applied to a running program without requiring specific adaptation of the patch. When using dynamic updates mainly to fix errors and security flaws, this definition of transparency is pertinent since fixing a security flaw requires developer intervention. When using dynamic updating to make the semantic of a program evolve, not requiring the systematic intervention of developers and allowing administrators with a more general knowledge of the program to specify its modifications using only business concepts would simplify the evolution of the program. The amount of work necessary to dynamically update the program would be significantly reduced and the adaptability of the program would increase significantly.

In the context of the SECOURT [LAMIH, 2020] project addressing the security of communications between vehicles and ground stations, this paper presents SoREn (Security REconfigurable Engine), a meta-model based platform which semantic can be dynamically modified through high level specification, with little developer intervention. Vehicles move in an uncontrolled environment communicating using wireless uncontrolled infrastructure. Their communications need to be protected from attacks and to adapt to their changing environment (*e.g.* use the available communication medium). SoREn is designed for such embedded systems. It detects attacks or changing conditions and reacts according to policies specified by administrators. To respond to the needs of the vehicle, the policies enforced by the platform should be dynamically modifiable, allowing change of policy while vehicles are moving, and their specification should be possible by administrators without requiring the intervention of developers. Note that because vehicles are not expected to be in perpetual communication with a ground control station, the decision module of the platform cannot be centralized in a ground station, regulating the decisions of each vehicle. The decision module needs to be embedded in the vehicle itself. As a result, several vehicles using the same security policy could behave differently according to their environment. SoREn is designed to be flexible and transparently modifiable. It should be possible to implement enforced policies with as little restrictions as possible and it should be possible to reconfigure the platform so that it enforces these new policies with as little developer intervention as possible. Developers should only be required for the development of new software components.

This papers presents SoREn, focusing on two contributions of the platform : a meta-model based design providing flexibility for the implementation of new policies and an automatic reconfiguration calculation system providing transparency for dynamic updates. Both contributions are interlinked. The meta-model defines general constraints on the possible configurations, allowing the selection of DSU mechanisms suited for every update and simplifying the calculation of reconfigurations. Section 2 presents related work in the field of security systems and dynamic updating platforms. Section 3 presents the meta-model of SoREn. It details the links between the meta-model and cyber-security business concepts and how it facilitates high-level specification of configurations. Section 4 discusses the issues of dynamic updating and how they are addressed in SoREn. It discusses the DSU mechanisms used in the platform. Section 5 presents an operational semantic designed for the specification of the automatic reconfiguration calculator of SoREn. Section 6 discusses the implementation of SoREn and presents an experiment, validating the pertinence of the design of SoREn. Section 7 discusses the contributions of the paper and section 8 concludes the paper.

2 Related works

SoREn is a dynamically reconfigurable platform detecting attacks and triggering responses to them. It combines several features of Security Information and Event Management (SIEM) systems, Intrusion Detection Systems (IDS) and DSU platforms.

In a general cybersecurity framework like [Malatji et al., 2019], two parts are present: the social dimension and the technical dimension. In this context, SoREn is the technical element. However, we introduce also the social dimension in our work with business concepts into a meta-model. So SoREn can be integrated into a general cybersecurity framework. [Buccafurri et al., 2015] proposes a processing approach to achieve cybersecurity compliance assessment. This approach is one step before the content of this paper where we discuss the technical achievement of our system.

Initially, SIEM systems were designed to collect event logs from a variety of sources and perform real-time aggregation and correlation to detect attacks or confirm alerts from sensors such as IDS[Sourcefire, 2020, OISF, 2020, Zeek, 2020]. SIEM alerts are technical sets of information, including targeted or compromised systems, as well as attacking sources. Alerts define a level that may be either the technical impact or the priority or both. Then information shared by SIEM systems mostly targets cyber security experts.

A host-based intrusion detection system (HIDS) is a software application that monitors a single host and the events occurring within that host for malicious activities. It analyses data such as log files, system calls, file accesses, user or application behaviour on the host on which it is operating, and generates alerts once an intrusion has been detected. Being restricted to one host provides a reliable and precise analysis to determine what processes and users are involved in a particular intrusion.

A network-based intrusion detection system (NIDS) is a standalone hardware device that monitors network traffic for particular network segments or devices to identify malicious activities such as denial of service attacks, port-scans or even attempts to crack into computers. It consists of a set of single-purpose sensors placed at various points in a network to inspect the data packets from all devices that reside inside its local area network, management server, console, and optionally database server. There are two primary data sources for a network-based intrusion detection: network packets and flows.

SoREn provides interfaces for sensors and detection, following the general principle of an IDS, while embedding information management of threats and watched system status, as SIEM systems. To allow dynamic reconfiguration, SoREn uses dynamic software updating mechanisms as other DSU platforms.

Dynamic Correspondence Proxyfication [Gregersen and Jørgensen, 2009] (DCP) is a framework using in place proxies to handle relinking of Java classes when updating components. When a component is updated, the classes it shares with other components are replaced by proxies to their newer version. DCP lazily migrates the state of components : the state of a component is migrated at the moment it is used.

Afpac [Buisson et al., 2006, Aldinucci et al., 2005] is a framework for distributed components. They can dynamically change their algorithms when ordered by their internal decider enforcing a specific policy and collecting data on the execution and its environment through monitors. Afpac embeds a distributed protocol for determining a best suited update point in the execution where all threads of the component can be updated without causing inconsistency.

In [Cavalcante et al., 2015], Cavalcante et al. provide support for dynamic updates in the π-ADL language. π-ADL is an Architecture Description Language (ADL) allowing software architects to define programmed reconfigurations, *i.e.* reconfigurations forseen at design time. Reconfigurations are handled by decomposing the system, installing, uninstalling or swapping the components and composing the system according to its new configuration. Cavalcante et al. use π-ADL to define the architecture of a flood monitoring system and give a method for transforming π-ADL architectures into Go [Meyerson, 2014] source code.

The K42 operating system [Appavoo et al., 2003, Baumann et al., 2005] uses dynamic reconfiguration for updating its components and adapting its configuration to contextual needs. For example, a component implementing an algorithm best suited for sequential accesses to a resource as a file can be replaced by another component best suited for parallel accesses when more than one application need to access the resource. Appavoo et al. also describe how dynamic reconfiguration can be used for monitoring security threats with little cost. The operating system could embed broad-based monitoring components that would be replaced by more precise components whenever an anomaly is detected.

Components of K42 handle transactions invoked by short-lived threads. Whenever a reconfiguration is requested, monitor components are attached to each replaced component and prevents new threads to invoke new transactions. When all ongoing transactions are completed, the component is replaced and the monitor allows new transactions.

Several platforms providing DSU capabilities for component based software can be found in the literature. However, software reconfiguration specification using high-level concepts is one originality of SoREn.

3 A meta-model suited for high-level specification of policies

To ensure simple dynamic updates ordered by high-level instructions, the updating process must be transparent and the running platform, its code and its state must be expressible through high-level concepts. Users wanting to dynamically modify the platform should be able to express new security policies without needing to consider DSU issues and only using concepts as close as possible to their business. That means that security policies should be expressed using, for instance, the concepts of *threat* and *detection* rather than concepts specific to the implementation of the platform. In the rest of this paper, underlined words refer to specific concepts of the presented work. Words written in italics refer to concepts from the domains of security or dynamic software updating. In both cases, the concepts are defined as they appear in the paper and are only emphasised when clarification is needed in some sections.

To handle dynamic updates transparently, mechanisms should be pre-selected and installed in the platform. They will be used for each update and as a consequence should be well suited for any future update. But future updates cannot be predicted and no DSU mechanism is well suited for every possible update. The platform and its updates should be subjected to constraints ensuring that for any update respecting these constraints, the pre-selected mechanisms are well suited. In SoREn, these constraints take the form of a meta-model specifying how every version of the platform should be modeled. According to that meta-model, the platform architecture is component oriented and key components are stateless. Component oriented architecture simplifies the management of updates as any update can be designed as the installation, the removal or the swapping of components. This architectural choice ensures a well defined update unit (*i.e.* the

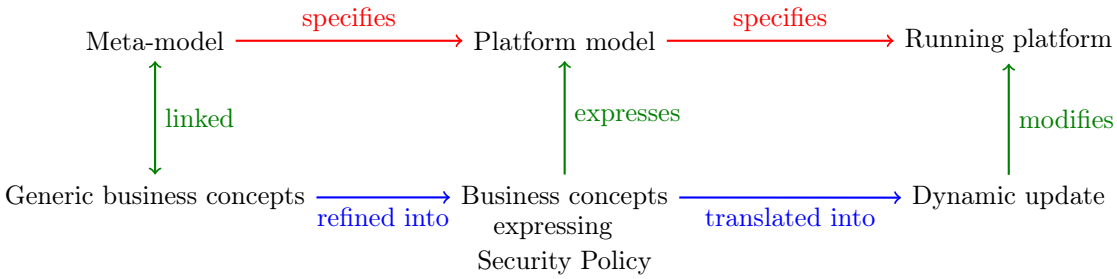

Figure 1: Relations between meta-model, platform model and running platform

Meta-types	Generic Business concepts	Example
Probe	Raw data	Measured Signal/Noise Ratio by a Wi-Fi card
Measure	Information	the source of an authentication attempt
Detector	High level information processing	an intrusion detection system
Situation	Threat	a deauthentication attack
Decider	Decision or enforced rule	a rule to switch between networks
Response	Reaction	a IP address block request
Actuator	Modify the behavior of the system	a firewall
Part	Asset status	the current protocol being used

Marker meta-type
Component meta-type

Table 1: Business concepts and meta-types in SoREn

most basic modifiable element), easy to manipulate at high level. Stateless components require less mechanisms when removed or swapped as their state does not need to be managed. Both constraints indicate the mechanisms to be used for modifying a running platform : they should be component oriented and do not need to consider state management.

Specifying the component oriented architecture with a meta-model enables the specification of a running platform using business concepts. Meta-types defined in SoREn are linked to business concepts. Defining a component type at model level is similar to defining specific cases from the business concepts. Specifying a configuration of the platform is similar to defining the security policy it should enforce. Figure 1 presents the relations between the meta-model and the business concepts. The meta-model of SoREn is linked to generic business concepts that are refined into business concepts, used to express the security policy. Expressing a security policy using business concepts is equivalent to specifying a platform model using the meta-model. The expressed security policy can then be translated to a dynamic update that will make the running platform comply to the model it expresses.

We distinguish two kinds of generic business concepts : action concepts, concerning actions taken by the running platform and information concepts, concerning information accessible and

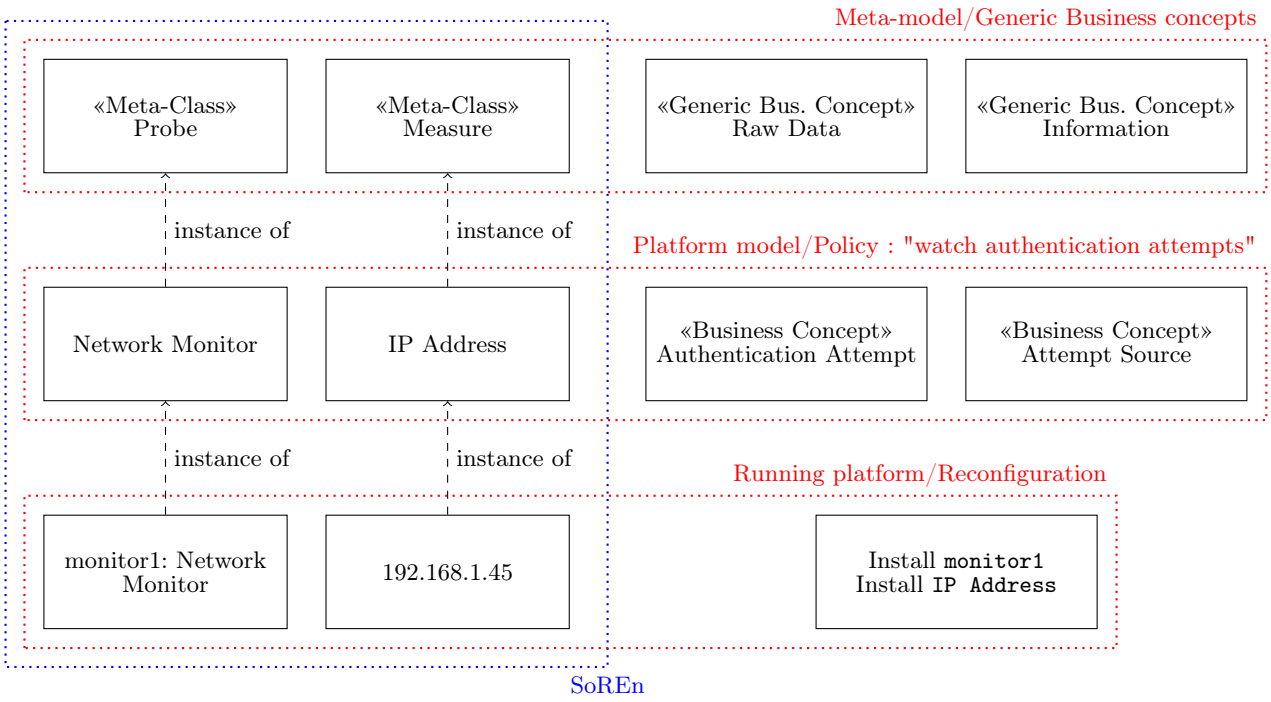

Figure 2: Relation between platform model and business concepts

used by the platform. The meta-model of SoREn specifies component meta-types linked to the action concepts and marker meta-types linked to the information concepts. Markers are objects present in the platform encoding information on the platform global state, the systems it watches and their environment. Table 1 lists the meta-types defined in the SoREn meta-model and their associated generic business concepts. It gives a real-life example for each generic business concept. Figure 2 gives an example of instantiation of the meta-model and its associated generic business concepts into a reconfiguration. Its only depicts the instantiation process for the Probe and Measure meta-classes as the instantiation of other meta-classes would work in a similar way. The Probe and Measure meta-classes are linked to the Raw Data and Information generic business concepts. The Authentication Attempt and Attempt Source business concepts are instantiation of the generic business concepts and are combined to express a policy for watching authentication attempts. In SoREn, these business concepts take the form of Probe and Measure types Network Monitor and Ip Address. Assuming the platform was enforcing no previous policy, translating the policy into a reconfiguration boils down to listing components and marker types to be installed.

The meta-model of SoREn defines meta-types and how they interact with each other. In a platform, components are connected to a core module through which they communicate with by the mean of markers. Figure 3 details the interaction between components and markers. Components of meta-type Probe (called "probes" in the following) get information from watched systems and their environment. They produce measures (markers of meta-type Measure) registering that

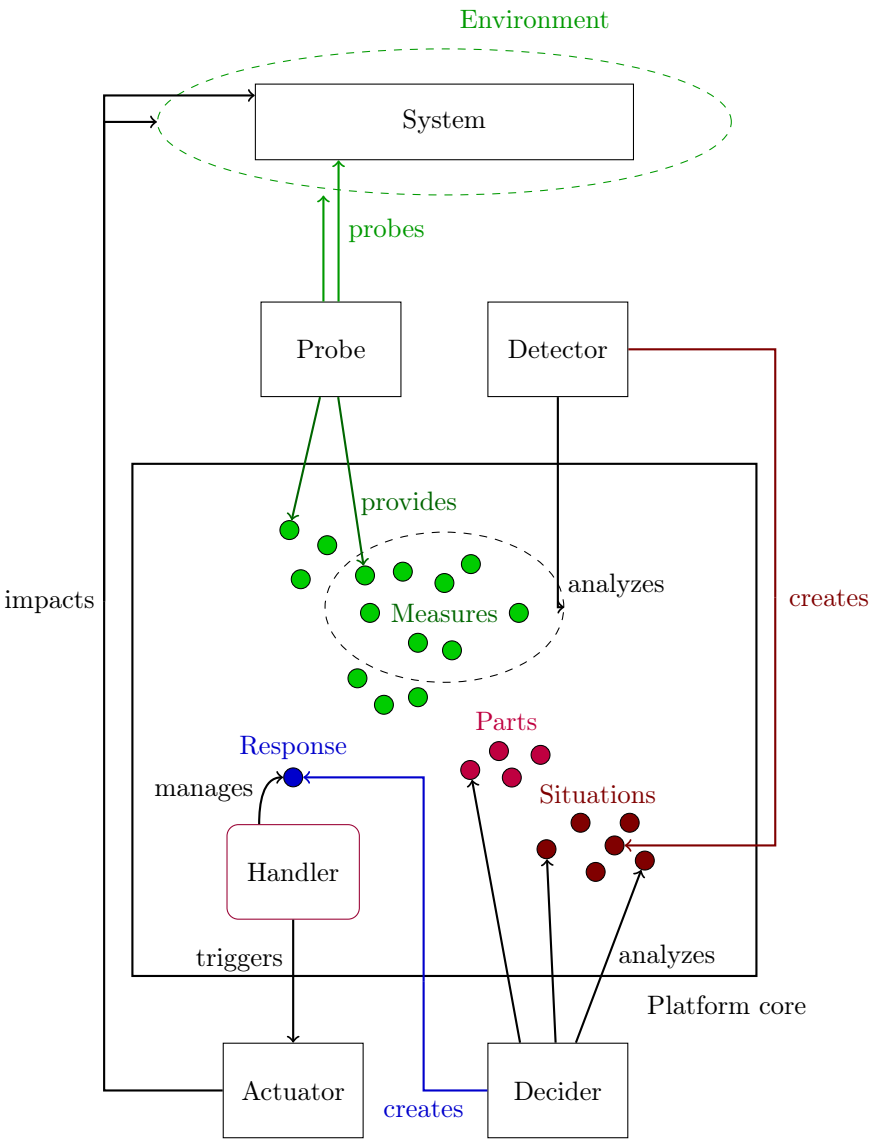

Figure 3: Interactions between markers and components according to the meta-model of SoREn

information. Detectors (components of meta-type Detector) analyze measures present in the platform core and process them according to their inner semantic (*e.g.* recognize patterns, check thresholds). If a detector detects a threat, it generates a situation (marker of meta-type Situation) registering all information about that threat (*e.g.* date of detection, concerned assets). Deciders (components of meta-type Decider) analyze situations and parts. Parts are markers of meta-type Part representing the state of a given asset. While other markers only register information and are deleted from the platform when they become irrelevant, parts persist in the platform core and are synchronized to the assets they represent. Their state changes whenever the state of their

associated asset changes. Using information collected by analyzing parts and situations, deciders may trigger the creation of responses (markers of meta-type Response) according to their inner semantic. Responses register requests for reactions and contain all information relevant to these reactions. Creating a response in the platform core triggers all installed actuators (components of meta-type Actuator). If one installed actuator can handle the requested reaction in the conditions detailed in the response, that actuator follows its inner semantic to act on the watched systems and/or their environment. If the action of the actuator impacts any asset associated to a part in the platform core, the state of that part is modified to represent the new state of that asset. Measures are never deleted from the platform. A measure indicates information received at a given time and date. Situations and responses are deleted when obsolete. They indicate a current threat or a requested action and delete when the threat is passed or when the requested action is complete. Responses are managed by a handler responsible for the triggering of the appropriate actuators. When a response is created, the handler looks up an installed actuator that can handle the response and triggers it. Parts are not produced by components. They are installed once and their state evolves in synchronization with their associated assets until they are uninstalled. The distinction between markers and components in the meta-model ensure a stateless design for components and distinguishes the purpose of both components and markers. Markers indicate what the platform "knows" on the watched systems and their environment. An external observer analyzing the markers present in the core platform would get an overview of the watched systems, their state, the threats they are subject to and what action is being taken as a response. Components indicate what the platform "can do". An external observer analyzing the installed components would get an overview of the security policy enforced by the platform.

The meta-model detailed in this section treats components as black boxes of which only inputs and outputs are known. Components are weakly constrained, allowing more possibilities for their inner semantics and ensuring flexibility of the platform.

4 Dynamic updating issues

This section discusses the selection and the configuration of DSU mechanisms embedded in SoREn. It presents the main issues of dynamic updates and how they are addressed in SoREn.

4.1 Update unit and update management

As presented in section 3, the architecture of SoREn is component oriented to simplify dynamic updating. Any update of a running platform can be modeled as operations on components. The smallest part of SoREn that can be modified through dynamic updating is either a component or a marker type. Because there is a strong dependency of components on the marker types they handle, updating a marker type requires updating all components handling it. By definition of an update unit (*i.e.* the smallest part of a program that can be updated), the update unit of SoREn is a component or a marker type. However, since every update includes modifications on at least one component, the update unit can be considered as being one component for simplification purposes.

Dynamic updates on the running platform are managed by a manager module embedded in the platform core. This module embeds a thread executing update operations concurrently with the general execution of the platform. Updates are applied sequentially, preventing the

simultaneous application of two updates. Parallel application of two updates modifying a same element in an incompatible way (*e.g.* one update deletes a field used by the other) raises the issue of choosing which modification should be applied while the other is discarded. That issue is not addressed in SoREn and is out of the scope of this paper.

4.2 Alterability and update life-cycle

Updates follow a straightforward life-cycle centered on the detection of alterability. A given program is in a *state of alterability* [Martinez et al., 2015] for a given update if the update can be applied to program without causing unwanted behavior (*e.g.* crash of the program, manipulation of inconsistent data). The state of alterability of a program can be detected by watching alterability criteria which are conditions on the state of a program. Several alterability criteria are defined in the literature [Vandewoude et al., 2007, Ghafari et al., 2012]. Selecting suited alterability criteria for a given program and a given update is a central issue of dynamic updating. In SoREn, the alterability of running platforms is detected using the *quiescence* of modified components, as defined by Kramer and Magee [Kramer and Magee, 1990]. In a component oriented architecture, a component is quiescent if :

(Condition 1) it is not engaged in a transaction

(Condition 2) it will not initiate new transactions

(Condition 3) no transaction engaging the component will be initiated by other components

A quiescent component can be safely removed or modified from the system. The system is therefore in a state of alterability for any update concerning that component alone.

In SoREn, probes, detectors and deciders periodically engage transactions by themselves. As a consequence, condition (3) is always satisfied for components of these meta-types. The quiescence of these components can be achieved by suspending them after they complete a transaction (satisfying conditions (1) and (2)). Transactions engaging actuators are initiated by the response handler embedded in the platform core, when response markers of the appropriate type are created. As a consequence, condition (2) is always satisfied for actuators. Condition (3) can be satisfied by suspending the creation of response markers handled by the actuator. The quiescence of actuators can be achieved by suspending the creation of associated responses and waiting for any engaged transaction to finish (satisfying condition (1)). New components to be installed in the platform are always quiescent because they are engaged in no transaction and no transaction will be initiated until the components are installed.

When an update is requested, the update manager guides the execution of the platform towards its alterability state by suspending concerned probes, detectors and deciders after they complete their current transaction and by suspending the creation of response markers handled by concerned actuators. When every component concerned by the update is quiescent, the platform is alterable and the update is applied.

Life-cycle of updates in SoREn considers others criteria than alterability criteria. In some cases, even if the platform is alterable, we may not want to apply an update. When a situation marker is present in the platform, we need all components and marker types necessary for handling the situation. Probes are needed for providing up-to-date information. Detectors are needed

to detect possible changes in the situation. Deciders are needed to analyze that situation and actuators are needed to respond to that situation. Before guiding the platform to alterability, the update managers checks that the update does not remove components needed by the platform. If the platform needs components removed by the update, the update is postponed to a later moment. This step of the update life-cycle is detailed in section 5

4.3 Data updating

Because components are stateless, the only data concerned by updates in SoREn are the markers present in the platform. Because of the strong link between components and marker types, component updates are likely to affect markers. A newly installed component may produce markers of a new type that should be installed. The removal of a component may cause a type of marker to be unecessary. In SoREn, new marker types are automatically installed along components that require them, and marker types required by no components are uninstalled when no marker of their type is present in the platform. Even if an update modifies a marker type, SoREn treats that update as the installation of a new marker type. Modifying an installed marker type implies modifying all markers of that type which raises issues if that modification extends the information held by the markers. In that case, the update developper needs to specify how to treat that information extension when converting oudated marker to their new type (*e.g.* add void fields, use default values). Because we want to limit intervention by developper during updates, we chose to consider the modification of a marker type as the addition of a new marker type. This choice has little impact on updates. Because markers have a short life span in the platform, oudated markers are rapidly replaced by markers of the new type. Situations and responses are eventualy deleted and measures become old enough that no detectors analyze them. Because each part type has only a few instances in the platform, parts and part types are treated as components during installation. Part types are installed and instanciated or uninstalled when explicitely requested.

5 Automatic reconfiguration

To change the configuration of a running platform, administrators supply a configuration specifying several scenarios. That configuration is a combination of scenarios detailing policies enforced by the platform. The configuration is parsed and translated into update instructions submitted to the plaform update manager. This section details how scenarios are specified and presents an operational semantic for calculating an update from a supplied configuration.

5.1 Defining a scenario

A scenario lists components that should be installed and specifies the configuration of each components. It also details links between installed components : for each pair of components that exchange information, the scenario lists the marker types they use to communicate. A scenario is the equivalent of one security policy. One configuration regroups one or more scenarios, the partition of the configuration into scenarios is left to administrators discretion provided each scenario is independent from the other. Each scenario must contain the dependencies of the components it lists. Figure 4 presents an example of scenario handling deauthentication attacks

```
Probes :
- DeauthProbe : /path/to/probe
Detectors :
- DeauthAttackDetector : /path/to/detector, detectFloor = 5, resetCeil = 3
Deciders :
- WiFiSwitcher : /path/to/decider
Actuators :
- WiFiController : /path/to/actuator
Parts :
- TransmissionProtocol : /path/to/part_type
Links :
- DeauthProbe to DeauthAttackDetector with DeauthenticationFrame
- DeauthAttackDetector to  WiFiSwitcher with DeauthAttack
- WiFiSwitcher to  WiFiController with DisableWiFi, EnableWiFi
```

Figure 4: Example of scenario

of WiFi communications. Note that parts are treated as components, as indicated in section 4. Only the names of the marker types are specified in scenarios. Full specification of marker types (*e.g.* fields, default values) are embedded in component types manifests. Each component type is associated with a manifest specifying dependencies, handled marker types and instructions for instantiating and installing components.

5.2 Calculating reconfiguration operations

A configuration can be formally defined as a set of Scenarios which in turn can be formally defined as a set of components and links.

Definition 1 Links and scenarios. A Link specifies exchanges of information between two components types τ_C and τ'_C. It lists all markers types $\mathcal{MT}_i$ through which the components of the two types exchange information.

$$\mathcal{L} = \{\tau_C, \tau'_C, \{\mathcal{MT}_0, ..., \mathcal{MT}_k\}\}$$

A scenario is a set of components (instances of component types), combined with a set of links. The components in S are instances of the components types listed in the links of S.

$$S = \{\{C_0, ..., C_k\}, \{\mathcal{L}_0, ..., \mathcal{L}_j\}\}$$

Assuming $\{S_0, ..., S_k\}$ is the configuration of the running platform, when a new configuration $\{S'_0, ..., S'_j\}$ is supplied by administrators, the reconfiguration process assumes that all components and marker types listed in the old configuration should be uninstalled and all components and marker types listed in the new configuration should be installed. If the configurations have no common element, this is a fair assumption. If they share similarities, the reconfiguration process should takes these similarities into account to avoid uninstalling then installing a same element under another name or with different parameters. To address that issue, we define an operational semantic for reducing the initial term $\mathbf{Remove}\, S_0\, t_0, ..., \mathbf{Remove}\, S_k\, t_k\ ,\ \mathbf{Add}\, S'_0\, t'_0, ..., \mathbf{Add}\, S'_j\, t'_j$. Figure 5 details the grammar of that semantic.

Before simplification, the term is divided into blocks. Subterms that have common elements are grouped into blocks according to the following rule : if t_c and t'_c contain scenario update terms

Configuration update term

$$t_c ::= \textbf{Add}\, S\, t \mid \textbf{Remove}\, S\, t \mid t_c\,,\, t'_c$$

Scenario update term

$$t ::= \textbf{add}\, e \mid \textbf{remove}\, e \mid \textbf{rename}\, C\, n \mid \textbf{modify}\, C\, n_0\, v_0 \ldots n_k\, v_k \mid \textbf{link}\, e\, S \mid \textbf{relink}\, e\, S\, S' \mid t\, t' \mid \bot$$

Context (scenario)	**Context (configuration)**
$\mathrm{C} ::= t\,[\cdot] \mid [\cdot]\,t$	$\mathrm{C_c} ::= t_c^0 \ldots t_c^k\,[\cdot]\,{t'_c}^0 \ldots {t'_c}^j$

e : an element (marker type, component)
n : a name (element name or element parameter name)
v : a value (element parameter value)
C : a component

Figure 5: Grammar of the update semantic

Alterability criteria

$$\mathcal{A} ::= \textbf{True} \mid \textbf{no}\, \{s_0, ..., s_k\} \mid \mathcal{A}\,\&\,\mathcal{A}' \mid \textbf{quiescence}\, \mathcal{C}$$

s : a situation type

$\mathcal{C}$: a component

Figure 6: Grammar for alterability criteria

concerning (respectively) elements e and e' (*e.g.* $t_c = \textbf{Add}\, S\, \textbf{add}\, e\,,\, t'_c = \textbf{Remove}\, S'\, \textbf{remove}\, e'$), t_c and t'_c are grouped in a block if any of the following requirements are verified :

1. e and e' are identical : they have the same type, name and parameters (noted $e = e'$)
2. e and e' are different only by name (noted $\textbf{diff}(e, e') = n$)
3. e and e' are different only by parameters (noted $\textbf{diff}(e, e') = n_1 v_1, ..., n_k v_k$)
4. e and e' are different only by name and parameters (noted $\textbf{diff}(e, e') = n, n_1 v_1, ..., n_k v_k$)

Each block is simplified using the reduction rules of figure 7. If an element e' is uninstalled through the uninstallation of a scenario and replaced by an identical element e from a new scenario, that element is kept installed in the platform and relinked to the new scenario (Rule 1). If e' only differs from e by name or parameters, e' is relinked to the new scenario and modified to become identical to e (Rules 2 through 4). The reduction rules also handle the duplication of elements. If two scenario S and S' install two identical elements, respectively e and e', only e is installed and linked to S', preventing the duplication of elements in the platform (Rule 5). If the installation of a scenario S relinks an element e to it while an identical element e' is installed by a different scenario S', e is linked to S' and e' is not installed (Rule 6).

After simplification, each blocks is transformed into one update. As a consequence, one reconfiguration requested by the administrator can be decomposed into several independent updates. This allows a faster application of these updates and a faster fulfillment of the reconfiguration requested. Indeed, smaller update have less restrictive alterability criteria which are easier to

$$\boxed{t_c \longrightarrow t'_c}$$

$$\frac{e = e'}{\mathrm{C_c}[\mathbf{Add}\ S\ \mathrm{C}[\mathbf{add}\ e]\ ,\ \mathbf{Remove}\ S'\ \mathrm{C}'[\mathbf{remove}\ e']] \longrightarrow \mathrm{C_c}[\mathbf{Add}\ S\ \mathrm{C}[\mathbf{relink}\ e'\ S'\ S]\ ,\ \mathbf{Remove}\ S'\ \mathrm{C}'[\ \bot\ ]]}\ 1$$

$$\frac{\mathbf{diff}\ (C\ ,\ C') = n_0\ v_0\ ,\ \ldots\ ,\ n_k\ v_k}{\mathrm{C_c}[\mathbf{Add}\ S\ \mathrm{C}[\mathbf{add}\ C]\ ,\ \mathbf{Remove}\ S'\ \mathrm{C}'[\mathbf{remove}\ C']] \longrightarrow \mathrm{C_c}[\mathbf{Add}\ S\ \mathrm{C}[\mathbf{relink}\ C'\ S'\ S\ \mathbf{modify}\ C'\ n_0\ v_0 \ldots n_k\ v_k]\ ,\ \mathbf{Remove}\ S'\ \mathrm{C}'[\ \bot\ ]]}\ 2$$

$$\frac{\mathbf{diff}\ (C\ ,\ C') = n}{\mathrm{C_c}[\mathbf{Add}\ S\ \mathrm{C}[\mathbf{add}\ e]\ ,\ \mathbf{Remove}\ S'\ \mathrm{C}'[\mathbf{remove}\ C']] \longrightarrow \mathrm{C_c}[\mathbf{Add}\ S\ \mathrm{C}[\mathbf{relink}\ C'\ S'\ S\ \mathbf{rename}\ C'\ n]\ ,\ \mathbf{Remove}\ S'\ \mathrm{C}'[\ \bot\ ]]}\ 3$$

$$\frac{\mathbf{diff}\ (C\ ,\ C') = n\ ,\ n_0\ v_0\ ,\ \ldots\ ,\ n_k\ v_k}{\mathrm{C_c}[\mathbf{Add}\ S\ \mathrm{C}[\mathbf{add}\ e]\ ,\ \mathbf{Remove}\ S'\ \mathrm{C}'[\mathbf{remove}\ C']] \longrightarrow \mathrm{C_c}[\mathbf{Add}\ S\ \mathrm{C}[\mathbf{relink}\ C'\ S'\ S\ \mathbf{rename}\ C'\ n\ \mathbf{modify}\ C'\ n_0\ v_0 .. n_k\ v_k]\ ,\ \mathbf{Remove}\ S'\ \mathrm{C}'[\ \bot\ ]]}\ 4$$

$$\frac{e = e'}{\mathrm{C_c}[\mathbf{Add}\ S\ \mathrm{C}[\mathbf{add}\ e]\ ,\ \mathbf{Add}\ S'\ \mathrm{C}'[\mathbf{add}\ e']] \longrightarrow \mathrm{C_c}[\mathbf{Add}\ S\ \mathrm{C}[\mathbf{add}\ e]\ ,\ \mathbf{Add}\ S'\ \mathrm{C}'[\mathbf{link}\ e\ S']]}\ 5$$

$$\frac{e = e'}{\mathrm{C_c}[\mathbf{Add}\ S\ \mathrm{C}[\mathbf{relink}\ e\ S''\ S]\ ,\ \mathbf{Add}\ S'\ \mathrm{C}'[\mathbf{add}\ e']] \longrightarrow \mathrm{C_c}[\mathbf{Add}\ S\ \mathrm{C}[\mathbf{relink}\ e\ S''\ S]\ ,\ \mathbf{Add}\ S'\ \mathrm{C}'[\mathbf{link}\ e\ S']]}\ 6$$

Figure 7: Reductions rules for the simplification of blocks

$$\boxed{t \vdash \mathcal{A}}$$

$$\frac{}{\mathbf{add}\ e \vdash \mathbf{True}}\ 1 \qquad \frac{}{\mathbf{link}\ e \vdash \mathbf{True}}\ 2 \qquad \frac{}{\mathbf{relink}\ e\ S\ S' \vdash \mathbf{True}}\ 3 \qquad \frac{}{\bot \vdash \mathbf{True}}\ 4 \qquad \frac{}{\mathbf{remove}\ C \vdash \mathbf{quiescence}\ C}\ 5$$

$$\frac{}{\mathbf{rename}\ C\ n \vdash \mathbf{quiescence}\ C}\ 5 \qquad \frac{}{\mathbf{modify}\ C\ n_0\ v_0 \ldots n_k\ v_k \vdash \mathbf{quiescence}\ C}\ 6 \qquad \frac{e\ \mathbf{is\ not\ a\ component}}{\mathbf{remove}\ e \vdash \mathbf{True}}\ 5$$

$$\frac{t \vdash \mathcal{A} \quad t' \vdash \mathcal{A}'}{t\ t' \vdash \mathcal{A}\,\&\,\mathcal{A}'}\ 6$$

$$\boxed{t_c \vdash \mathcal{A}}$$

$$\frac{t \vdash \mathcal{A}}{\mathbf{Add}\ S\ t \vdash \mathcal{A}}\ 7 \qquad \frac{t \vdash \mathcal{A} \quad \mathbf{situations}(S) = \{s_0 \ldots s_k\}}{\mathbf{Remove}\ S\ t \vdash \mathbf{no}\ \{s_0, \ldots, s_k\}\,\&\,\mathcal{A}}\ 8 \qquad \frac{t \vdash \mathcal{A} \quad t' \vdash \mathcal{A}' \quad S > S'}{\mathbf{Add}\ S\ t\ ,\ \mathbf{Remove}\ S'\ t' \vdash \mathcal{A}\,\&\,\mathcal{A}'}\ 9$$

$$\frac{t \vdash \mathcal{A} \quad t_0 \vdash \mathcal{A}_0 \quad \ldots \quad t_k \vdash \mathcal{A}_k \quad S > S'_0, \ldots, S > S'_k}{\mathbf{Add}\ S\ t\ ,\ \mathbf{Remove}\ S_0\ t_0\ ,\ \ldots\ ,\ \mathbf{Remove}\ S_k\ t_k \vdash \mathcal{A}\,\&\,\mathcal{A}_0\,\&\,\ldots\,\&\,\mathcal{A}_k}\ 10 \qquad \frac{t_c \vdash \mathcal{A} \quad t'_c \vdash \mathcal{A}' \quad t_c \not> t'_c}{t_c\ ,\ t'_c \vdash \mathcal{A}\,\&\,\mathcal{A}'}\ 11$$

$\mathbf{situations}(S)$: situations markers handeld by scenario S

Symmetric versions of rules 9 and 10 switching order of terms are not presented.
For simplicity purposes, rule 10 uses syntactic sugar notations.

Figure 8: Inference rules to calculate alterability criteria

satisfy. The update terms of each block specify the operations of the update. Adding alterability criteria to a block provides enough information to achieve a full specification of the update. Figure 6 presents the grammar for alterability criteria and figure 8 details inference rules for deducing alterability criteria from update terms. Note that alterability criteria include security criteria discussed in section 4. A scenario cannot be removed while situations it handles are present in the platform. Rules 1 through 6 detail how quiescence criteria are deduced from scenario update terms. The removal, the renaming and the modification of a component require that component to be quiescent. Other operations do not produce alterability criteria because the platform is always alterable for them. Rules 7 through 11 detail how security criteria are deduced from configuration update terms. Removing a scenario requires the absence of situations handled by it (Rule 8). Adding a scenario does not have such requirements (Rule 7). In some situations, one scenario is removed to be replaced by a new scenario extending it. The new scenario may be an update of the removed scenario, replacing components by updated versions or even add new components to handle more marker types. In this case, situations handled by the older version can be handled by both scenarios in a same maner and the presence of such situations should not prevent the application of the update.

We consider the extension of a scenario by another. A scenario S extends a different scenario S' if the *general semantic* of S' is included in the semantic S. *i.e.* S extends S' if through the modification of components parameters, S' is included in S. Such general semantic is encoded in the links of scenarios. Links specify the marker types handled by the scenario and how the components of the scenario interact with each other through markers. Two scenarios sharing the same links only differ by the configuration of their components. We define scenario extension using scenario links.

Definition 2 Link extension. A link extends another if it includes the exchange of information specified by the other link.

Let $\mathcal{L} = \{\tau_C^0, \tau_C^1, \{\mathcal{MT}_0, ..., \mathcal{MT}_k\}\}$ and $\mathcal{L}' = \{{\tau'}_C^0, {\tau'}_C^1, \{\mathcal{MT}'_0, ..., \mathcal{MT}'_n\}\}$ be two links.

$\mathcal{L}$ extends $\mathcal{L}'$ if $\{\mathcal{MT}'_0, ..., \mathcal{MT}'_n\} \subset \{\mathcal{MT}_0, ..., \mathcal{MT}_k\}$. We note $\mathcal{L} > \mathcal{L}'$.

Definition 3 Scenario extension. A scenario S extends another scenario S' if all the links of S' are extended by links of S. We note $S > S'$.

$$S > S' \Leftrightarrow \forall \mathcal{L}' \in S', \exists \mathcal{L} \in S \ s.t. \ \mathcal{L} > \mathcal{L}'$$

We note $S \not> S'$ if there is no extension relation between S and S' (neither S extends S' nor S' extends S). We extend that notation to configuration update terms and note $t_c \not> t'_c$ if there is no extension relation between any of the scenarios present in the terms t_c and t'_c.

$$t_c \not> t'_c \Leftrightarrow \forall S \in t_c, \forall S' \in t'_c, S \not> S'$$

Rules 9 through 11 of figure 8 uses the scenario extension relations to infer alterability criteria when combining configuration update terms. Rule 9 precises that no criteria on the presence of situations are added when extending a scenario. Rule 10 extends that inference to the extension of several removed scenarios by one new scenario. The new scenario includes the general semantic of each removed scenario. Rule 11 specifies that when no scenario extension is involved, alterability criteria add themselves when two configuration update terms are combined.

5.3 Discussion on reconfiguration calculation

The principal issue when calculating a reconfiguration is the calculation of alterability criteria. A simple solution would be to add the quiescence of all uninstalled or modified components and the absence of any situation handled by a removed scenario while applying the whole reconfiguration as a single update. It is however not satisfying as it could cause higher delays in the application of the update. The probability of meeting all alterabilty criteria at the same time decreases while the number of criteria increases. For that reason we decided to divide each reconfiguration into independent updates and to have these updates as small as possible. Splitting the original configuration term into block allows such division while preventing unnecessary operations like installing a component that was just uninstalled.

By design, SoREn treats components as black boxes which inputs and outputs are markers. The semantic of a scenario lies in the marker types it handles and through which components. Marker types are the building blocks of that semantic. Defining the concept of scenario extension allows the detection of unchanging parts of the semantics of a platform during a reconfiguration and the simplification of alterability criteria. The update instructions may uninstall a scenario and install a new one while keeping the semantic of the former in the platform. Using links, we can detect such situation by detecting when an uninstalled scenario is replaced by an extension of itself. An update extending a scenario can be applied even if situations handled by the scenario are present in the platform. For that update, alterability criteria concerning the absence of situations are unnecessary.

The exclusion of parts from the links is not a problem. Parts are inner representations of assets of the watched systems and do not bring more meaning in the scenario semantic than response types and actuators. Indeed, parts do not specify the actions the scenario takes, response types do. Parts do not specify how these actions are handled, actuators do. Including parts in the links of a scenario would only add redundant information.

6 Implementation and experimentation

We implemented SoREn in Python, using the Pymoult library [Martinez et al., 2015] for the DSU features of the platform. To test the design of SoREn and the automatic reconfiguration calculation presented in this paper, we implemented components and markers for two scenarios and used the two scenarios to test a platform reconfiguration.

6.1 Platform core and base classes

The platform core is build around a central class providing interfaces for the installed components, allowing them to access the markers present in the platform core and to create new markers. That central class is linked to classes handling the internal database, the application of updates and the management of the configuration. The platform core embeds a MongoDB document oriented database, storing measures, situations and responses. Any data structure can be stored in the database provided it is converted into a dictionary's (also called *document*). Document oriented databases are well suited for SoREn because the data model do not need to be defined beforehand.

Figure 9 presents a simplified class diagram of SoREn. Liked to the `Platform` class (fulfilling the role of entry point of the platform), the `Configuration` and `ReconfigurationInterpretor`

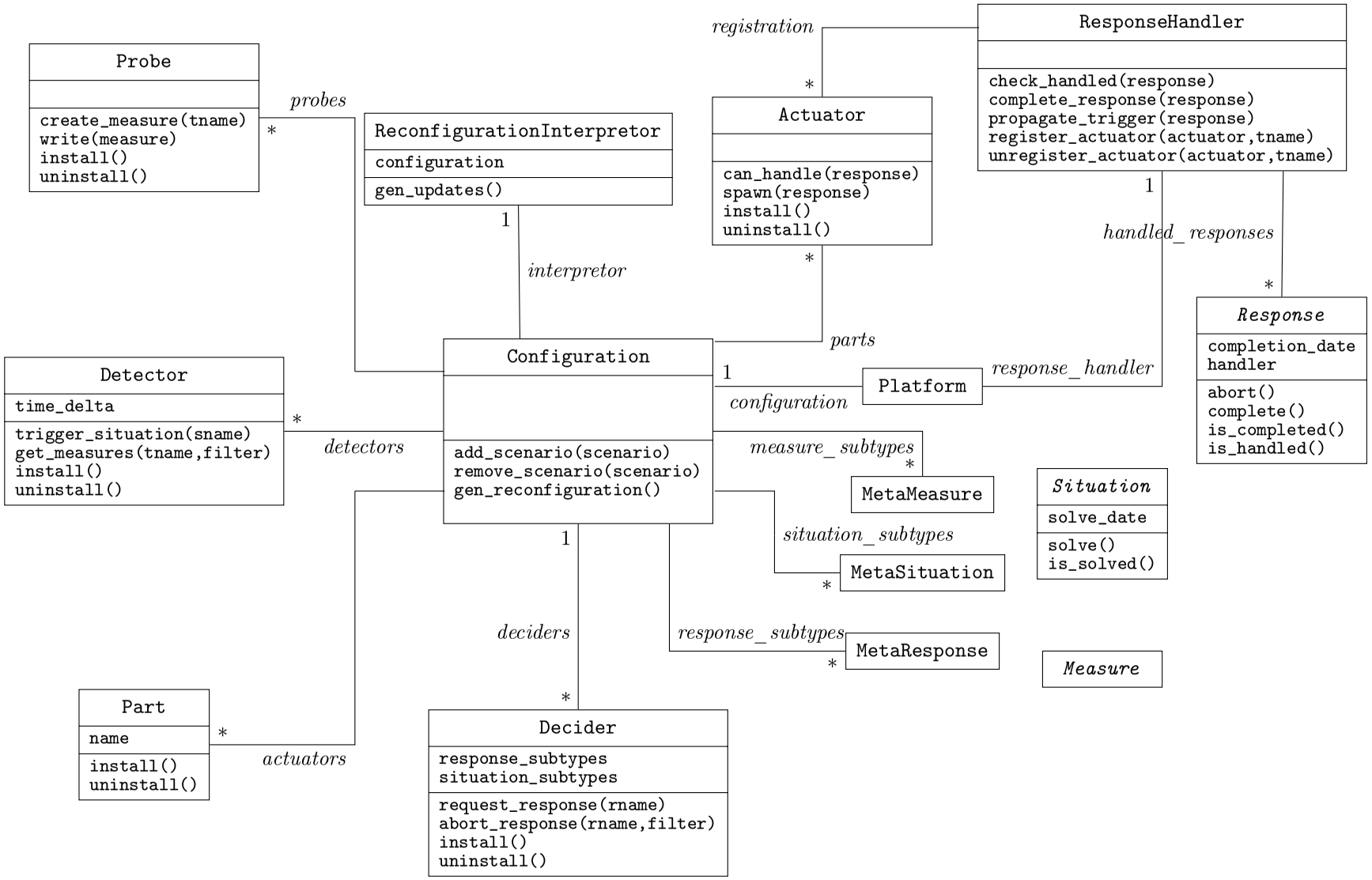

Figure 9: SoREn simplified class diagram

classes respectively hold the curent configuration of the platform and handle reconfigurations. Base classes are defined for each components and marker meta-types. Each component or marker type inherits from these base classes. For instance, to define a new Probe type, one must create a new class inheriting from the `Probe` class embedded in SoREn. Marker types can be defined simply by providing a text specification of their fields. At runtime, that specification is read by the platform and supplied to one of the `MetaMeasure`, `MetaSituation` or `MetaResponse` classes that will generate a child class of the appropriate base class. These three special classes are defined in SoREn using python meta-classes. In Python, meta-classes inherit from the basic Python class `type` and are instantiated into classes (whereas regular classes inherit from the the basic Python class `object` and are instantiated into objects). Because Python meta-classes are handled as regular classes, the meta-classes associated to the marker meta-types are depicted as regular classes in figure 9. When reconfiguring the platform, the text specification of marker types is provided in the manifest file of the installed components. Supplied components are paired with a manifest file specifying component type specific parameters and the specification of each marker type they handle. Because parts are usually more complex, they cannot be generated the same way as other marker types. They may have methods implementing synchronization with the watched system, preventing them from being simply instantiated using a meta-class. For this reason, Part types must be defined as a new class inheriting from the `Part` base class, as if they where component types.

Reconfiguration calculation is handled in the `ReconfigurationInterpretor` class that implements the internal representation of the platform configuration and an interpreter based on the reduction rules detailed in section 5. This module provides an interface class linked to the central class of the platform core. An internal representation of the platform current configuration is kept in the module to help the calculation of reconfigurations. When a new configuration is supplied to the platform (in the format of a structured YaML file), it is parsed and transformed into an internal representation of the wanted configuration. Both internal representations of the old and the new configuration are used to build the original term **Remove** $S_0\, t_0, ...,$ **Remove** $S_k\, t_0$, **Add** $S'_0\, t'_0$, ..., **Add** $S'_j\, t'_j$ that will be simplified. After calculation of alterability criteria, an update object (instance of a Pymoult update class) is created for each block. Update objects are then sent to the update manager (instance of the Pymoult `UpdateManager` class), which handles their application.

Figure 10 presents a deployment diagram of SoREn. The platform runs on a server providing a Linux environment embedding Python capabilities. Components are also deployed on that server. Probes, parts and actuators are linked with embedded parts of the watched system to collect data, synchronize their state and trigger action. In this figure, the server running SoREn is outside of the watched system. However, it would be possible to embed that server inside the watched system.

6.2 Experimentation on the case of a scenario extension

To validate the pertinence of SoREn, we implement components for a case study implying a scenario extension with two objectives :

1. showing that the meta-model of SoREn allows the design of security policies
2. showing that the reconfiguration protocol of SoREn allows simple and fast reconfigurations

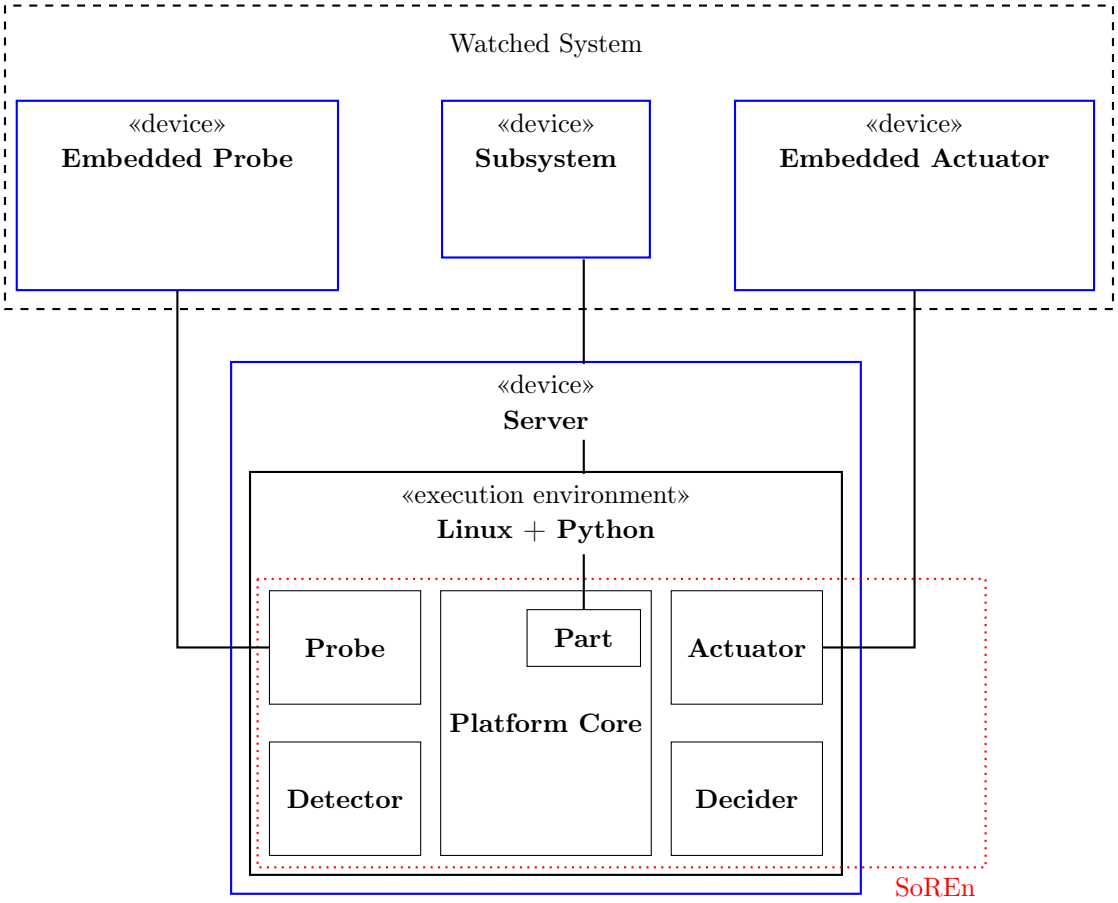

Figure 10: SoREn Deployement diagram

We adress the case of a vehicle communicating wirelessly with a ground infrastructure via Wi-Fi or 4G protocols. By default, the vehicle uses the Wi-Fi protocol but is the target of deauthentication attacks : attackers send false deauthentication frames to the vehicle to disrupt communication or as a preliminary to man-in-the-middle attacks [Sethuraman et al., 2019]. When such attacks happen, we want the vehicle to switch to the 4G protocol until Wi-Fi can be safely used again.

Assuming the vehicle embeds a SoREn platform, we design and implement a scenario that fulfills this requirement. As depicted by figure 11, we define component types `DeauthProbe`, `DeauthDetector`, `DeauthDecider` and `MediumSwitcher` of respective meta-types Probe, Detector, Decider and Actuator. We also define marker types `DeauthMeasure`, `DeauthAttack`, `SwitchMedium` and `Medium` of respective meta-types Measure, Situation, Response and Part. The `Medium` part installed by this scenario represents the communication medium used by the vehicle (Wi-Fi or 4G). It has one attribute which value can be switched to *Wi-Fi* or to *4G* . Responses of type `SwitchMedium` express requests for the medium to be switched. Markers of

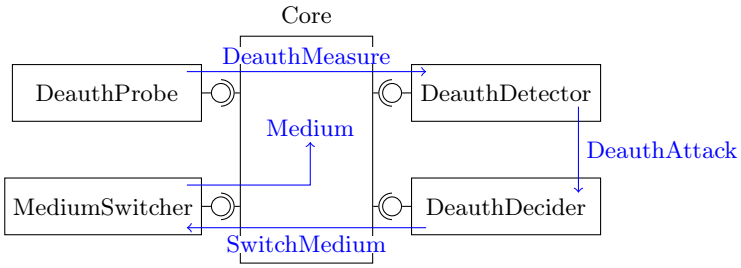

Figure 11: Deauthentication attack scenario

that type have one field : the medium to which communications should be switched (Wi-Fi or 4G). Situations of type `DeauthAttack` indicate that a deauthentication attack has been detected. Markers of that type have one field : the date of detection. Measures of type `DeauthMeasure` indicate the date a deauthentication frame was received. Markers of that type have one field : the frame reception date. Components of type `DeauthProbe` analyze all received frames and generate a measure of type `DeauthMeasure` whenever a deauthentication frame is received. Components of type `DeauthDetector` read `DeauthMeasure` measures and generate a situation of type `DeauthAttack` whenever more than a predefined number n_f^t of deauthentication frame have been received during a predefined time period Δ_f^t. They remove `DeauthAttack` situations from the platform when less than n_f^r deauthentication frame have been received during Δ_f^r. n_f^t, n_f^r, Δ_f^t and Δ_f^r are parameters defined in the scenario that specifies the installation of the components. Components of type `DeauthDecider` generate a response of type `SwitchMedium` with field value *4G* whenever a `DeauthAttack` is created and the `Medium` part is set to *Wi-Fi*. They generate a `SwitchMedium` response with filed value *Wi-Fi* when no `DeauthAttack` is present in the platform and the `Medium` part is set to *4G*. They also delete response markers of type `SwitchMedium` when the requested response is no longer needed (*e.g.* the `DeauthAttack` situation that triggered the generation of the response marker is deleted before any actuator handled the requested response). Components of type `MediumSwitcher` handle responses of type `SwitchMedium` and change the protocol used by the vehicle. They also set the `Medium` part to the proper value and delete before deleting the response marker after completing the requested task.

We implement the components and configure the platform with that scenario, setting values $n_f^t = 5, \Delta_f^t = 200ms, n_f^r = 2, \Delta_f^r = 200ms$. We read frames from a file containing Wi-Fi frames captured using Wireshark [Wireshark, 2020] during a lab experiment reproducing a Wi-Fi deauthentication attack. The setup for the experiment was an attack computer sending deauthentication frames to a target computer connected to a Wi-Fi hotspot.

We assume the vehicle is on service and the platform is running. We discover that, with our scenario, the vehicle is vulnerable to the jamming of 4G communications. We decide that the vehicle should switch back to Wi-Fi if 4G communications are jammed. If a deauthentication attack is detected and 4G communications are jammed, we decide the vehicle should use Wi-Fi and used reinforced protocols at application layer to defend itself in the case of a man-in-the-middle attack. We define an extension on the scenario as depicted by figure 12.

4G Communications [European Telecommunications Standards Institute, 2010] use OFDM modulation and the quality level of the received 4G signal can be measured by the Error Vector

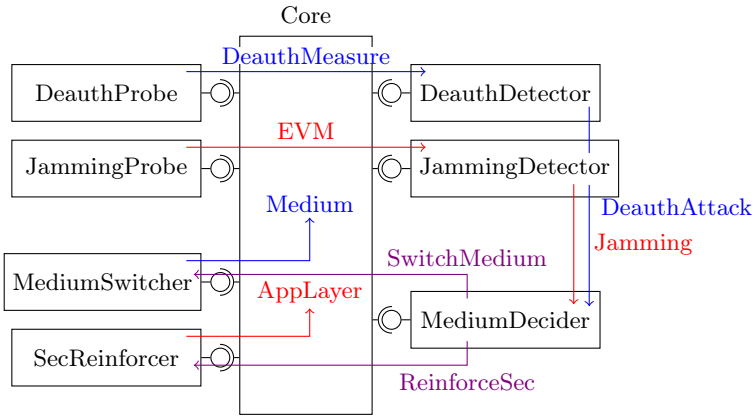

Figure 12: Deauthentication and Jamming attack scenario

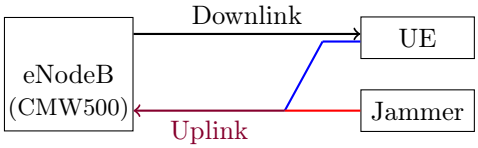

Figure 13: Jamming of uplink 4G signals

Magnitude (EVM). EVM is commonly used to assess the quality of digital communication signals. It expresses as a percentage the difference between an expected symbol and the symbol actually received. The maximum permissible EVM to ensure sufficient quality of the communication depends on the modulation coding scheme. We consider the QPSK modulation which allows a 17.5% maximum EVM. We define the `EVM` measure type with one field containing the value of the EVM measured by the probes of type `JammingProbe`. Detectors of type `JammingDetector` read the `EVM` measures and produce a situation of type `Jamming` when the measured EVM is above 17.5% for more that Δ_j. Like `DeauthAttack` situations, `Jamming` attack have one field : the date of detection of the situation. We also define a new type of decider for requesting responses according to the previous description. If a deauthentication attack is detected, `MediumDecicer` deciders generate a `SwitchMedium` response requesting the medium to be switched to Wi-Fi. If a jamming attack is detected, the deciders generate `SwitchMedium` response requesting the medium to be switched to 4G. If both attacks are detected, the deciders generate a `SwitchMedium` and a `ReinforceSec` responses requesting the medium to be switched to Wi-Fi and the application layer to use reinforced security protocols. A new type of actuator is defined to handle the reinforcement of application layer security and the `AppLayer` part type is defined to represent the application layer security protocols.

We implement the components and define the extended scenario, setting values $n_f^t = 5, \Delta_f^t = 200ms, n_f^r = 2, \Delta_f^r = 200ms$ and $\Delta_j = 3s$. As for the previous scenario, we use the captured frames and use EVM readings recorded during a lab experiment reproducing a jamming attack. Using a CMW500 device as eNodeB and a Huawei dongle as User Equipment, we used the

setup depicted on figure 13. The jammer attacks uplink signals from the User Equipment (i.e. the watched system) and the CMW500 measures the EVM of these signals. After measuring the EVM of jammed and non-jammed uplink signals, we construct a trace of measured EVM with one measure each 200ms, including jamming and non jamming periods. The `JammingProbe` probes replays that trace, creating `EVM` measures, assuming symmetry of the setup : downlink signals are jammed and the UE measures the EVM of theses signals. An alternative to EVM measurement would be measure the BLER (Block Error Rates) of signals which can be done at MAC layer. How this can be done is out of the scope of this paper.

We supply a configuration with the extended scenario to the running platform configured with the previous scenario and examine its behavior, checking that attacks are well handled and the reconfiguration is correctly applied. Figure 14 presents the logs produced by the platform during the reconfiguration. At t_0, the platform is configured with the simple deauthentication attack scenario. A deauthentication attack is detected and a situation is created. As a reaction, the decider creates a response at $t_0 + 0.074s$ requesting the comunication medium to be switched to 4G. The responsed is handled by the actuator at $t_0 + 0.077s$ and completed at $t_0 + 1.08s$ but when the medium is switched to 4G, the deauthentication attack is over. The decider requests the medium to be switched back to Wi-Fi at $t_0 + 1.109s$ which is completed at $t_0 + 2.125s$. The reconfiguration of the platform is requested at $t_0 + 142.295s$, during a deauthentication attack. Because the reconfiguration extends the installed scenario, the update is imediately applied and the new scenario is completely installed at $t_0 + 12.707s$. The newly installed decider named `mediumdecider` requests the medium to be switched back to Wi-Fi at $t_0 + 147.807s$ when the deauthentication attack is over. At $t_0 + 270.477s$, a deauthentication attack is detected and a response is created to request the medium to be switched to 4G. At $t_0 + 270.839s$, a jamming attack on 4G communication and the decider, noticing the vehicle is curently using Wi-Fi, generates a response requesting the reinforcement of application layer security. When the deauthentication attack is over at $t_0 + 271.356s$, the decider cancels the reinforcement of application layer security. Because `mediumswitcher` was already engaged on the switching of protocols, the decider could not cancel the switching of the medium to 4G when the jamming attack was detected. As a consequence, the decider immediately requests the medium to be switched back to Wi-Fi right after it is set to 4G. To test the application of a reconfiguration not extending scenario, we request a reconfiguration of the platform to the previous scenario at $t_0 + 356.766s$ while a `Jamming` situation is present in the platform. Because the new configuration does not handle `Jamming` situations, the application of the reconfiguration is delayed until $t_0 + 377.656s$, when the jamming attack is over. At $t_0 + 377.658s$, the platform has been sucessflly reconfigured two times, without disrupting the handling of deauthentication and jamming attacks.

7 Discussion and Future work

SoREn was designed in the context of the SECOURT project focusing on vehicles. As a consequence, the design of SoREn is specifically targeted at embedded systems evolving in an uncontrolled environment. The meta-model of SoREn is well suited for handling threats in that context but may be less suited in more general contexts. For example, companies usually deploy detection software on their computers and centralize decision on a different server. The decentralized nature of SoREn, having each protected system running their own detection/decision/reaction system, would incur drastic changes in the practices of such companies. For that

```
t0  Situation  DeauthAttack triggered
t0+0.074s  Response  SwitchMedium requested with parameters target=4G
t0+0.077s  Response  SwitchMedium requested at t0+0.074s handled by mediumswitcher
t0+0.855s  Situation  DeauthAttack triggered at t0 solved
t0+1.078s  System  Medium switched to 4G
t0+1.080s  Response  SwitchMedium requested at t0+0.074s completed
t0+1.109s  Response  SwitchMedium requested with parameters target=Wifi
t0+1.120s  Response  SwitchMedium requested at t0+1.109s handled by mediumswitcher
t0+2.121s  System  Medium switched to Wifi
t0+2.125s  Response  SwitchMedium requested at t0+1.109s completed
t0+141.713s  Situation  DeauthAttack triggered
t0+141.822s  Response  SwitchMedium requested with parameters target=4G
t0+141.825s  Response  SwitchMedium requested at t0+141.822s handled by mediumswitcher
t0+142.295s  System  Reconfiguration requested
t0+142.702s  DSU  Decider mediumdecider uninstalled successfully
t0+142.703s  DSU  Part applayer installed successfully
t0+142.703s  DSU  Measure type EVM installed successfully
t0+142.703s  DSU  Situation type Jamming installed successfully
t0+142.703s  DSU  Response type ReinforceSec installed successfully
t0+142.703s  DSU  Probe evmsenser installed successfully
t0+142.704s  DSU  Detector jammingdetector installed successfully
t0+142.704s  DSU  Decider wirelessdenialdecider installed successfully
t0+142.707s  DSU  Actuator securityreinforcer installed successfully
t0+142.826s  System  Medium switched to 4G
t0+142.828s  Response  SwitchMedium requested at t0+141.822s completed
t0+147.598s  Situation  DeauthAttack triggered at t0+141.713s solved
t0+147.807s  Response  SwitchMedium requested with parameters target=Wifi
t0+147.814s  Response  SwitchMedium requested at t0+147.807s handled by mediumswitcher
t0+148.818s  System  Medium switched to Wifi
t0+148.822s  Response  SwitchMedium requested at t0+147.807s completed
t0+169.824s  Situation  Jamming triggered
t0+212.039s  Situation  Jamming triggered at t0+169.824s solved
t0+270.477s  Situation  DeauthAttack triggered
t0+270.698s  Response  SwitchMedium requested with parameters target=4G
t0+270.703s  Response  SwitchMedium requested at t0+270.698s handled by mediumswitcher
t0+270.839s  Situation  Jamming triggered
t0+270.920s  Response  ReinforceSec requested with parameters target=reinforce
t0+270.922s  Response  ReinforceSec requested at t0+270.920s handled by securityreinforcer
t0+271.356s  Situation  DeauthAttack triggered at t0+270.477s solved
t0+271.578s  Response  ReinforceSec requested at t0+270.920s cancelled
t0+271.706s  System  Medium switched to 4G
t0+271.709s  Response  SwitchMedium requested at t0+270.698s completed
t0+271.791s  Response  SwitchMedium requested with parameters target=Wifi
t0+271.793s  Response  SwitchMedium requested at t0+271.791s handled by mediumswitcher
t0+272.795s  System  Medium switched to Wifi
t0+272.798s  Response  SwitchMedium requested at t0+271.791s completed
t0+284.406s  Situation  Jamming triggered at t0+270.839s solved
t0+332.680s  Situation  Jamming triggered
t0+356.766s  System  Reconfiguration requested
t0+374.929s  Situation  Jamming triggered at t0+332.680s solved
t0+377.656s  DSU  Part applayer uninstalled successfully
t0+377.657s  DSU  Measure type EVM uninstalled successfully
t0+377.657s  DSU  Situation type Jamming uninstalled successfully
t0+377.657s  DSU  Response type ReinforceSec uninstalled successfully
t0+377.657s  DSU  Probe evmsenser uninstalled successfully
t0+377.657s  DSU  Detector jammingdetector uninstalled successfully
t0+377.657s  DSU  Decider wirelessdenialdecider uninstalled successfully
t0+377.657s  DSU  Actuator securityreinforcer uninstalled successfully
t0+377.658s  DSU  Decider mediumdecider installed successfully
t0+420.386s  Situation  DeauthAttack triggered
t0+420.555s  Response  SwitchMedium requested with parameters target=4G
t0+420.559s  Response  SwitchMedium requested at t0+420.555s handled by mediumswitcher
t0+421.560s  System  Medium switched to 4G
```

Figure 14: Platform trace

reason, although the pertinence of SoREn was validated for a specific context, discussion of the general pertinence of SoREn would require more experimentation in the future. Automatic update calculation takes advantage of the restrictive meta-model of SoREn. Automatically calculating reconfiguration of more generic systems that are not subjected to such restrictions would be more complex . Indeed, with no a priori knowledge on updates, selecting well suited mechanisms for applying all future updates would be a complex task. A probably easier solution would be to select the used mechanisms at each update, implying that a tool or a person selects the mechanisms. The reconfiguration system of SoREn shows that dynamic updates can be made easier by the definition of a restrictive meta-model for applications. Future work addressing that observation would help understand the impact of the restrictiveness of the meta-model and the model of an application on the automatic calculation of its dynamic updates.

Platform configurations are expressed as a list of scenario to be installed, which in turn are expressed as a list of component and marker types to be installed. As shown in table 1, the proximity between component and marker meta-types with business concepts allows a higher level of specification for configuration, but an even higher level of expression for configuration should be targeted. Platform administrators should be able to reconfigure platform using only business concepts, without considering components and marker types. This goal could be reached by specifying an ontology for the specification of security policies and linking that ontology to the component and marker types. Administrators would use the ontology when specifying a configuration and an extended version of the reconfiguration interpreter would calculate the scenarios to be installed.

8 Conclusion

This paper presented SoREn, a reconfigurable platform enforcing security policies for communicating vehicles. It detailed its meta-model centered on four stateless meta-types of components exchanging information through objects name markers. Component and marker meta-types are associated with high level business concepts to facilitate the expression of configuration. The reconfiguration engine, calculating the update tasks to be executed from a supplied configuration, was also presented. The operational semantic on which the engine is based was detailed. The paper showed the pertinence of the design of SoREn for expressing security policies, enforcing them and for applying dynamic updates without requiring administrators to write updating code or provide any specification on the used update mechanisms.

The example presented and discussed in this paper shows the pertinence of Dynamic Software Updating topics in the study of safety and security issues. DSU techniques allow security systems designs to be more reactive to new needs emerging from changes of context by enabling dynamic reconfigurations. High level specification of updates also increase the reactivity of security systems. Indeed, it allows administrators to order the reconfiguration of the system without needing to develop specific updating code. The time between the reporting of new needs and the effective application of the required update is reduced. Security systems would be updated sooner, leaving security issues unaddressed for a shorter time.

The source code of SoREn is released under the GPL License. Its source code and the code of the components used in the experimentation presented in section 6 can be downloaded on its repository : `https://bitbucket.org/smartinezgd/soren`.

References

[Aldinucci et al., 2005] Aldinucci, M., André, F., Buisson, J., Campa, S., Coppola, M., Danelutto, M., and Zoccolo, C. (2005). Parallel program/component adaptivity management. In International Conference ParCo, NIC Series.

[Appavoo et al., 2003] Appavoo, J., Hui, K., Soules, C. A. N., Wisniewski, R. W., Silva, D. D., Krieger, O., Auslander, M. A., Edelsohn, D., Gamsa, B., Ganger, G. R., McKenney, P. E., Ostrowski, M., Rosenburg, B. S., Stumm, M., and Xenidis, J. (2003). Enabling autonomic behavior in systems software with hot swapping. IBM Systems Journal, pages 60–76.

[Baumann et al., 2005] Baumann, A., Kerr, J., Silva, D. D., Krieger, O., and Wisniewski, R. W. (2005). Module hot-swapping for dynamic update and reconfiguration in k42. In In 6th Linux.Conf.Au.

[Buccafurri et al., 2015] Buccafurri, F., Fotia, L., Furfaro, A., Garro, A., Giacalone, M., and Tundis, A. (2015). An analytical processing approach to supporting cyber security compliance assessment. In Proceedings of the 8th International Conference on Security of Information and Networks, SIN '15, page 46–53, New York, NY, USA. Association for Computing Machinery.

[Buisson et al., 2006] Buisson, J., André, F., and Pazat, J.-L. (2006). Afpac: enforcing consistency during the adaptation of a parallel component. Scalable Computing : Practice and Experience.

[Cavalcante et al., 2015] Cavalcante, E., Batista, T., and Oquendo, F. (2015). Supporting dynamic software architectures: From architectural description to implementation. In 2015 12th Working IEEE/IFIP Conference on Software Architecture, pages 31–40.

[European Telecommunications Standards Institute, 2010] European Telecommunications Standards Institute (2010). Lte; evolved universal terrestrial radio access (e-utra); user equipment (ue) radio transmission and reception.

[Ghafari et al., 2012] Ghafari, M., Jamshidi, P., Shahbazi, S., and Haghighi, H. (2012). An architectural approach to ensure globally consistent dynamic reconfiguration of component-based systems. In Proc of the 15th Symposium on Component Based Software Engineering, CBSE, pages 177–182.

[Giuffrida and Tanenbaum, 2010] Giuffrida, C. and Tanenbaum, A. (2010). A taxonomy of live updates. In Proceedings of the 16th Annual Conference of the Advanced School for Computing and Imaging.

[Gregersen and Jørgensen, 2009] Gregersen, A. R. and Jørgensen, B. N. (2009). Dynamic update of java applications—balancing change flexibility vs programming transparency. Journal of Software Maintenance and Evolution: Research and Practice.

[Kramer and Magee, 1990] Kramer, J. and Magee, J. (1990). The evolving philosophers problem: Dynamic change management. IEEE Trans. Softw. Eng.

[LAMIH, 2020] LAMIH (2015, accessed November 30 of 2020). Cyber-sécurité dans les systemes communicants pour les transports. `https://www.uphf.fr/LAMIH/fr/cyber-securite-dans-les-systemes-communicants-pour-les-transports`.

[Malatji et al., 2019] Malatji, M., Von Solms, S., and Marnewick, A. (2019). Socio-technical systems cybersecurity framework. Information & Computer Security, 27(2):233–272.

[Martinez et al., 2015] Martinez, S., DAGNAT, F., and Buisson, J. (2015). Pymoult : On-line updates for python programs. In ICSEA 2015 : 10th International Conference on Software Engineering Advances.

[Meyerson, 2014] Meyerson, J. (2014). The go programming language. IEEE Software, 31(5):104–104.

[OISF, 2020] OISF (2020, accessed November 30 of 2020). Suricata, open source ids. `http://www.suricata-ids.org`.

[Sethuraman et al., 2019] Sethuraman, S. C., Dhamodaran, S., and Vijayakumar, V. (2019). Intrusion detection system for detecting wireless attacks in ieee 802.11 networks. IET Networks, 8(4):219–232.

[Sourcefire, 2020] Sourcefire (2020, accessed November 30 of 2020). Snort, open source network intrusion prevention and detection system. `http://www.snort.org/`.

[Vandewoude et al., 2007] Vandewoude, Y., Ebraert, P., Berbers, Y., and D'Hondt, T. (2007). Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. IEEE Transactions on Software Engineering.

[Wireshark, 2020] Wireshark (2020 (accessed Novembre 30, 2020)). Wireshark website. `https://www.wireshark.org/`.

[Zeek, 2020] Zeek (2020, accessed November 30 of 2020). Zeek tool. `https://zeek.org/`,.