Title:	*Risk-driven Continuous Delivery of Trustworthy Smart IoT Systems — Final Version*
Authors:	*Franck Chauvel (SINTEF), Franck Dechavanne (CNRS), Jacek Dominiak (Beawre), Nicolas Ferry (CNRS), Thibaut Gonnin (CNRS), Elena Gonzalez (Beawre), Stéphane Lavirotte (CNRS), Victor Muntés (Beawre), Luong Nguyen (MON), Wissam Mallouli (MON), Edgardo Montes de Oca (MON) Phu Nguyen (*SINTEF*), Gérald Rocher (CNRS), Daniel Rudziński (Beawre), Jean Yves Tigli (CNRS).*
Editors:	*Jacek Dominiak (Beawre)*
Reviewers:	*Erkuden Rios (Tecnalia) & Ugis Grinbergs (BOSC)*
Identifier:	*Deliverable # D2.3 v1.0*
Nature:	*Report for Final Software deliverable (Software)*
Date:	*Nov. 30, 2020*
Status:	*Delivered*
Diss. level:	*Public*

Executive Summary

D2.3 provides the final version of the ENACT Continuous Delivery toolkit, including the Orchestration and Continuous Deployment Enabler and the Test, Simulation and Emulation Enabler, the Risk Management Enabler, and the Actuation Conflict Management Enabler. In particular, in this document, we provide an overall as well as a technical presentation of each of the enablers developed in WP2.

Members of the ENACT consortium:

SINTEF AS	Norway
BEAWRE DIGITAL SL	Spain
MONTIMAGE	France
EVIDIAN SA	France
INDRA Sistemas SA	Spain
Fundación Tecnalia Research & Innovation	Spain
TellU AS	Norway
Centre National de la Recherche Scientifique	France
Universitaet Duisburg-Essen	Germany
Istituto per Servizi di Ricovero e Assistenza agli Anziani	Italy
Baltic Open Solution Center	Latvia
Elektronikas un Datorzinatnu Instituts	Latvia

Revision history

Date	Version	Author	Comments
August 20, 2020	Initial	Franck Chauvel (SINTEF)	Draft outline
November 3, 2020	Initial	Luong Nguyen, *Wissam Mallouli,* Edgardo Montes de Oca (Montimage)	First complete version of Section 6
November 7, 2020	Initial	Franck Chauvel (SINTEF), Nicolas Ferry (CNRS)	Section 2, Section 4.1, 4.2, 4.3, and 4.6
November 9, 2020	Initial	Jean Yves Tigli (CNRS), Stéphane Lavirotte (CNRS), Gérald Rocher (CNRS)	Section 5.1 et 5.3
November 13, 2020	Initial	Chauvel Franck, Phu Nguyen (SINTEF)	Section 4.4
November 16, 2020	Initial	Jean Yves Tigli (CNRS), Stéphane Lavirotte (CNRS), Gérald Rocher (CNRS), Chauvel Franck (SINTEF), Nicolas Ferry (CNRS), Jacek Dominiak (Beawre), Victor Muntes (Beawre)	First complete version of the document
November 16, 2020	1.0	Jacek Dominiak (Beawre)	Version submitted for internal review
November 20, 2020	1.1	All authors	Revision

Contents

1 Introduction

1.1 Context and Objectives

To fully exploit the potential of the IoT, it is important to facilitate the creation and operation of the next generation IoT systems that we denote as *Smart IoT Systems* (SIS). SIS typically need to perform distributed processing and coordinated behaviour across IoT, edge and Cloud infrastructures, manage the closed loop from sensing to actuation, and cope with vast heterogeneity, scalability and dynamicity of IoT systems and their environments.

Major challenges are to improve the efficiency and the collaboration between operator and developer teams for rapid and agile design and evolution of SIS. To address these challenges, ENACT embraces the DevOps approach and principles. DevOps has recently emerged as a software development practice that encourages developers continuously patch, update, or bring new features to the system under operation without sacrificing quality. Software development and delivery of SIS would greatly benefit from DevOps as devices and IoT services requirements for reliability, quality, security, privacy and safety are paramount. However, even if DevOps is not bound to any application domain, many challenges appear when the IoT and its requirements for trustworthiness intersect with DevOps. As a result, DevOps practices are far from widely adopted in IoT, in particular, due to a lack of key enabling tools.

WP2 of Enact delivers a set of tools for the development part of the DevOps process (see blue part of Figure 1). These tools aim at improving the **management and continuous delivery of trustworthy SIS. Note that there are other tools, which are developed in the WP3 and WP4 of the project to cover the whole DevOps process.**

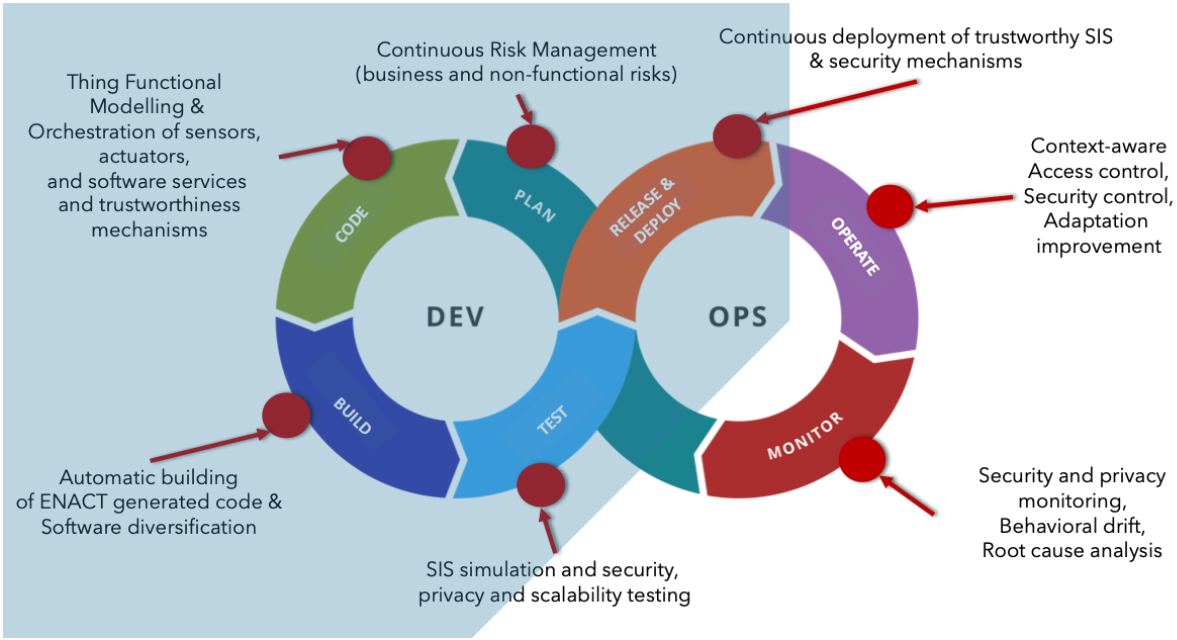


Figure 1. WP2 focuses on the Dev part of the DevOps process.

In particular, WP2 developed four enablers:

1. The ENACT **Risk Management** Enabler: This enabler supports DevOps engineers and architects in managing risks in an agile and continuous way, also supporting the overall development process of trustworthy SIS.

2. The ENACT **Orchestration and Continuous Deployment** Enabler (aka., GeneSIS): This enabler facilitates the development and continuous deployment of trustworthy SIS, allowing decentralized processing across heterogeneous IoT, edge, and cloud infrastructures.
3. The ENACT **Actuation Conflict Management** Enabler: Actuation conflicts can occur when concurrent applications have a shared access to an actuator and when actuators produce actions within a common physical and local environment, whose effects are contradictory. This enabler supports the detection, analysis, and resolution of actuation conflicts.
4. The ENACT **Test and Simulation** Enabler: This Enabler provides the possibility to test the IoT system from very early state (even when there is not any IoT device yet), to the completed IoT systems. The testing scopes are covered from functional testing, operational testing, scalability testing to security testing. With the simulation tool, this enabler helps to test the IoT system in many different scenarios which are not easy to perform the test with a real system.

Figure 2 depicts an example of workflow between these four enablers. First, a DevOps engineer can use GeneSIS to specify the overall architecture of a SIS (①). This model can thus serve as input for the Risk Management enabler, which will help conducting a risk analysis and assessment and may result in a set of mitigation actions, for instance advocating the use of a specific set of security mechanisms (②). As a result, the DevOps engineer may update the model describing the architecture of the SIS before its refinement into a proper deployment model. The DevOps engineer might also use ThingML in order to implement some of the software components that should be deployed as part of the SIS (③). At this stage, the Actuation Conflict Management enabler can be used to identify actuation conflicts - *e.g.*, concurrent accesses to an actuator (④). This enabler will support the DevOps engineer in either selecting or designing an actuation conflict manager to be deployed as part of the SIS (typically as a proxy managing the accesses to the actuator). Finally, the SIS can be simulated and tested, in particular against security threats and scalability issues (⑤) before being deployed by GeneSIS.

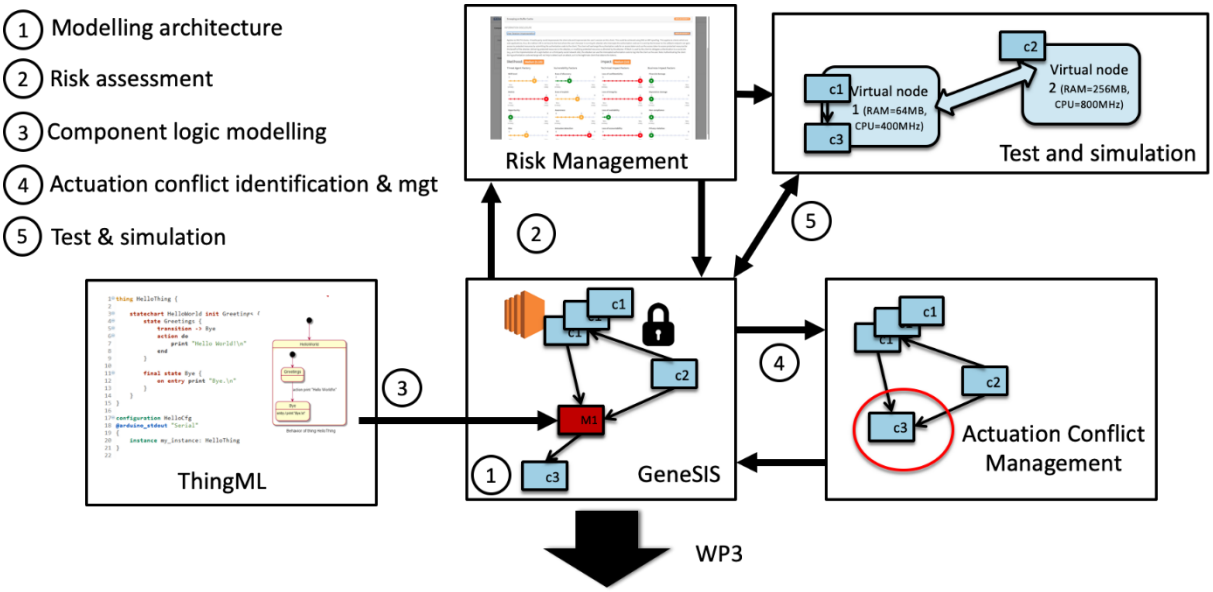


Figure 2. An example of workflow between WP2 tools

Deliverable D2.3 consists of an overview of the state of the WP2 enablers at the time of near finish of the ENACT project. It contains a description of main achievements per each enabler, improvements over the previous deliverables, addresses the state of the evaluation and finally showcases the plans for each of the enablers beyond the project end.

1.2 Support to trustworthiness in SIS

This section describes the characteristics of the support offered by ENACT WP2 tools to the five trustworthiness properties of SIS. The following table summarizes how each of the final tool prototypes developed in WP2 addresses each of the properties.

Table 1. ENACT WP2 methods and tools' support to trustworthiness in SIS at the time of D2.3 delivery.

WP2 tool	Security	Privacy	Reliability	Resilience	Safety
Risk Management	**Yes.** Covered by OWASP and ability to cover security controls.	**Yes.** Covered by privacy controls.	**Yes.** By specification of the mitigation actions.	**No.**	**No.** Embedded within the risk mitigation strategy.
GeneSIS	**Yes.** Specific support for deployment of security mechanisms and for availability	**Yes.** Can specify the privacy requirements within the model	**Yes.** Support for Blue/Green deployment	**Yes.** Only by providing support for adaptation and by supporting rollback in blue/green deployment	**No.**
ACM	**No.**	**No.**	**Yes.** Validate logical and temporal properties of custom actuation manager.	**No.**	**No.**
Test & Simulation	**Yes.** Can simulate malicious and non-malicious situation	**No.**	**Yes.** Can simulate anomalous behaviours	**Yes.** Can simulate system with different conditions	**No.**

1.3 Achievements

As this deliverable provides the final version of the WP2 enablers with the main progress from the previous deliverable summarized in the subsections specific to each deliverable named "Improvements over D2.2" in their respective sections listed in the Table 1 below. **The executable code of the implementations is provided in the ENACT online repository (https://gitlab.com/enact) or others listed and a link to each subproject of each enabler is provided in Table 1. Documentation on how to use the enablers is provided in the respective readme-files of the subprojects in GitLab.** This technical documentation and code are complemented by a conceptual description of each enabler as well as the evaluation results for the enablers described in Sections 3–6 below.

Enabler	Main repository	Details of achievements
Risk Management	https://github.com/eclipse-researchlabs/enact-rm-API	See section 3.1.3 and Section 3.4.1 for reasoning why code of Enact is hosted on Eclipse.
GeneSIS	https://gitlab.com/enact/GeneSIS	See section 4.1.3
Actuation Conflict Management	https://gitlab.com/enact/actuation_conflict_manager	See Section 5.1.3
Test & Simulation	https://gitlab.com/enact/test_and_simulation	See Section 6.2

Table 1. Enablers with sources and pointers to sections listing achievements

1.4 Structure of the Document

The remainder of the document follows the structure of the WP2 and is composed of the following 7 sections. After a brief introduction, Section 2 describes the "Lab experiment" which was set up in order to showcase a case study that aims to use all the enablers of this work package. Section 3-6 provide conceptual solutions for each of the enablers of WP2, improvements over the previous deliverables, evaluation results as well as plans for each enabler after the project is over. Section 7 concludes the document.

2 Smart Home, the "Lab experiment"

In this section, we describe a motivating example inspired by the smart building case study provided by Tecnalia (details can be found in Deliverable D1.2), which needs support for the continuous risk assessment, testing and deployment of its Smart IoT System and the management of actuation conflicts.

The Smart Building IoT System aims at improving user comfort and daily routines in the building and is formed by three different systems as illustrated by the informal diagram shown in Figure 3.

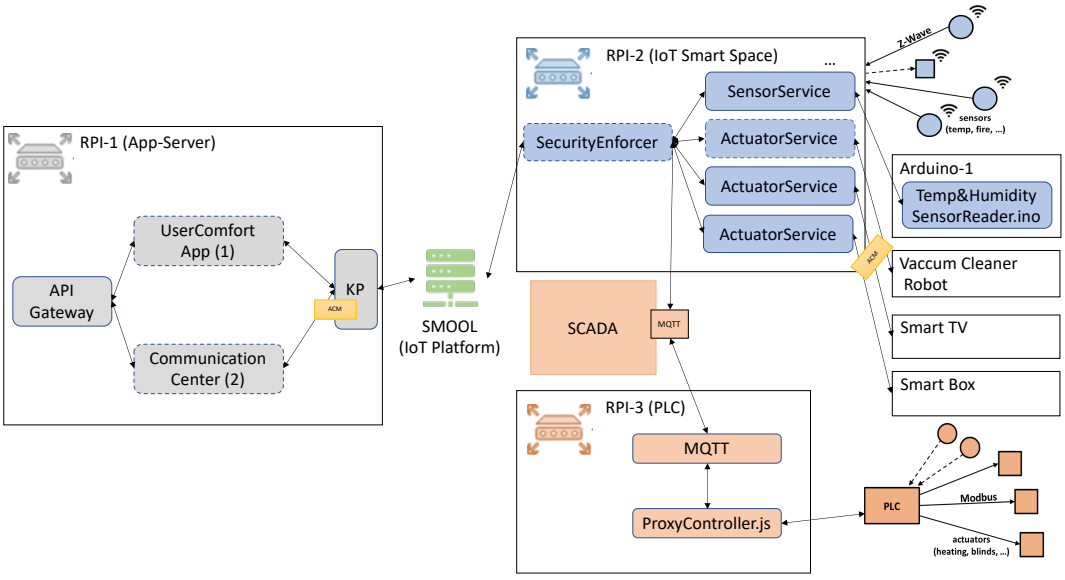


Figure 3. Informal architecture of the smart home

The first system (namely `IoT Smart Space` in Figure 3) includes a gateway (RPI-2) that connects to *(i)* a wireless sensor/actuator network using the Z-Wave protocol and *(ii)* smart devices (*i.e.*, Arduino, Smart TV, Smart TV Box). Whilst the Arduino is physically connected to the gateway using serial port, the Smart TV and Smart TV Box are connected via WiFi. The second system (namely `Building Control` in Figure 3) is a proprietary building control system that consists of a gateway (RPI-3) and a PLC (Programmable Logic Controller) that uses building automation protocols (KNX, DALI, PROFIBUS, etc.) and direct control over the devices through relays or analog/digital outputs. These two systems communicate using an MQTT broker of the SCADA. Finally, the third system (namely `App-Server`) hosts applications that receives sensor data from the sensors and sends actuation commands to the actuators available in the other two systems. The `App-Server` interoperates with the IoT Smart Space and the building control using a Cloud-based instance of the Interoperability SMOOL IoT middleware[1] that has a semantic broker for connecting heterogeneous devices or sources of information. More precisely, the App-Server only communicates with the IoT Smart Space, which in turn forwards authorized and secured messages to the Building Control system. For security reason, all the messages between the App-Server and the IoT Smart Space are checked and controlled by the `SecurityEnforcer` component deployed on RPI-2. All the applications deployed on the `App-Server` are accessible through an `API gateway` ensuring their secured remote access.

For the delivery and deployment of the Smart IoT System, we consider the following three-stage scenario. In the first stage, all the software components depicted in Figure 3 with plain border are deployed. This includes: *(i)* the security and privacy mechanisms, *(ii)* the software components on the Arduino boards, and *(iii)* the `UserComfort` application. The later gets access to sensors' data to make decisions for user comfort and send commands to control the actuators, *e.g.*, roller shutters. In particular, it controls luminosity level and tries to maximize the exploitation of daylights, regulates the in-door temperature and enable basic management of multimedia devices, *i.e.*, starting the Smart TV and Smart TV Box via remote controllers.

In the second stage, a second application (*i.e.*, `CommunicationCenter`) is deployed on the `App-Server`. This application is responsible for ensuring an acceptable ambient acoustic level, including when emitting and receiving phone or audio/video conferences calls. In particular, it has the mechanisms to manage the sound level of multimedia devices such as the Smart TV, the Smart TV box, and Amazon Alexa Dot during such a call. On the one hand, the `SecurityEnforcer` component is updated with a new security policy to provide the `CommunicationCenter` application with access to sensors and actuators. On the other hand, the appearance of the `CommunicationCenter` application introduces direct actuation conflicts. By controlling the Smart TV sound volume with a remote controller, the `UserComfort` application may hinder the `CommunicationCenter` application from maintaining an acceptable sound level (*e.g.*, decreasing or disabling sound volume during a phone call). This conflict must be managed, and the SIS must be updated accordingly.

In the third stage, a new vacuum cleaner robot is added into the IoT Smart Space. The `UserComfort` application is modified with the objective to trigger cleanings at scheduled times. On the one hand, the `SecurityEnforcer` component needs to be updated with a new security policy to provide this application with access to this new actuator, *i.e.*, the vacuum cleaner robot. On the other hand, the modification of the `UserComfort` application results in the introduction of indirect actuation conflicts -- the vacuum cleaner is quite noisy and may prevent the `CommunicationCenter` application from achieving its goal, for instance when the vacuum cleaner is running in a room it is not possible for the person in the room to hold a phone/conference call.

In the following we will detail all the WP2 enablers and use this scenario to illustrate their technical contributions as well as how they facilitate the agile delivery of these three stages. The Risk Management enabler helps DevOps team in planning each stage of the scenario and evaluating the risks. GeneSIS provides the team with mechanisms to continuously deploy the SIS on an infrastructure involving Cloud, Edge and IoT resources. The actuation management enabler and its integration with

[1] noguero27towards

GeneSIS enables DevOps team to detect and manage actuation conflict with agility at each evolution of the system. Finally, the test and simulation enabler can support the testing and validation of each evolution of the system.

3 Agile & Continuous Risk Management

3.1 Overview and Main Achievements

3.1.1 *Overall approach*

Risk Management has been a centre piece of decision making for decades. More so in critical infrastructures and IoT agglomerations. The current proposed Risk Management enabler of ENACT is an evolution of the MUSA[1] (H2020 Project No 644429) Risk Management tool which focuses in assessing risks and mitigation actions of Cloud Security focusing primarily on security related risks.
ENACT Risk Management opens the scope of the risk assessment to any type of risks where the user is free to express the scope of risks from non-intangible non-technical risks down to the tangible technical risks which in effect dictate actionable mitigation actions that need to be included in the DevOps process. The novelty of the enabler comes from the following fact; Risk Management shall be approached in a continuous and agile fashion, which the tool facilitates. ENACT Risk Management enabler aims at embedding risk management in an agile development context in a non-intrusive way[2,3]. In particular, we try to solve several different challenges, namely:

- Traditional risk analysis practices for software development do not easily translate to Agile.
- Analysis of risks should be continuous.
- Development teams (*i.e.*, scrum teams) do not have enough expertise on risk analysis.
- Tools to manage risk in Agile do not foster collaboration.

Besides, in the particular context of IoT, new threats arise from combining several components together. IoT components due to the connection of hardware and software can be a subject of risky situation more often than other type of application components where that consists only of the software paradigm to analyse. Current technology for risk management is mostly focused in detecting threats on specific isolated assets. However, the composition of different assets may also be the origin of new vulnerabilities and threats. ENACT Risk Management enabler also considers this aspect and provide mechanisms for multi-asset vulnerability and threat definitions.

One of the main innovation factors of this enabler is the capacity to perform continuous risk management. The continuous factor of the risk management enables the third-party tools to influence the status of the risks and provide better insights on the actual state of the risk mitigation implementation. ENACT Risk Management tool offers these capabilities via:

- Evidence collection: our tool is connected to JIRA and Git. When mitigation actions are defined, they can be defined in the form of a JIRA issue or connected to a Git repository.
- Ability to integrate with infrastructure monitoring tools, including the one provided by the WP4. Such integration enhances the understanding of the effectiveness of the mitigation actions by providing the state of the implementation of the mitigation as well as notification in case the system encounters any exceptional event or state.

These two capabilities not only allow for factual based evaluation of the state of the risk mitigation but also can be treated as a trigger towards re-evaluation of the risk assessment. The part of the re-evaluation of the risk is a last pending feature that needs to be added to the tool within the scope of the project and it currently being worked on actively.

[2] Victor Muntés-Mulero, Jacek Dominiak, Elena González, David Sánchez, **Model-driven Evidence-based Privacy Risk Control in Trustworthy Smart IoT Systems**, International Workshop on Model-Driven Engineering for the Internet-of-Things (MDE4IoT) colocated with MODELS, Munich, Germany, 2019
[3] Victor Muntés-Mulero, Jacek Dominiak, Elena González, David Sánchez, **Enabling Continuous Privacy Risk Managementin IoT Systems**, Now Publishers, 2020

3.1.2 *Main achievements and innovations*

Table 1. Main achievements since M22.

Feature	Description	Status at M22	Status at M33
DevOps features			
Loading system descriptions (defined though GeneSIS)	See D2.2 and section 3.1.3.4.	Completed	Revised
Collaborative risk management using a threat-based Kanban board	See D2.2	Completed	Completed
Injection of mitigation actions in other planning tools (*e.g.* Jira)	See Section 3.1.3.5. Extended with ability to collect evidences from git based repositories, 3rd party tools for project management such as Oracle Primavera as well as scrapper from normal websites.	Ongoing	Completed and extended
Trustworthiness features			
Automated Vulnerability Detection	See Section 3.1.3.1	Planned	Completed
Identification of risks and mitigation actions (including assessment and reassessment)	See Section 3.1.3.3	Ongoing	Ongoing
Continuous monitoring of mitigation action implementation and efficiency	See Section 3.1.3.6	Ongoing	Completed
GDPR dashboard control	See Section 3.1.3.7.	Planned	Completed

3.1.3 *Improvements over D2.2*

3.1.3.1 Automatic Vulnerability Detection

In order to guarantee the systematic analysis of the vulnerabilities of a Smart IoT System, organizations need to accurately examine their systems' specifications and designs. In order to facilitate an effective identification of security-related risks, it is important to make it easy for users to detect those vulnerabilities that expose the system to attacks that may jeopardize security.

In this section, we describe the Automatic Vulnerability Detector (AVD) implemented in the ENACT Risk Management enabler. An AVD uses a set of data flow diagrams (DFD) to describe a software system under development. Based on these DFDs, it is able to detect potential vulnerabilities to kick off the risk analysis process. The AVD is the result of the joint effort performed in collaboration with the PDP4E project[4]. We provide more details about this collaboration later on in this document.

In order to create the AVD, we propose the following methodology:

i. for each DFD component type and for each STRIDE threat tree[5], we analyse all the nodes in the tree (containing vulnerabilities) and examine the conditions for those vulnerabilities being *relevant* in the system;
ii. we create a list of conditions that need to hold for a vulnerability to be effective;
iii. for each instance of each element in every DFD, we collect information about these conditions when defining the system;
iv. for each component in each DFD related to the system, we filter out vulnerabilities depending on the information collected about these conditions and show those vulnerabilities that are still relevant.

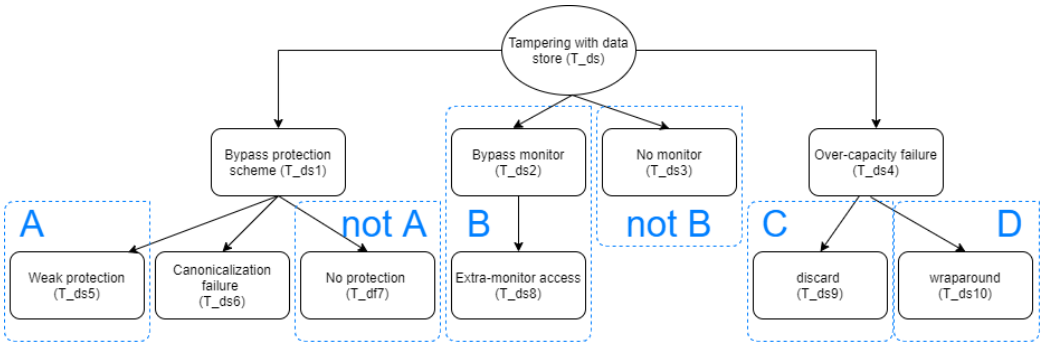


Figure 4. Analysis of vulnerability scope in the STRIDE threat tree for Tampering with data store based on the eHealth use case

As an example, in Figure 4 we show the threat tree related to tampering threats with a data store in a DFD[6]. Apart from the upper node of the tree, which represents the threat under analysis (*i.e.* Tampering with a data store), the rest of the nodes refer to vulnerabilities related to this threat. In the figure, we identify several areas from A to D that we will use to define conditions that must hold for the vulnerabilities in those areas to be relevant. Table 2 provides an example of some conditions that must hold for a subset of vulnerabilities extracted from STRIDE threat trees, related to those areas.

For instance, in area B vulnerabilities related to bypassing the monitoring mechanism used in a data store only become relevant if a monitoring system is implemented for that data store. As a second example, in area C, depending on whether a particular over-capacity policy is implemented, attacks based on exploiting this vulnerability may or may not be an actual threat. Note also that implementation may change along time, increasing or decreasing the relevance of this vulnerability. Therefore, continuous risk management may also include the continuous monitoring of metrics that allow measuring the level of relevance of vulnerabilities or the likelihood of threats to occur.

Table 2. Example of conditions for vulnerabilities related to tampering with a data store to be relevant

STRIDE Tree Area	Vulnerability	Conditions for relevance	Rationale

[4] PDP4E Project | European Project (pdp4e-project.eu)
[5] Shostack, A. (2014). Threat modeling: Designing for security. John Wiley & Sons
[6] Based on the STRIDE Tampering threat tree.

A	The rules (ACLs, permissions, policies) allow people with questionable justification to alter the data (T_ds5).	The data store under analysis implements some protection mechanisms.	If protection mechanisms are not implemented, it is not relevant to check vulnerabilities related to weak implementations. Note that opposite to A is used as a condition to consider vulnerabilities related to not implementing bypass protection schemes (T_ds7).
B	Exploit the lack of a "reference monitor" through which all access requests pass, or take advantage of bugs (T_ds8).	The access to the data store under analysis is monitored.	If a monitoring mechanism is not implemented, it is not relevant to check vulnerabilities related to bypassing the monitor. Note that opposite to B is used as a condition to consider vulnerabilities related to not implementing monitoring mechanisms (T_ds3).
C	When the data store is full data is discarded (T_ds9).	Does overcapacity result in discarding data?	If this policy is not implemented, related vulnerabilities are not relevant.
D	When the data store is full data is written to the beginning of the data store (wraparound) (T_ds10).	Does overcapacity result in overwriting previous data?	If this policy is not implemented, related vulnerabilities are not relevant.

3.1.3.2 Including privacy aspects

ENACT has been actively collaborating with the PDP4E project to exchange research results and integrate them in a unique code repository to create a risk management enabler that included both security and privacy aspects together. While ENACT takes STRIDE as the security threat framework, PDP4E bases their proposal in LINDDUN, detailed in D3.2 of PDP4E project, an analogous tool for privacy. Note that, in most cases vulnerabilities related to a particular LINDDUN threat tree are related to vulnerabilities detected in other LINDDUN or STRIDE threat trees. In order to understand the actual relations between STRIDE and LINDDUN threat trees we did a joint effort to create a tree map that depicts these relations.

Figure 5 describes the detail of this connection in form of a graph, where every node is one of the LINDDUN threat trees (blue nodes) or one of the STRIDE threat trees related to LINDDUN trees (red nodes). As it can be observed, Tampering with a data store (T_ds), the threat tree selected in our example is used by some LINDDUN trees like the Non-compliance threat tree (NC).

In general, in order to understand the vulnerabilities of a particular component in a DFD, it is important to navigate through these connections. For instance, the analysis of vulnerabilities related to the Identifiability of an entity (I_e) generates a cascade analysis of vulnerabilities that may include I_ds, I_df, ID_df, ID_ds, S_e and T_p, following directed edges in the graph. Note that edges are colored in grey if threat trees refer to the same DFD element (*e.g.* I_ds ® ID_ds), and they are colored in red if they refer to different types of DFD elements (*e.g.* I_e ® I_ds). For those relationships represented in grey, we assume that we refer to the same element in the same DFDs. For those relationships represented in red, we have explored them one by one and established a rule to propagate the analysis from one tree to another. For instance, given an entity in a DFD, for I_e ® I_ds, we refer to the data store where the identity credentials of entity *e* or other identifiable account information are stored. This means that we will need to ask for the data store names where identifiable account information is stored for each entity. As another example, for I_e ® ID_df, we will need to examine all the potential vulnerabilities for all the

data flows in the DFD where the origin of the data flow is *e*. We repeat this analysis for all red arrows in the graph.

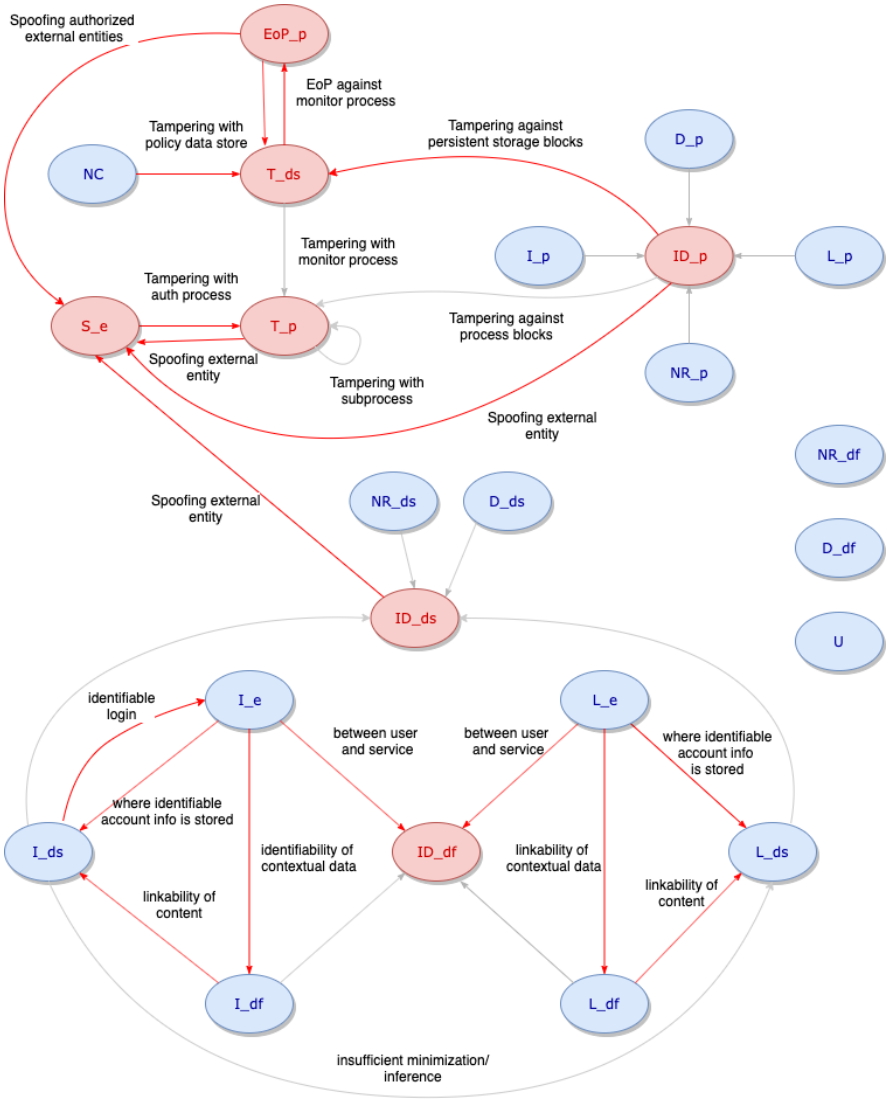


Figure 5. Graph describing the relationship between LINDDUN (blue) and STRIDE (red) threat trees.

In Figure 5, the first part of the ids stand for each of the threat categories of LINDDUN (Linkability, Identifiability, Non repudiation, Detectability, Unawareness, and Non-compliance - Disclosure of information is not represented as it is understood as the Information disclosure already captured by STRIDE) and STRIDE (only for those related to LINDDUN: Spoofing identity, Tampering, Information disclosure and Elevation of privilege). The second part after the "_" represents the type of DFD component (entity, data flow, data store, process). Labels in the edges represent indications extracted from LINDDUN descriptions in the corresponding threat trees.

3.1.3.3 Knowledge Base

In ENACT, we have created a knowledge base to capture information relevant to STRIDE threat categories. In Figure 6, we describe the schema that represents the data schema of the information stored in this knowledge base. This schema has also been defined in collaboration with the PDP4E project. From ENACT perspective, we have created the content related to conditions, vulnerabilities, threats and controls associated to STRIDE.

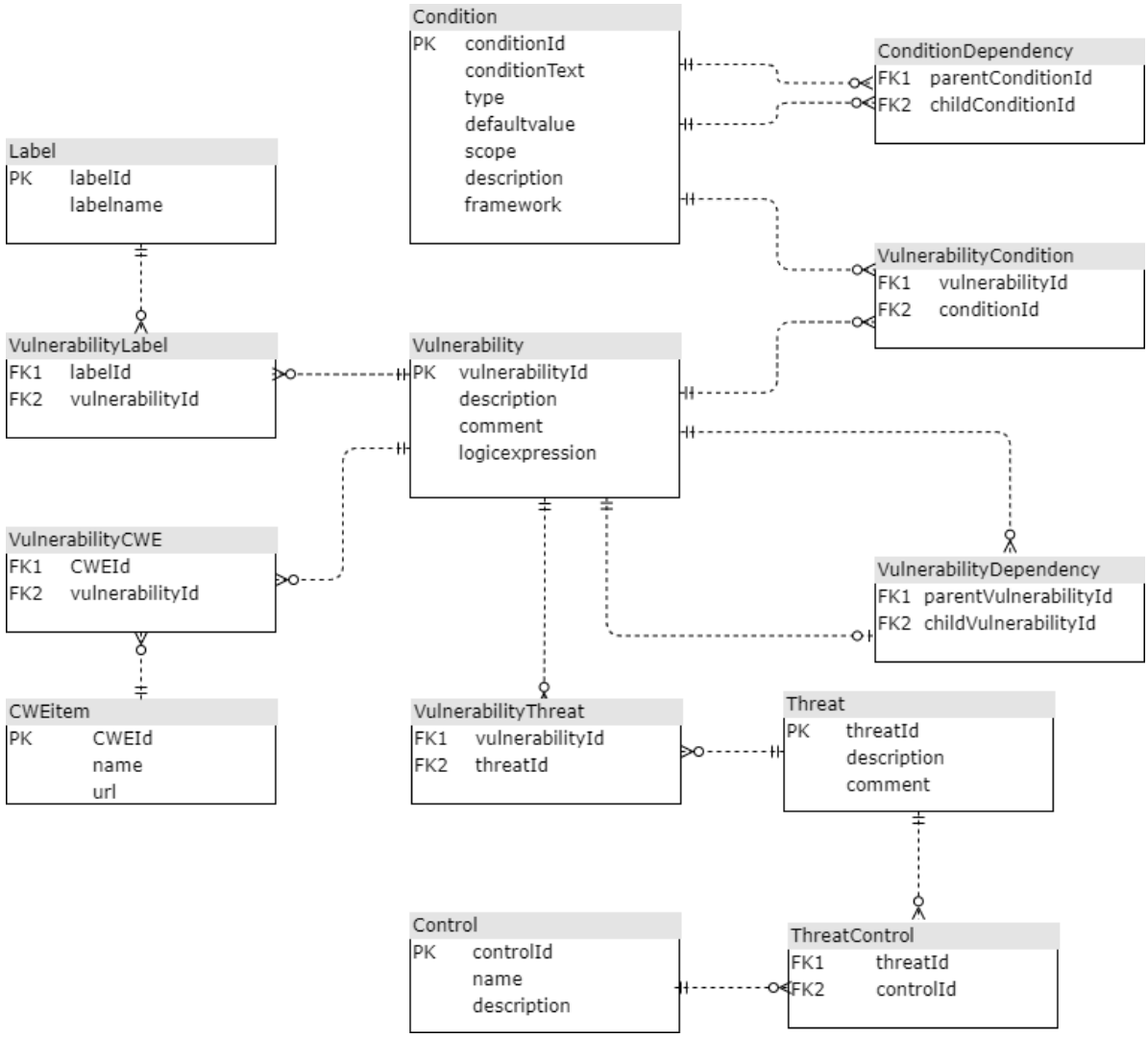


Figure 6. Knowledge Base schema.

The central concept of the schema is the *Vulnerability*. Vulnerabilities are connected to *threats* and threats are connected to *controls*. These link between these three types of elements constitutes the backbone of any risk management tool to provide options related to vulnerabilities and related threats and controls or mitigation actions. As explained in the Automated Vulnerability Detector section, vulnerabilities are also related to conditions. Besides, our knowledge base allows to establish dependencies among conditions. A certain condition may only make sense if another condition holds. For instance, let as assume a condition with id DFD-i_e1 that evaluates the following question: "Does this entity represent a data subject or a proxy to data subjects?". From a privacy perspective, if DFD-i_e1 is true then other conditions may be relevant such as for instance "Does this entity need to be authenticated to execute this DFD?" or "Are credentials of this entity shared with any potentially untrustworthy receiver entity?". So, the relevance of conditions may depend on the evaluation of other conditions.

Besides, vulnerabilities are related to other vulnerabilities. In reality, inspired by STRIDE and LINDDUN, we have also created and extended threat trees in the knowledge base. Therefore, a vulnerability may be decomposed in more detailed vulnerabilities in a structure shaped like a tree. Relation *VulnerabilityDependency* represents the link between vulnerabilities in the tree.

Vulnerability relation is also related to Label relation and CWE item relation. The first link represents keywords related to the vulnerabilities that we will later on to look for cases in

www.enforcementtracker.com. This way, we will be able to show actual cases of GDPR enforcement in cases related to this type of vulnerability. For instance, "Information disclosure of a process" may be labelled with "Confidentiality" label or vulnerability named "Data is not encrypted" may be labelled as "Encryption". The actual mapping between cases and labels has been implemented in the PDP4E project. The second link corresponds to a manual exercise to connect the STRIDE vulnerabilities with CWE items. When the link is established, our tool uses the url stored in the database for each CWE item and scans online information to automatically detect mitigation actions and add them as controls in the Control relation. We also follow links to the CAPEC database related to attack patterns related with that vulnerability and store the information in the Threat relation and potential connected mitigations in the Control relation.

3.1.3.4 Graphical approach to analysis

The approach to risk management was usually based on the lists of risks and estimations of the risk as well as analysis of the risk likelihood and impact of the risks. This in effect traditionally produces a risk matrix representation. Although visual, this representation is not the most effective mean of representing the risk management process of the architectural components, such as IoT system architecture and its components. ENACT Risk Management enabler takes a slightly different approach on the topic. In the application view of the enabler shown in Figure 7 an overall view of the application is shown. Each of the components of the application are represented by an icon. Graphical editor allows also to express connections between the application components. Ideally, this is consumed from the GeneSIS model loaded to the tool.

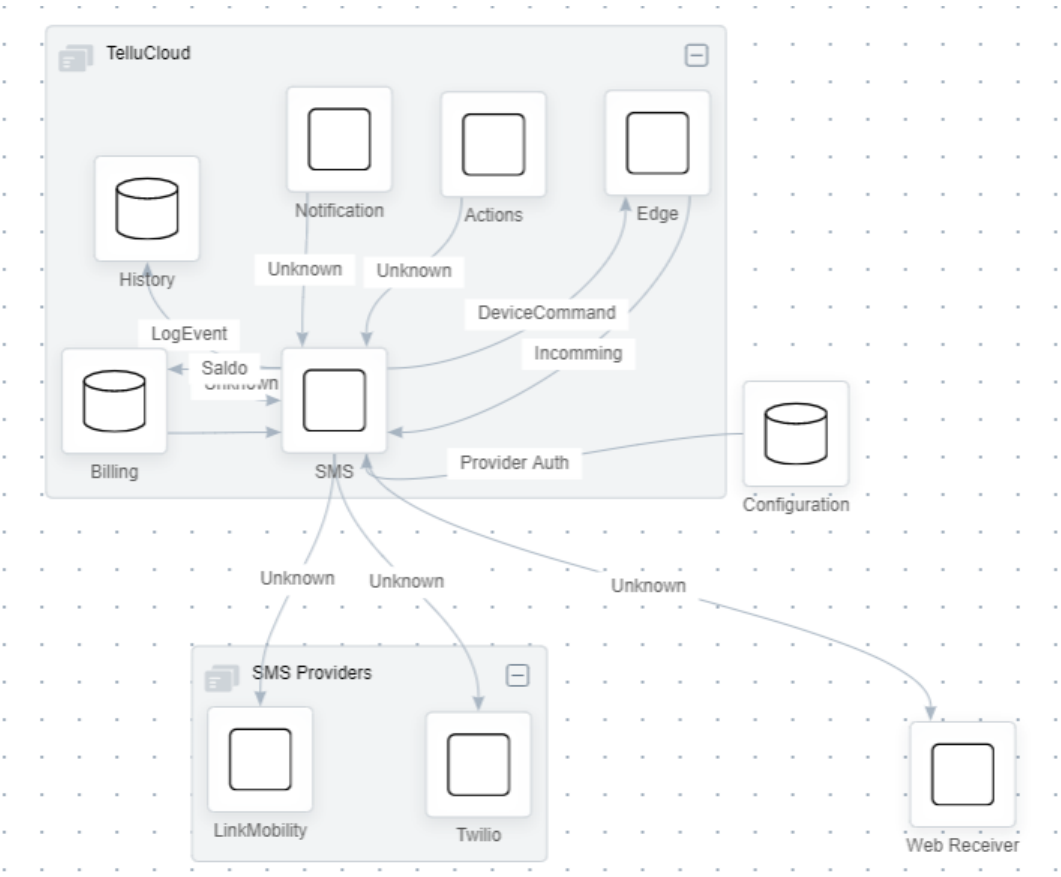


Figure 7. Example of Risk Components description based on the eHealth use case.

Additional architectural models are allowed, such as a group of components. Groups are used in order to enable the risk management on top of the group of the components where the risks associated are only relevant when the group of components are connected together.

Additionally, the editor allows for:

- Visualization of the status of the analysis by clicking on the application component. In the future releases of the tool, the status of the analysis will be shown directly in the graphical representation of the component or group. This in effect should enable a glance view of the architecture and its status of the analysis.
- Representation of the application components among three main component types:
    - Storage component – used to store data either within the application or via external software such as databases
    - Application view – used to represent the component which displays the data to the end users or / and enables the user input.
    - Generic component – used to represent components carrying logic of the application as well as other type of actions not categorized by storage or application type of component.
- Possibility to jump into the risk analysis by selecting the component or group of components and clicking on the button "Go to Analysis" within the details side panel of the editor.
- Evaluation of the Data flow component by addressing its specification via the associated questionnaire (only applicable to DFD components within the "Data Flow Diagram" of the editor).

The aim of the graphical editor in the Risk Management tool is to become a live editable dashboard of the IoT system at stake and drive the analysis of the initial risk assessment as well as subsequent continuous risk management and monitoring.

3.1.3.5 Evidence collectors of Git

In order to address one of the requirements of the market and the use cases identified within the Deliverable 1.1 of the project, an evidence collection notion was added to the tool. These evidence collectors provide snapshot of events which change the status of the mitigation actions. This was described in detail in D2.1 and D2.2.

During the evaluation of the use case requirements of the eHealth use case it became apparent that the treatment status monitoring should be capable of consuming the code repository would beneficial for understanding the status of the mitigation action implementation. As such, a new type of treatment type was added to the list of treatment specification view. This "Repository" type of the component requires the specification of the repository to be monitored, a minimal of read-only authentication key in order to query the repository and the branch name which will represent the treatment. Specification of such treatment by example is shown in the Figure 8.

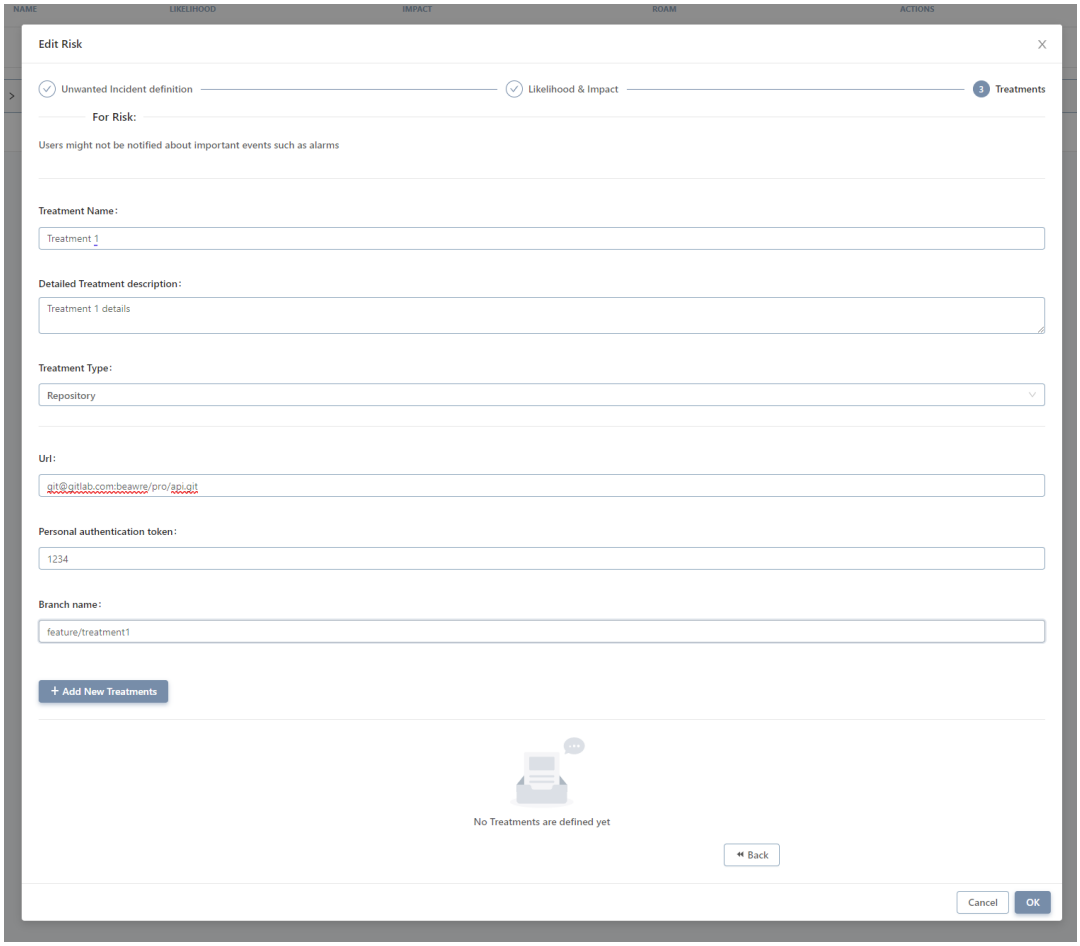


Figure 8. Set up of GIT based treatment monitoring

Once the treatment is added, a subsequent job is added in the Risk Management scheduling engine, which does query the repository on a periodical basis and collects evidences of changes within the repository. This is then directly translated to the status of the treatment as specified bellow:

- **branch name not existent on the repository -** the evidence collector marks the treatment status as "planned",
- **branch name created in the repository –** the evidence collector marks the treatment status as "in progress",
- **branch name merged with the master of develop branch of the repository –** the evidence collector marks the treatment status as "completed",
- **branch name removed before the merge to branch master or develop of the repository –** the evidence collector marks the treatment status as "exception".

At any time, the user can see the evidences collected from the repository. Each of the change of the repository is marked with statement composed of:

- **timestamp of the change –** taken directly from the repository,
- **owner of the change –** an actor responsible for the subsequent push to the repository,
- **change Id –** id of the commit which marked the change of the treatment status,
- **message –** commit message passed in the connected repository event.

All of the above enables DevOps engineers to focus on the everyday operations without the need of frequent status update in the project management tool. It also allows for collection of the factual information about the status of the treatment to enhance the control of the overall risk control process.

3.1.3.6 Reporting & Compliance

One of the main requirements for risk management within the IT sector is compliance with development process risk assessment controls. These are often required when developing software for highly critical environments in eHealth, transportation etc. Ever present software poses a greater challenge to understand the state of the risk not only during the process of building the IoT based software, but also when deploying or even when using such solutions.

In order to address such requirement and also address the commercial requirement identified by the partner of the project, TellU, Risk Management enabler got equipped with ability to generate the risk overview reports. These reports address two main requirements set by TellU which represents eHealth use case in the project. Example of the interface of the enabler is shown Figure 9.

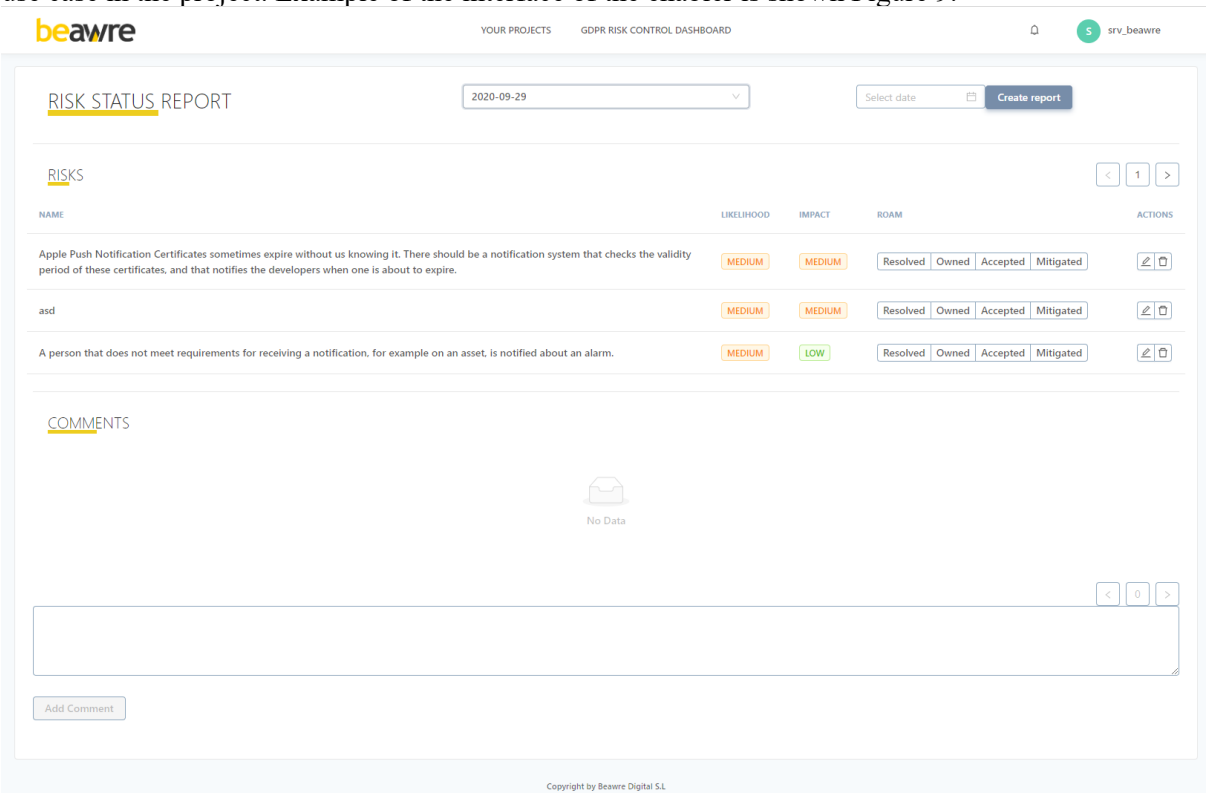


Figure 9. A showcase of the report building page

This certificate is of great importance in the emerging eHealth space to comply with ISO 27001 certificate which treats about the information risk management. One of the required steps to comply with the certificate is to hold periodical risk management board meetings. These meetings aim to understand current risks status within the company and adjust the necessary mitigation actions. Once reviewed, new status of mitigation action can be set, which is then reviewed by the risk manager and possibly planned and implemented. This workflow is currently supported by Risk Management enabler. The Risk Status report produces a snapshot of risks at the time of the creation where the new likelihood / impact status can be set, a treatment added. Descriptive analysis of the status is also possible directly in the report risk analysis or within the main comments section offered by the Risk Management enabler. These saved reports can then be used for risk compliance purposes either directly from the tool or by exporting them to the desired format via the GraphQL endpoint in the tool. Beawre commercial offering of the tool builds on top of the interfaces in order to enable export of the data directly to the third party compliance tools set by the compliance authorities.

3.1.3.7 GDPR Questionnaire

The legal consequences of not complying with GDPR guidelines have been clearly defined as to leave very little to the imagination. Companies in violation of the GDPR may be fined between 2 percent to 4

percent of their annual global turnover up €20 million, depending on which is higher. Frequent GDPR violations can raise the level of legal penalties to the €40 million range. GDPR penalties are becoming more and more frequent and do affect company regardless of their size.
In order to mitigate that, new solutions for GDPR compliance are flooding the market. ENACT Risk Management takes systematic approach to this topic. The GDPR is a complex 11 chaptered document with 99 articles that cover a wide range of user privacy issues. This set of regulations can be hard to digest and interpret, which is where this checklist enters the picture. Fortunately, many governmental bodies as well as EU experts itself work towards making the compliance more approachable. One of such official bodies is ICO.[7] Information Commissioner's Office, in short ICO, is a "*UK's independent authority set up to uphold information rights in the public interest, promoting openness by public bodies and data privacy for individuals*." ICO's work on an official approachable checklist of the GDPR principal has been outstanding. It does provide a simple checklist of steps which does need to be taken within the company and its products and services in order to assure that GDPR compliance is fulfilled.

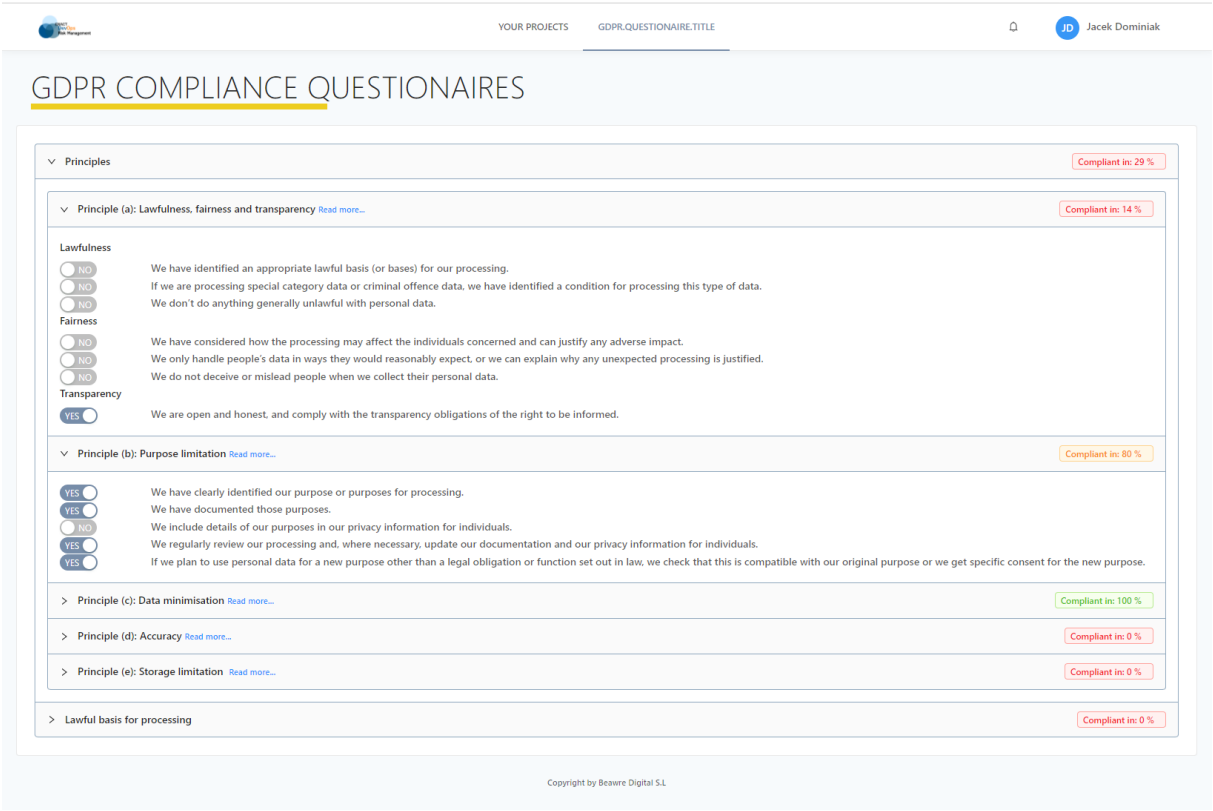


Figure 10. GDPR Compliance Questionnaire embedded in Risk Management tool

ENACT Risk Management builds on top of this checklist and integrates the simple set of questionnaires, shown in Figure 10, directly within the tool in order to foster better risk management where the treatment actions are directly resulted from the checklist questions. The Questionnaire also provides a quick overview of the status of the compliance against each of the GDPR principals. This can be directly used in order to prove the company compliance if subjected to a compliance audit.

[7] https://ico.org.uk/

3.2 Risk Management Enabler

3.2.1 *Risk Management Methodology Updates*

We have slightly updated the Risk Management methodology presented in D2.2. Next, we present a summary indicating the main aspects that are affected and what improvements over D2.2 are associated to these aspects:

- While vulnerability specification was initially considered an optional step, this step has been promoted to become a first-class citizen in our process. The amount of potential vulnerabilities and threats in a medium-complex system may be large, especially when the scope of the detected risks goes to the low level of the technical implementation aspects of each component in a system. Therefore, this made it specially challenging for engineers to have to manually identify all vulnerabilities. In order to make it actionable for engineers, we decided to include vulnerability detection as a necessary part of the Threat Identification step and automate the process as much as possible. This motivates the Automated Vulnerability Detector presented in subsection 3.1.3.1.
- The need for including privacy aspects, made us evolve from architecture-based description of the aspect towards the definition of Data Flow Diagrams (DFD). The main reason for this evolution is that, beyond STRIDE and through our collaboration with PDP4E project, we have merged STRIDE and LINDDUN frameworks in the same tool. The system representation used in LINDDUN is the DFD, which by the way it has also been a common way of representing the system in the context of STRIDE. Thus, our enabler is able to allow for both an architecture-based representation of the system as well as a representation based on the definition of DFDs. In particular, for privacy-related risk management, the specification of DFDs will be necessary, to enable the application of LINDDUN and the LINDDUN threat trees in particular. The information linking these two frameworks is coded in the knowledge base proposed in Subsection 3.1.3.3.
- In terms of Risk Assessment, we consume results from the PDP4E project and include the extension proposed in that project for estimation of risks related to privacy. As it is known, risk assessment is composed of risk identification, estimation and evaluation. In particular, we include in our tool the risk rating extension of OWASP proposed in PDP4E.
- Because, risk management is different from compliance, our risk management tool assumes that the “risk of not being compliant” is not acceptable unless it is caused by the residual risks after making all the reasonable steps to implement mandates from well-defined compliance policies. In the extension to privacy, because it was important for a potential user of the tool to make sure the organization had implemented all reasonable action for compliance with GDPR, we have implemented a GDPR Questionnaire which consists of a checklist of aspects that are essential for compliance. This effort is a direct contribution of ENACT project, as compliance is managed differently from PDP4E project’s perspective and this tool extension was not necessary in that context. See Subsection 3.1.3.8.

The architecture presented in deliverable D2.1 has not been modified and therefore we just cite the document that contains detailed information about it.

3.3 Evaluation

3.3.1 *KPIs & Requirements*

In the following section is explained how our approach addresses the requirements defined in Section 3.3 of deliverable D2.1 where two use cases expressed their requirements that need to be covered by the enabler. In the table below, the case is referred to as UC2, and the *Smart Building* use case is referred to as UC3.

Table 1: WP2 requirements for Risks Management. Underlined text represents text that has been refined with respect to previous deliverables.

ReqID	Requirement	Description	Status at M15
UC2 **R1**	Risks Overview	The tool is ought to provide means to express and analyse all types of risks, including risks on technical and non-technical assets.	**Covered.** This feature is implemented. Users are free to express any type of risks and there is a mechanism to connect related risks. Users also can use reporting capabilities as well as Kanban in order to check the status of the risks at any given time.
UC2 **R2**	Risks Status	The tool is ought to provide means to check the status of the risks and their mitigation at any given time, that being development or operation time. Within the DevOps cycle, risks analysis is believed to be continued so the importance of understanding the status of risks becomes critical.	**Covered.** This feature is covered by number of views provided in the tool, One of which is Kanban representation of the risk management process. The other one, is Treatment status dashboard that lists all the treatments identified in the project as well as risks connected to them. Another view is the graphical representation of the IoT application that shows the aggregated status of the risks against the component to which the risks are associated. Lastly, a reporting view where all the risks are aggregated into a snapshot and shown across the associated projects.
UC3 **R3**	Active cross actor collaboration	The tool is ought to enable communication and collaboration between the actors of risk management process in order to foster better understanding of risks and the steps required in order to fulfil the mitigation strategy.	**Covered** by Kanban-like representation provides a tool that is collaborative by nature. The current version allows different stakeholders to participate in the risk management process.
UC2 **R4**	Treatments implementation prioritization	The tool is ought to provide evaluation means which would enable the actors to understand the impact of mitigation actions on the current development plan and accommodate it within the software development process.	**Covered.** The enabler supports specification of the risks by likelihood, impact and ROAM status. This has a direct effect on the treatments prioritization. Furthermore, integration with JIRA enables the treatment to be used as a work order and therefore prioritized among any other work which needs to be performed by a project or product manager.
UC2 **R5**	Mitigation impact on operations	The tool is ought to provide mechanism to assess if a change of the architecture might be necessary in order to mitigate the risks. This is especially true in IoT, hybrid highly dynamic environments where Enact is planning to make the most impact.	**Covered.** The enabler allows to run architectural scenarios with pervious analysis and in effect addresses the requirement by providing a risk view on each of the proposed deployment scenario. This can later be used as an aid for evaluation and decision making.
UC3 **R6**	Personalized, architecture crafted mitigation actions	The tool is ought to provide suggestions on the mitigation actions which take into consideration the type of the architecture against the risks analysed as well as types of risks which might occur within whole or subset of the IoT architecture.	**Covered**. The enabler allows for running parallel architecture analysis for the different types of mitigation actions implementation.

UC3 **R7**	"Just enough" level of risks setting	Since unnecessary mitigation actions may be introduced during the risk analysis process, which may be costly, the tool is ought to provide the necessary means to evaluate the minimum subset of treatments to consider a specific risk mitigated.	**Covered.** The enabler allows for fast risk status based on the ROAM strategy in order to analyse the "just enough" level of the risk mitigation.
UC2 **R8**	Treatment ineffectiveness detection	The tool is ought to provide means for detecting if previously defined mitigation actions are not effectively mitigating some risks. The current proof of concept partially implements the features related to this requirement. It provides a treatment status dashboard.t	**Covered.** Ability of the enabler to consume the evidences from third party tools as well as underlying ability to assess the evidences through the notions of jobs enable this requirement to be effective. Furthermore, provided "Treatment status" dashboard allows for a at a glance view of the system, including exception detection, with ability to show which exact evidence triggered the exception status.
UC2 **R9**	Release schedule impact	The tool is ought to provide means to evaluate the impact of the mitigation actions against the current release planning, so that actors can detect potential clashes of mitigation actions vs. planning.	**Covered.** The enabler allows to automatically create actions within JIRA which is used for release schedule and work organization. This is both way integration, which means that the tool can both push the new issues, attach treatment monitoring to an existing issue in JIRA but also monitor the progress of the work scheduled by querying the tool for the changes in the status of the ticket. Evidences supporting the change collected from the Release schedule tool are directly visible in the enabler.
UC3 **R10**	Architecture weak points detection	The tool is ought to provide means to evaluate the architectural robustness of the IoT application by automatically matching risks to the architecture described. This would enable the possibility of enhancing the architecture during the design time without exposing the application to potential risks which can be addressed otherwise and become cured.	**Covered.** The enabler allows to run concurrent analysis of different architecture strategies along with the copy of the previous analysis from the other analysis to build on top. This allows for detection of the weak points of the architecture through analysis of the risks associated within each of the approaches and impact analysis of architecture at stake.

3.3.2 *Other evaluations*

Due to the open-source strategy described in detail in the subsequent section of this deliverable, the Risk Management enabler is a subject of further evaluation by the Eclipse foundation. As the final results of the project will be released as a separate Eclipse project, the foundation is actively evaluating the value proposition of the tool, the usefulness to the market as well as quality of the tool as a whole, including the evaluation of the source code. The evaluation however does not include the approach to the problem or the coverage of the domain knowledge by the tool. As of the time of writing of this deliverable, most of the checks have been successful and minor changes to the code are being done in order to ensure the standard set by the Eclipse foundation.

3.4 Beyond ENACT

3.4.1 *Open Source Strategy*

Although ENACT Risk Management tool was meant to be open-sourced from the very beginning of the project, the commercialization of the tool outcome described in the next section dictated an approach to the open-source outcomes of the project. The enabler is fully open sourced and consists of the 11 modules which do compose the open-source version of the enabler. These modules are released under EPL v2.0[8]. Due to participation of Beawre in the other European funded project called PDP4E, where Eclipse Foundation is part of the project consortium, Beawre agreed to release the ENACT Risk Management enabler via the Eclipse project.

The Eclipse Foundation has been a partner in many publicly funded research projects since 2013. They help organizations to successfully create, publish, and sustain an open source software platforms, making the results of the research projects available for commercial or public exploitation.

The enabler is being released via the Research channel. The Research @ Eclipse foundation is specifically designed to foster exploitation of the research projects. In Figure 11 we show how the Eclipse foundation fosters the exploitation of the research results.

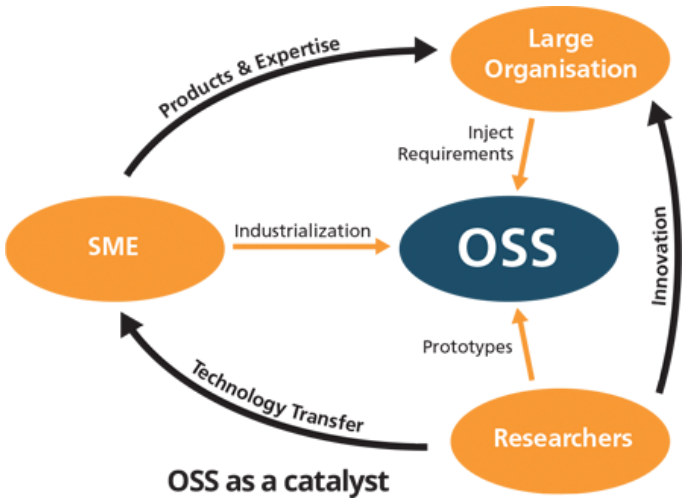


Figure 11. Value proposition of the process at Research @ Eclipse Foundation

Eclipse foundation states that: *"experts enable the research team to create, evolve and maintain open source software projects that capture the results of the research"*.

The proven processes of the Eclipse Foundation and services for IP management and community building assist the research partners throughout the lifecycle of the research project. They ensure that the open source code can be used at any given time for research or commercial exploitation.

These processes are set out in the Eclipse Foundation governance model, the Eclipse Development Process and the Eclipse IP management practices. In addition, all code is made available under the Eclipse Public License, a commercial-friendly Open Source license approved by the Open Source Initiative."[9]

Following that approach Beawre is already hosting the results of the projects in the code repository of Eclipse, where it is currently undergoing the process of evaluation and alignment to the standards set by the foundation. Within the next months, a new Eclipse project will be created, which will represent the outcomes of the project under new generic trademarked name. A set of exploitation actions are already planned in order to ensure the outcomes continuity. These are: announcements to all members of the

[8] https://www.eclipse.org/legal/epl-2.0/
[9] Ref: https://www.eclipse.org/org/research/

foundation about the new project being appointed, talks at the EclipseCon[10] to showcase the project and others.

Lastly, the state of the art technology stack of the enabler ensures that the project will keep on being easy to maintain. Beawre embeds the enabler modules produced in ENACT project in its product offering. This is further exploited in the next chapter of this document. Such set up guarantees continuity of the results far beyond the project end, both from the open-source but also commercial perspective.

3.4.2 *Commercial extensions of the tool.*

Beawre, which is the main contributor to the ENACT Risk Management enabler, as a company was set up within the duration of the project. As such, the baseline of the company was established from a ground up to disseminate and commercially exploit the results of the project.

ENACT Risk Management enabler was organized in modules in order to allow for scalable and deployable risk management solution to be formed regardless of the knowledge base space used. Around 70% of the project results are directly applied to the commercial solution offered by Beawre in the commercial spaces. Visualization on the deployment of the project results and how the results on the project are directly embedded within the commercial solution. It is important to note that 100% of the core functionality used in open-source version of Risk Management enabler is encapsulated and available to the commercial user. Each of the modules can be switched on or off depending on the need.

On top of the enabler, two closed source evidence collectors are a direct result of the requirements identified by the ENACT project use cases. These evidence collectors support the continuous risk management process described in the deliverables D2.1 and D2.2 and are integral part of the evaluation of the eHealth use case.

The commercial exploitation of the project results encapsulated within the commercial solution of Beawre is directly aligned with the project aims. That is, to enhance the DevOps cycle with capability to ensure that the IoT based solutions proposed are of the highest trustworthiness value. Through the Beawre Risk Control tool, our value proposition is to remove the burden of handling risk management and make it actionable, understandable and with enhanced compliance. This is achieved by allowing splitting the risk management into two main factors, one which allows for standardized risk identification and assessment, and the other which enables continues risk control in a non-invasive, benefit-first way. The risk assessment proposed includes the method supported in ENACT Risk Management tool with addition of the automatic vulnerability detection. The automatic vulnerability detection has been described in more detail in the section 3.2.2 of this deliverable.

Very early in the project, we have identified that the approach to risk management shall follow a non-invasive approach, such that the benefit of the risk management should be allowed with the minimal possible effort.

[10] https://www.eclipsecon.org/

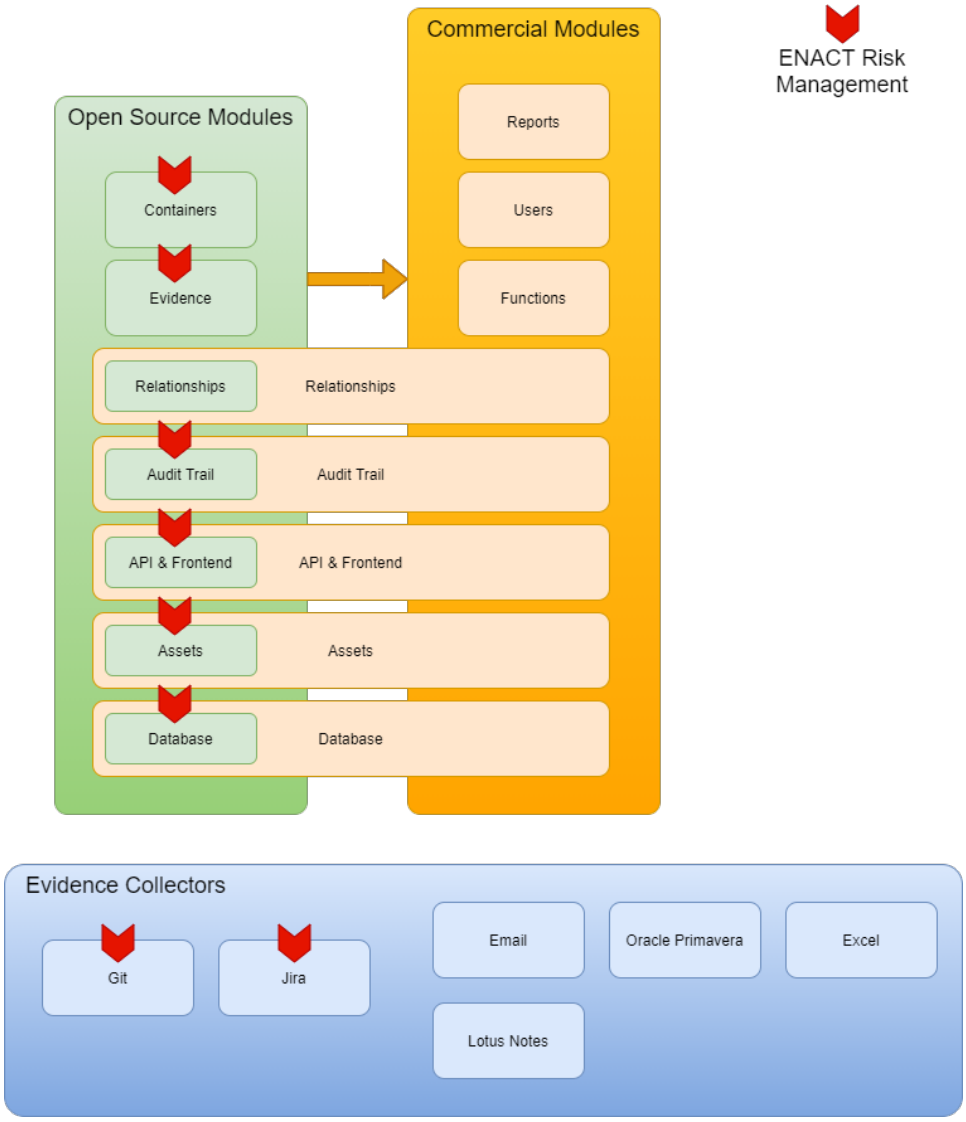


Figure 12. Visualization of the project results encapsulation in the commercial solution.

Minimizing requirement of feedback from DevOps team is achieved by deploying evidence collectors, that collect factual evidences from the IoT and DevOps cycle systems. In the case of the use cases shown in the project as well as Beawre customers, we see that for the scope of risk management, collection of evidences from DevOps cycle is the most important. Further evidence collectors have been developed outside of the project, which are compatible with the open source version of the enabler.

These evidence collectors facilitate the necessity of the market to provide risk compliance reports that state the factual information of the state of the system at the time of the report generation. In the commercial solution Beawre enablers such integration as well as provides a set of functionalities which allows for the Risk committee board in the company to perform weekly or bi-weekly risk management meeting with notes and conclusions to follow. This type of meeting is a standard in ISO 27001 certified companies.

Lastly, Beawre is exploiting the application of its commercial technology in other domains where the same system applied, these are:

- Continuous risk control in the airline service space, such that the IoT equipped ground equipment is risk evaluated in real time basis, providing accommodation of the service continuity. The solution has been proposed and it is among the winners of Singapore Airline App challenge. This is undergoing a further pilot negotiation for a pilot deployable in the main Singapore airport as of time of writing this deliverable.

- Continuous risk control of the business processes in the construction space, where the evidence collectors and the Beawre Risk Control are closing the gap between the ability to control the project schedule and impacting factors of the process execution. This was directly translated from the approach of managing DevOps cycle and it is currently recognized as an innovative and necessary approach in the process of digitization of the construction sector. Such approach and solution was appointed as a Top 10 winner in the CEMEX Start-up Challenge competition, which is the most important start-up competition in the world in the construction space and it is led by the industry itself.

Concluding, Beawre is determined to continue commercializing the project results in the mentioned spaces where ENACT Risk Management is a core technology which builds the commercial solutions.

4 Continuous Orchestration and Deployment of SIS

In this section we present the final version of GeneSIS (aka. the orchestration and deployment enabler). GeneSIS is a joint effort between WP2 and WP3, while WP2 focuses on all aspects from the specification to the enactment of a deployment, WP3 concentrates on the support for adapting a deployment. Therefore, more details about the latter can be found in D3.3.

4.1 Overview and Main Achievements

DevOps promotes an iterative and incremental approach enabling the continuous evolution of software systems. A key feature to enable such continuous evolution is to support the continuous deployment of the system. This includes ensuring availability and supporting the deployment of security mechanisms, which must evolve along with the smart IoT systems, continuously fixing security defects and dealing with new security threats. GeneSIS aims at supporting the automatic deployment and adaptation of SIS, including of security and privacy mechanisms, over IoT, Edge and Cloud infrastructure.

4.1.1 *Overall approach*

GeneSIS enables continuous orchestration and deployment of Smart IoT Systems throughout the IoT – Cloud continuum. Given a description of a deployment topology, GeneSIS deploys and configures the needed software components, by connecting to the hardware (or software) nodes. This topology, so called the *deployment model*, only prescribes what components must be deployed, how a single component can be deployed, and how they connect to each other. GeneSIS automatically derives how to deploy them. Therefore, GeneSIS is composed of two key components: *(i)* a domain specific modelling language for specifying deployment models, and *(ii)* an execution engine to enact the provisioning, deployment and adaptation of a SIS.

The target user groups of GeneSIS are mainly DevOps engineers, software developers, and architects. The GeneSIS modelling language has been conceived so the deployment model can act as a touch point for developers and operators. Both teams can use it to deploy either in development, staging or production environments. It is also worth noting that deployment model written using the GeneSIS modelling language is independent of the underlying technologies, that is, GeneSIS can deploy components anywhere in the IoT-Cloud continuum: from microcontrollers without direct Internet access, to virtual machines running in the Cloud.

The main task of the GeneSIS deployment engine is to reconcile two views of the system: the deployment model given by the user, and the current state of the running infrastructure. Software components may already be running on the infrastructure, say during a system upgrade for instance. To reconcile these two views, the GeneSIS deployment engine adheres to the “models@runtime” architectural pattern depicted in Figure 14. It compares these two views and deduces what changes the

adaptation engine must carry out on the running infrastructure to align it with the prescription, that is, with the deployment model given by the user.

4.1.2 *Main Achievements and Innovations*

As detailed in our different literature reviews [1-3], whilst many solutions exist for the continuous deployment of cloud-based systems, very little effort has been spent on providing solutions tailored to the delivery and deployment of applications across the whole IoT, Edge, and Cloud space. Cloud and Edge solutions typically lack languages and abstractions *(i)* that can be used to support the orchestration of software services and their deployment on heterogeneous IoT devices possibly with limited or no direct access to the Internet, *(ii)* supporting the deployment of security mechanisms, and *(iii)* with the platform independent high availability mechanisms thereby avoiding vendor lock-in.

To address these limitations, GeneSIS' main contributions are the following:

- GeneSIS offers a single domain-specific language that accommodates both design-time and run-time activities. At design-time, developers identify the software component, the constraints those components place on the underlying hardware, and how they should be deployed. At runtime, operators adjust the deployment by changing the deployment models, for instance to upgrade a given component to a newer version.
- GeneSIS provides the necessary abstractions to operate over the whole Cloud / IoT continuum: From IoT microcontrollers to Cloud virtualized servers including gateways at the Edge. GeneSIS can therefore also operate devices with no or limited connectivity.
- GeneSIS provides the platform-independent or specific mechanisms and abstractions to enhance availability and automatically inject well-documented fault-tolerance patterns, such as replication, watchdogs, and rolling upgrades. Together, they improve tolerance to both internal faults and scheduled outages.
- GeneSIS provides means to specify security requirements, for the continuous deployment of security mechanisms, and can enforce encrypted communications between components.
- GeneSIS provides fully declarative adaptations: It computes what must be changed on the running infrastructure to transition from the current state to the new deployment given by the user.
- GeneSIS maintains the deployment model in synchrony with the running infrastructure. Therefore, the deployment model is a living documentation that does not get out-of-date, and whose maintenance cost is minimized.
- GeneSIS offers dedicated APIs for 3rd third parties to trigger an adaptation and manipulate a model.

Table 2. Main achievements since M22.

Feature	Description	Status at M22	Status at M33
DevOps features			
Specify deployment model of the overall SIS over Cloud, Edge, IoT resources	See Section 4.2. Extended with new concepts for security and rolling deployment.	Completed	Revised
Check validity of deployment model	See D3.3	Ongoing	Completed
Specify hardware specificities and requirement to ensure compliance of the deployment with respect to these specificities	See Section 4.2.	Completed	Completed
Provisioning and automatic deployment over Cloud, Edge, IoT resources and the	See D3.3 and Section 4.2.2	Completed	Completed

orchestration of the deployed software instances—*i.e.*, create VMs in the cloud, deploy using Docker, SSH, Ansible, update firmware, and deploy on resources with limited access to Internet			
Automatically adapt a deployment in a declarative way	See D3.3	Completed	Completed
Same language used for design and runtime activities – *i.e.*, runtime information is reflected in the language	See D3.3	Completed	Completed
Trustworthiness features			
Security and Privacy: Specify requirements in term of security and privacy and support the deployment of the selected corresponding mechanisms	See Section 4.2. Extended with new concepts for security policy deployment	Completed	Completed and extended
Security and Privacy: Specific GeneSIS components for the deployment of security and privacy mechanisms (at least the ENACT enablers)	See Section 4.4	Ongoing	Completed
Security and Privacy: Support automatic encrypted communication between deployed components without any change in the component code.	See Section 4.4.3	Planned	Completed
Provide a means to deploy actuation conflict manager and include the concept in the language	See Section 4.2.	Completed	Completed
Support the automatic blue/green deployment of Cloud and Edge resources	See Section 4.3	Planned	Completed

4.1.3 *Improvements over D2.2*

Compared to the initial release of the enabler in D2.2, the core evolutions of GeneSIS can be summarised as follows:

- Deployment of dedicated mechanisms to improve availability in the presence of internal faults or scheduled outages. These mechanisms either rely on pre-existing mechanisms from the underlying platform or specific ad hoc implementation provided by GeneSIS.
- The deployment of encryption to secure the communications between components. When requested, GeneSIS injects proxies that encrypt communication in order to improve confidentiality.
- The support for the deployment of security mechanisms has been extended, providing means to integrate third party security solutions, or to inject ad hoc security policies in software component (in the form of software dependencies) during deployment, in particular in the context of IoT platforms (evaluated on the SMOOL implementation of the SOFIA IoT middleware).
- Extended logging capacity for ThingML programs. ThingML now includes a novel logging framework, based upon a binary encoding of events (transition, variable assignments, function calls), which reduces runtime overhead, compared to our previous approach based on string concatenation.

In the following sections we shortly recall the core features of GeneSIS (*i.e.*, the GeneSIS Modelling language and its supporting execution engine), highlighting their evolution, before we dig into the details of the newly delivered features.

4.2 GeneSIS and Latest Support for New Features

4.2.1 *The GeneSIS modelling language*

In the following we present the core concepts of the GeneSIS modelling language, including changes compared to D2.2. More detailed description of the language including examples of the textual syntax can be found in [4].

The GeneSIS modelling language follows a component-based approach in order to facilitate separation of concerns and reusability. Deployment models can be regarded as assemblies of components. It implements the type-instance pattern [5] to facilitate the definition and reuse of generic type of components. As a result, components can remain device- and platform-independent or specialized into device- or platform-specific components.

To deploy an application on the selected target environment, its application components need to be allocated on host services and infrastructure. More precisely, what needs to be allocated is the implementations of those components. This is often referred as deployable artefact [6]. Some examples of deployable artefacts are binaries, scripts, etc. A deployable artefact can be physically allocated independently to multiple hosts (*e.g.*, a JAR file can be uploaded and executed on different Java runtime environments). Where and how these deployable artefacts are allocated is specified in a deployment model. Deployment approaches typically rely on the logical concept of software artefacts or components [7]. A deployment model is thus a connected graph that describes software components along with targets and relationships between them from a structural perspective [6]. GeneSIS includes a domain-specific modelling language to specify deployment model—*i.e.*, the orchestration and deployment of Smart IoT Systems (SIS) across the IoT, Edge, and Cloud spaces.

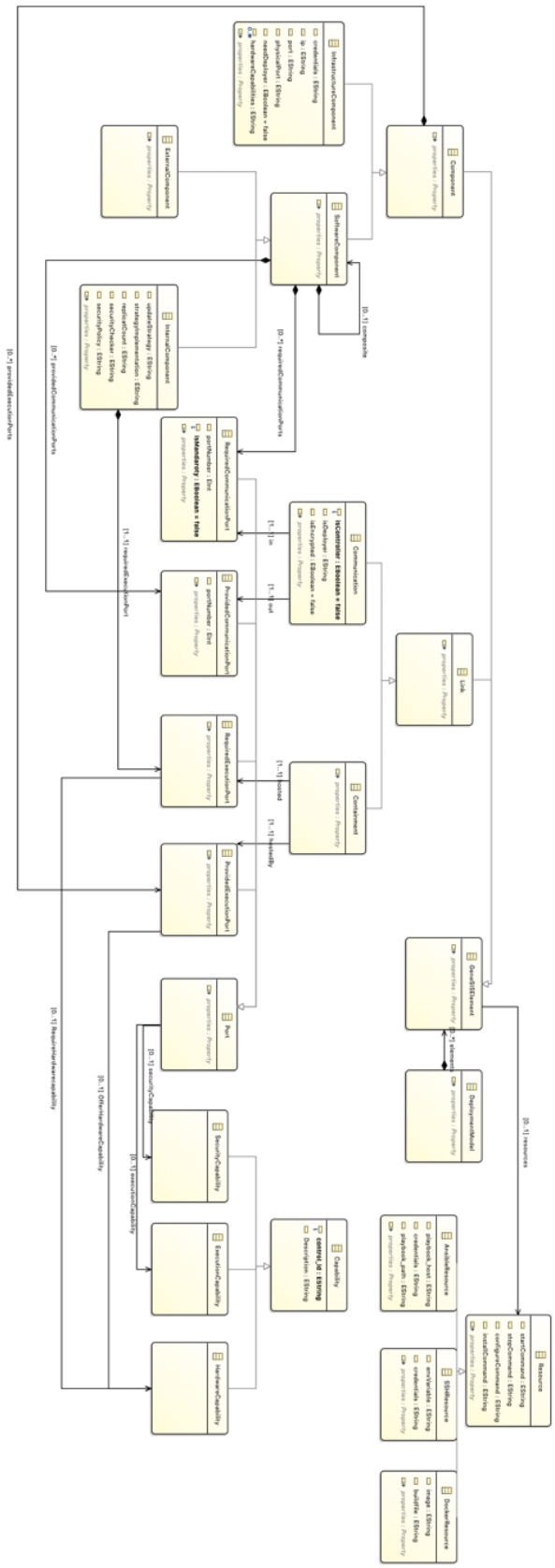


Figure 13. Excerpt of the GeneSIS metamodel

A *DeploymentModel* consists of *GeneSISElements*. Each *GeneSISElement* inherits from *NamedElement* and thus has a unique name. In addition, they can all be associated with a list of properties in the form of key-value pairs. The two main types of *GeneSISElements* are *Components* and *Links*.

A *Component* can be a *SoftwareComponent* representing a piece of software to be deployed on a host (*e.g.*, the *Temp&HumiditySensorReader* Arduino sketch to be deployed on *Arduino-1*). A *SoftwareComponent* can be an *InternalComponent* meaning that it is managed by GeneSIS (*e.g.*, the instance of MQTT to be deployed on RPI-3), or an *ExternalComponent* meaning that it is either managed by an external provider (*e.g.*, the SMOOL IoT middleware offered as a service) or hosted on a blackbox device (*e.g.*, Z-Wave transceiver). A *SoftwareComponent* can be associated with *Resources* (*e.g.*, scripts, configuration files) adopted to manage its deployment life-cycle (*i.e.*, download, configure, install, start, and stop). Finally, the property *isController* depicts that the *SoftwareComponent* associated to the attribute is controlled by the other.

An *InfrastructureComponent* provides hosting facilities (*i.e.*, it provides an execution environment) to *SoftwareComponents*. The property *needDeployer* depicts that a local connection is required to deploy a *SoftwareComponent* on an *InfrastructureComponent* via a *PhysicalPort* (*e.g.*, the Arduino board can only be accessed locally via serial port). This property is typically used for devices with no direct access to Internet, which can only be reached indirectly via other devices (*e.g.*, a Raspberry Pi gateway), configured by GeneSIS.

There are two main types of *Links*: *Hostings* and *Communications*. A *Hosting* depicts that an *InternalComponent* will execute on a specific host. This host can be any component, meaning that it is possible to describe the whole software stack required to run an *InternalComponent*. A *Communication* represents a communication binding between two *SoftwareComponents*. Finally, the property *isDeployer* specifies that the *InternalComponent* (one of the endpoint of the *Communication*) hosted on an *InfrastructureComponent* with the *needDeployer* property should be deployed from the host of the other *SoftwareComponent* (the other endpoint of the *Communication*) (*e.g.*, the artefact to be executed on Arduino-1 will be deployed from the RPI-2). This property is important as several hosts may have a local access to the host with limited Internet access but only one should run the deployment agent. The property *isLocal* indicates that the source and target of the communication have to be deployed on the same host.

The concept of *Capability* can be used to specify that a component provides or requires a specific feature. Capabilities are used to validate that one component is fulfilling the requirements from another one. A *Capability* is defined by a *description* and a *controlId*, which is a unique identifier for different type of capabilities. *Capabilities* are attached to *Ports*. A required *Port* may require *SecurityCapabilities* (*i.e.*, the SmartEnergyApplication can only be accessed with proper authorizations and *ExecutionCapabilities* (*i.e.*, a specific execution environment or a feature is required for the component to execute). By contrast, a provided *Port* may offer *SecurityCapabilities* and *ExecutionCapabilities*. The ports attached to an *InfrastructureComponent* can expose a set of *HardwareCapabilities*, which represents the interfaces toward specific hardware facilities attached to the component (*i.e.*, the temperature and humidity sensors attached to Arduino-1 are accessible via I2C). For a deployment model to be valid, all the required capabilities must match a provided capability. More details about the validation of a GeneSIS deployment model, please refer to D3.3, Section 5.

Compared to D2.2, the following changes have been introduced into the language as a mean to specify: *(i)* a deployment strategy (regular or applying rolling deployment pattern), *(ii)* Security policies to be deployed, and *(iii)* specifying that a communication should be encrypted. In the following we detail these changes.

The deployment strategy to be applied when deploying an *InternalComponent* can be specified using the attribute *updateStrategy*. At the time of writing, two strategies are available: normal (regular deployment) and Blue/green. How the blue/green deployment will be enacted is defined *(i)* by a *strategyImplementation*, which indicates if the deployment should leverage Docker swarm or the GeneSIS ad-hoc blue/green deployment support (see Section 4.3.2); and *(ii)* a *replicaCount,* which indicates how many instances of the *InternalComponent* should be instantiated in parallel.

There are two ways to specify security policies, the first approach assumes the component to be deployed already includes a set of predefined security policies. In such case, the *securityPolicy* attribute of the *InternalComponent* can be used to specify the desired policy. The second approach consist in the injection of some security code into a software component. The code being integrated into the source code of the deployable artefact via ThingML. This code can be attached to an *InternalComponent* using the *securityChecker* attribute. It is worth noting these two attributes only applies to *InternalComponents* as the underlying GeneSIS deployment mechanism may need to inject code and compile the deployable artefact. Finally, a Communication can be attached with the *isEncrypted* attribute representing that the communication between two *InternalComponents* should be encrypted.

Two concrete syntaxes can be used to specify GeneSIS deployment models: *(i)* a JSON-based textual syntax (more details can be found in [4]) and *(ii)* a graphical syntax (see Figure 34 in Section 4.6.3 as an example).

The GeneSIS modelling language is extensible. Subtypes of *SoftwareComponents* can be loaded in the form of plugins, to support user-defined elements for deployment. These plugins can be dynamically loaded in the supporting execution engine. More details about the plugin mechanism can be found in Section 5 of D3.3.

4.2.2 *The GeneSIS deployment engine*

From a deployment model specified using the GeneSIS modelling language, the GeneSIS deployment engine is responsible for enacting the deployment of the SIS. The overall architecture of this engine is depicted in Figure 14.

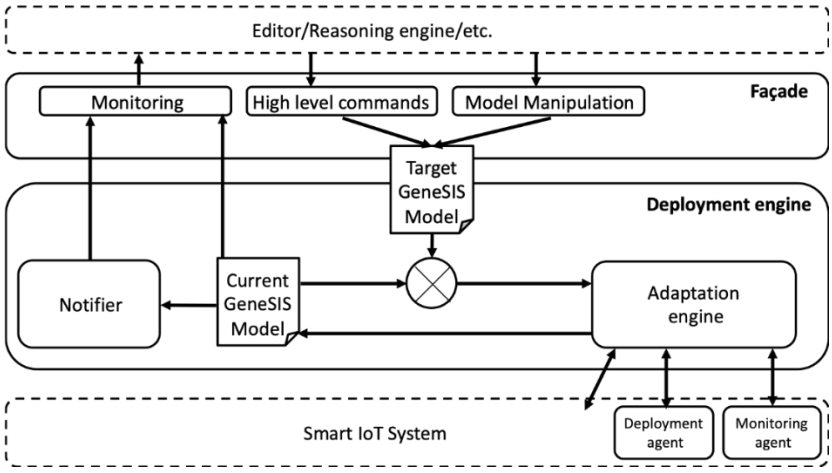


Figure 14 The Models@runtime architecture as implemented by GeneSIS

The top part depicts the views of the user, that is the system as it should be deployed. This view is prescriptive, in the sense that it is a requirement placed upon the running system. By contrast, the bottom layer depicts the current state of the running system. This view is descriptive since it only results from monitoring information. The heart of GeneSIS (and of the models@runtime architecture) is the "deployment engine", which compares these two views and deduces what changes the adaptation engine must carry out on the running infrastructure to align it with the prescription, that is, with the deployment model given by the user.

The deployment engine can delegate part of its activities to deployment agents running on the field. It is not always possible for the deployment engine to directly deploy software on all hosts. For instances, tiny devices do not always have direct access to the Internet or even the necessary facilities for remote access (in such case, the access to the Internet is typically granted via a gateway) and for specific reasons (*e.g.*, security) the deployment of software components can only be performed via a local connection (*e.g.*, a physical connection via a serial port). In such case, the actual action of deploying the software on the device has to be delegated to the gateway locally connected to the device. The deployment agent

aims at addressing this issue. It is generated dynamically by GeneSIS based on the artefact to be deployed and its target host.

The design as well as the mechanisms to generate of deployment agent have not changed since D2.2, we thus refer the reader to this deliverable or to [4, 8, 9] for more details. Similarly, for more details about the GeneSIS deployment engine and its adaptation capabilities, we refer the reader to D3.3. In the following sections we detail the core features introduced in GeneSIS that being reported in this D2.3.

4.3 Deploying Availability Mechanisms

4.3.1 *Motivations*

Availability refers to "*the ability of the system to mask or repair faults such as the cumulative service outage period does not exceed a required value over a specified time interval"* [10] (Bass et al. 2012, p. 174). Availability is a primary concern for business stakeholders because service interruption often translates into money loss. The failure of an electricity meter for instance may very well affect the capacity of the electricity company to properly bill its customers.

Availability, as any extra-functional requirements, does not affect the system function, but rather affects its architecture. Building high-availability systems requires additional components to detect, repair, or even prevent faults, such as monitors, watchdogs, replicas, or voting mechanisms to name a few. Availability tactics are now well documented, so we refer the reader to [10] (Bass 2012, Chap. 5) for an introduction.

Many things can go wrong in Smart IoT Systems, including incorrect algorithms, network failure and delays, hardware failure, etc. In the following, we focus on scheduled outages, which are interruptions of service needed because of software upgrade, and internal faults, which are faults that occur because of defects in the source code of the components.

We extended GeneSIS with the ability to deploy mechanisms that cope with these two kinds of fault. To mask internal faults, GeneSIS deploys multiple instances of the same service/component (so called replicas) behind a proxy. When one replica fails, the proxy can query another replica. To mask scheduled outages, GeneSIS provides zero-downtime upgrades. We leverage the same architecture and deploys the new version (behind the proxy) before to decommission the older one. That way, there is always a replica available and upgrades do not affect availability.

4.3.2 *Platform-agnostic Availability Strategies*

Modern execution platforms such as Docker or AWS already implement various availability mechanisms, and, they have strategies for both scheduled outages and fault-tolerance. Docker Swarm for instance performs zero-downtime upgrades by deploying new services instances before to decommission the older ones. The challenge is that, from a deployment perspective, the availability tactics are tightly coupled to the underlying execution platform. Changing platform requires changing the deployment configuration.

To decouple availability from deployment platform, GeneSIS captures these deployment tactics independently of the underlying platform. If the platform already provides mechanisms (such as Docker Swarm), GeneSIS uses those, otherwise it deploys built-in components to implement the selected tactics. In the following we illustrate three scenarios that show how GeneSIS copes with scheduled outages and internal faults.

- **Scenario 1: "Initial deployment"**. GeneSIS deploys the system following the availability tactics selected by the user.
- **Scenario 2: "Internal Fault".** A fault occurs in the system and we explain how the mechanisms that GeneSIS has deployed deal with that fault, so that it is not visible to the end-user.

- **Scenario 3: "Zero-downtime upgrade".** The user requests the deployment of a new version of the system and we illustrate how GeneSIS leverage the underlying mechanisms to minimize service disruption.

4.3.2.1 Using built-in components on top of Docker

By default, GeneSIS does not make any assumption of the capability of the platform where it should install a component. It could be a very fully featured platform such as Docker Swarm (see Section 4.3.2.2) or simply an operating system offering remote access (through SSH, Telnet, etc.). We detail here the later case, that is when the host is a bare OS.

Recall that GeneSIS uses two strategies to improve availability: Replication to deal with internal faults, and zero-downtime deployment to deal with scheduled outages. To implement these two strategies, we need three capabilities that are provided by additional components:

- *Routing,* that is, the capability of redirecting incoming traffic to a selected replica. Network proxies provide this and in GeneSIS, we selected Nginx.
- *Error detection*, that is the capability to proactively detect replicas that have failed (for whatever reasons). We used a watchdog, that is a component that periodically connects to the replicas and runs a so-called "health check". The health check is an application specific behaviour that confirms that the replicas is up and running. It could be requesting a predefined resource using HTTP, checking the status or OS-level services, or any other "quick-check". In GeneSIS, we have implemented simple watchdogs using Shell scripts and CRON tasks.
- *Spatial isolation,* that is, the capability to deploy multiple instances of the same application with guarantees that they can access external resources (network port, files on disks, etc.) without stepping on each other. GeneSIS uses containers (*i.e.*, Docker in the current implementation) to ensure spatial isolation of replicas, but other container technologies such as LXC apply.

Scenario 1: Initial deployment. The first step is for GeneSIS to ensure that the underlying host offers "spatial isolation" guarantees. To do so, GeneSIS first install Docker as container offer such guarantees. Figure 15 below illustrates how GeneSIS interacts with the host to install docker and to create a "replicable image" of the software stack.

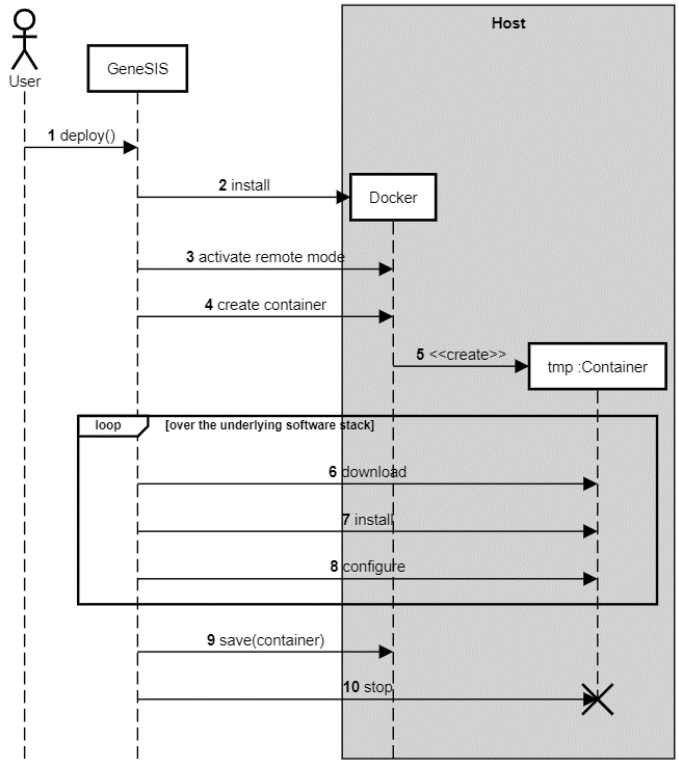


Figure 15: Converting SSH resources into Docker images

Given a component to deploy, GeneSIS first connects to the host through SSH and installs Docker (Step 1). Then GeneSIS configures Docker in remote mode so that other components (including itself) can access it through the network. Then, GeneSIS creates a new temporary container (by default, using the image "debian:10-slim") and installs the underlying software stack. To do this, GeneSIS traverses the underlying software stack and installs all underlying components by triggering the associated SSH commands into the container (Step 6, 7 and 8). Once the stack is installed and configured, GeneSIS converts it to a separate Docker image, that it later uses to install multiple replicas (Step 9). Finally, GeneSIS destroy the temporary container. At this stage, GeneSIS has enforced spatial isolation, and can then proceeds with replication and zero-downtime upgrades.

Once Docker is operational and that the software to install is available as a Docker image, GeneSIS proceeds with the two remaining capabilities, namely detecting errors and routing as shown on Figure 16. First GeneSIS installs the proxy component through Docker (Step 1 and 2). Then, GeneSIS installs the watchdog and configures it with the endpoints of the Docker host, the proxy and with the number of replicas to maintain (Step 2 and 3). For each missing replica, the watchdog requests Docker to provision a new instance of the image built in Scenario 1 and then the watchdog start checking the health of each replica periodically (Step 6 and 7). As soon as a replica is detected as healthy, the watchdog registers it to the proxy (Step 8), which uses it process user requests (Step 9 and 10).

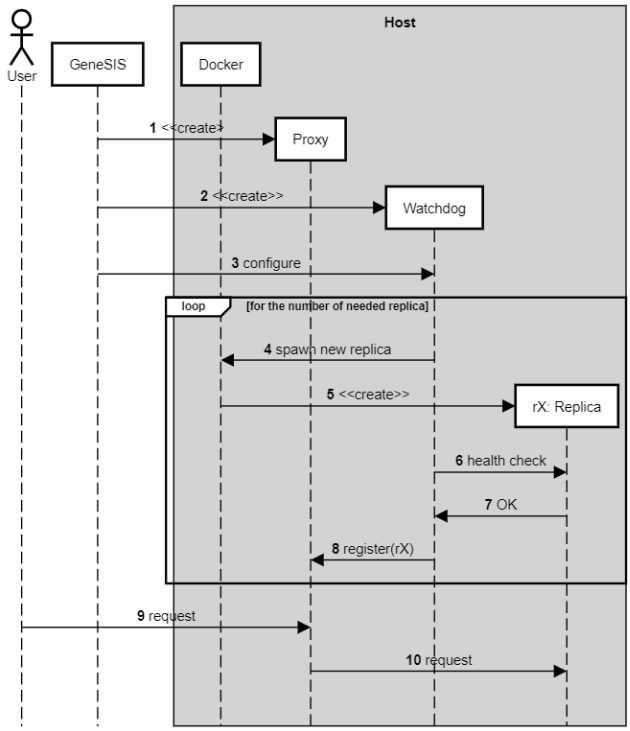


Figure 16 Configuring watchdogs and proxy to improve availability

Scenario 2: Fault tolerance. We now turn to the second scenario where one replica fails and we explain on Figure 17 how the watchdog detects and reacts to such a failure. The main mechanism to detect failure is the health check. Since a health check is an application-dependent behaviour, the user must provide it as a script to be executed periodically by the watchdog. GeneSIS defines the interface of the health check script as follows. The health check must accept the endpoint of the replica to query as its sole input parameter and must output the replica status in return through its exit code: Zero if the replica is healthy and any other value otherwise. This gives the user the capability to integrate any application-specific health check logic. The listing below shows one such health check script based on the HTTP status code, returned by a service.

```
#!/bin/bash
ENDPOINT="${1}"
response=$(curl --write-out '%{http_code}' --silent --output /dev/null
"${ENDPOINT}")
if [ "${response}" != 200 ]
then
    exit 1
fi
```

As mentioned about Figure 16, the watchdog periodically triggers the user given health check script to detect failures of replica. Should the health check fail or timeout, the watchdog considers the replica faulty and immediately informs the proxy, which then switches replica (Steps 3 and 4). Meanwhile, the watchdog spawns a new instance of the image built during Scenario 1 (Steps 5 and 6) and stops the faulty replicas (Step 7). As soon as the new replica is operational, the watchdog registers it to the proxy (Steps 8, 9 and 10), which can use it to process user requests (Steps 11 and 12).

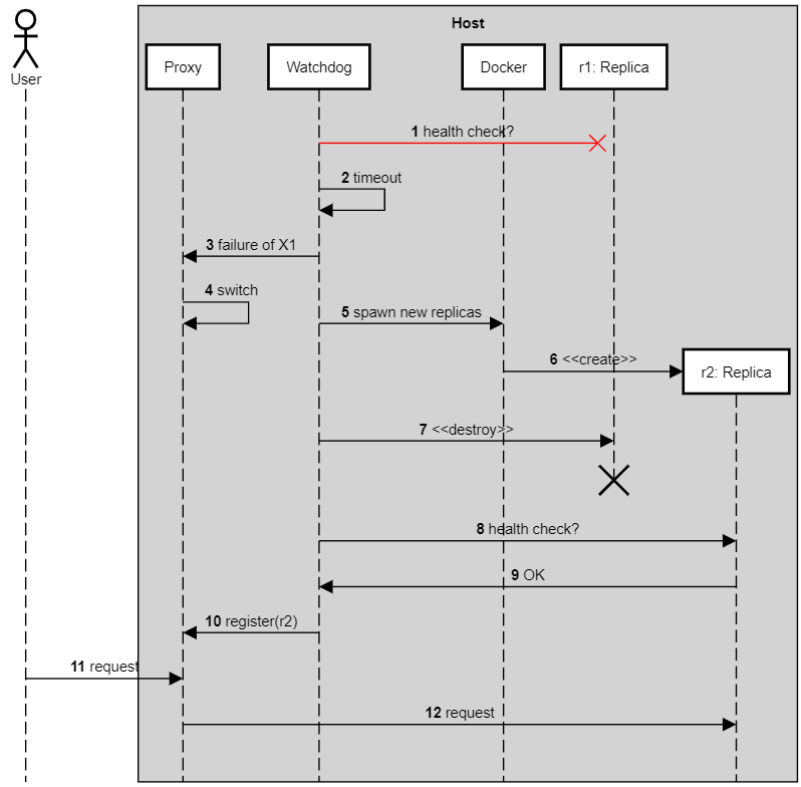


Figure 17 Masking failures of replicas to improve availability

Note that this architecture can only detect replicas' failure as fast as the watchdog waits between two health checks. Besides, for the failure to be invisible to the user, there must at least two replicas, for the proxy to switch between them as soon as a delegation fails. Finally, transient phenomena such as network delays may be mistaken for replica failures and lead to unnecessary starts and stops of the container.

Scenario 3: Zero-downtime redeployment. Finally, GeneSIS leverages these proxy and watchdog to guarantee zero-downtime upgrades, as shown on Figure 18.

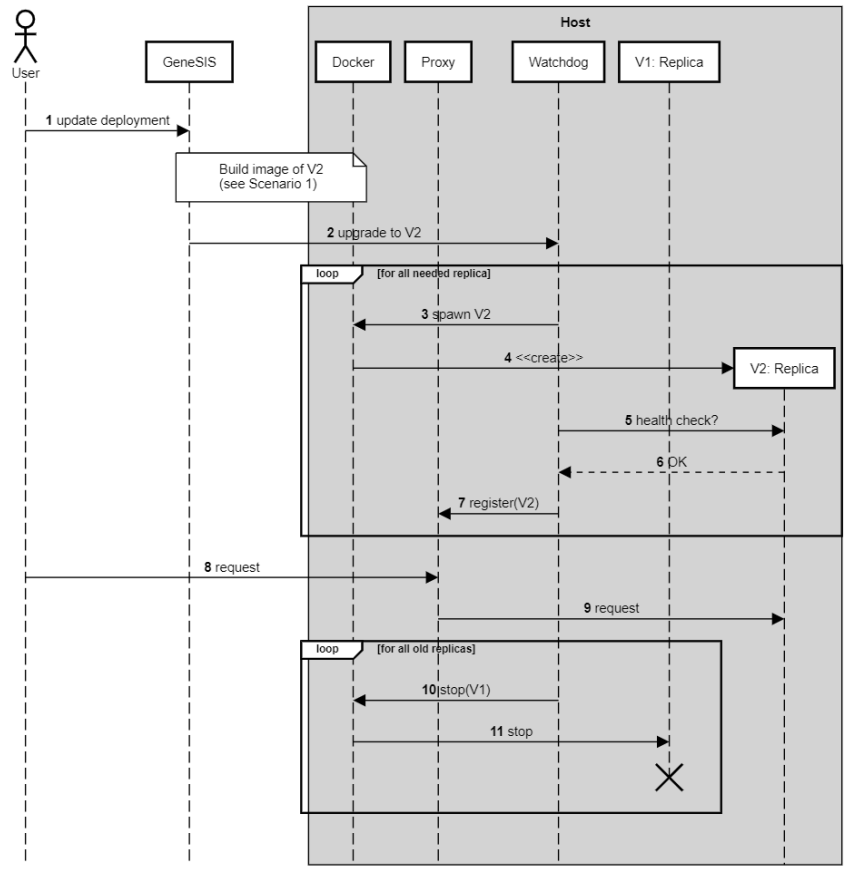


Figure 18 Using proxy and watchdog to guarantee zero-downtime upgrades

When the user requests an upgrade, that is the deployment of a new version, GeneSIS first builds a new docker image of the software stack, including this new version. We described this process in Figure 15. Once this new image is ready, GeneSIS request the watchdog to perform the upgrade (Step 2). The watchdog thus provisions new instance of the new version (Steps 3 and 4) and, once these new replicas are operational, the watchdog registers them to the proxy (Steps 4, 5, 6 and 7). At that stage, incoming requests from the user are still delegated to the older version (Steps 8 and 9). Only once all replicas of the new version is operational, then the watchdog starts to decommission the older versions (Steps 10 and 11).

4.3.2.2 Using Docker Swarm

In many cases, developers do not choose the platform on which their software runs: It may result from organizations' policies, customer requirements, etc. Platform such as Kubernetes, Docker Swarm or Rancher for instance all implement availability tactics, including replication and zero-downtime releases. Should the user use such platform, GeneSIS can exploit these native features to ensure fault-tolerance and zero-downtime upgrades. Docker swarm already implements routing among multiple replicas and fault detection. GeneSIS therefore delegates these features to Docker Swarm. We briefly example how GeneSIS handles our three scenarios using Docker Swarm.

Scenario 1: Initial deployment.

This step is the simplest as we assume here that the host already runs Docker Swarm and that it therefore already guarantees spatial isolation. Here, GeneSIS simply requests Docker Swarm to deploy the given number of replicas of a given Docker image.

Scenario 2: Fault tolerance

This is also fully transparent from the GeneSIS standpoint. When GeneSIS delegates the deployment to Docker Swarm it specifies the number of replicas and the health check script to be used to detect faults. It is docker swarm that periodically checks the replicas status, provisions new ones if some have failed and route incoming requests accordingly.

Scenario 3: Zero-downtime redeployment

Further Docker Swarm also offers various strategies to upgrade a service (*i.e.*, the set of replicas in Docker Swarm parlance). Among many options, Docker Swarm lets the user specify the "update order". If this order is "stop-first", then Docker Swarm first stops all the replicas of the older version, and only then starts provisioning replica for the new version. By contrast, if the update-order is "start-first", then Docker Swarm provisions all new replicas before to decommissions, as GeneSIS would do without Docker Swarm (see Figure 18). This "start-first" option let Docker Swarm minimize service interruptions.

4.3.3 *Limitations*

The support for availability tactics in GeneSIS is limited to Docker platform, although the general principle applies regardless of the underlying technology and other can extend GeneSIS and support other technologies. In addition, there are other types of faults that the current tactics cannot deal with. Hardware failure for instance would take down the whole host and therefore all the replicas at once. To tackle hardware failure, replication would have encompassed hardware, but this goes beyond GeneSIS whose mission is to provide platform agnostic deployment. Nevertheless, using the ENACT framework, the Root Cause Analysis enabler can be used to monitor and identify such failures and DevOps engineers can use GeneSIS to migrate the software components on a new host, benefiting from its platform independence.

Programming faults are also not dealt with. Because GeneSIS is oblivious to the inner working of the components it deploys, all replicas are similar and fail in a similar manner. For instance, if a defect in the code lead to a fault of one replica (say because of invalid user input), then all replicas will exhibit this fault. Only diversification techniques [11] (Baudry et al 2015) could help having replicas whose behaviours differ from one another, and that exhibit different failure profiles. D4.3 discussed some diversification mechanism for IoT systems, in the architecture and communication levels, and the DivEnact tool to manage the deployment of diverse software on the fleet of devices.

Improving availability from a pure deployment perspective, as GeneSIS provides, is bound to stateless component that can be easily replicated. Otherwise, if components have persistent state, the application logic must ensure the proper consistency (possibly eventual) of all replicas, and it is not possible to solely rely on generic mechanisms.

Finally, on an Edge platform there are resources that cannot be replicated and that would require further investigation. A serial link for instance cannot be shared between replicas and, in this case, dedicated, application-specific logic must be in place to ensure consistent behaviour between all the replicas.

4.4 Continuous Enhancement of Security Controls

GeneSIS empowers a DevOps team to cope with security and privacy concerns of SIS as it natively offers support for including, as part of the deployment models, concepts to express security and privacy requirements and for the automatic deployment of the associated security mechanisms [8]. More importantly, GeneSIS enables the continuous enhancement of security controls in a DevSecOps cycle

to keep security mechanisms up-to-date and well-aligned with the evolution of SIS, as well as addressing IoT security risks that are always evolving.

In D2.2, we presented how GeneSIS can support for the automatic deployment of the default security mechanisms of the existing IoT platforms such as SMOOL [12]. In this deliverable D2.3, we present the latest development of GeneSIS for better supporting the continuous enhancement of security controls that can refine or override existing the associated (default) security mechanisms of the existing IoT platforms such as SMOOL and extend them with other (third-party) security mechanisms to provide enhanced security controls in a DevSecOps fashion.

We first present in Section 4.4.1 the latest development of GeneSIS for better supporting the specification and deployment of security components. Then, we elaborate on how GeneSIS can enable the continuous enhancement of security controls in Section 4.4.2. Based on the motivating example in Section 2, we demonstrate how to use GeneSIS to extend the associated (default) security mechanisms of the existing IoT platforms such as SMOOL with other (third-party) security mechanisms to provide enhanced security controls in a DevSecOps cycle (see Section 4.6.3 for more details). Moreover, the latest version of GeneSIS also provides support for easily enabling secure (encrypted) communications between components from the deployment model to operation (Section 4.4.3).

4.4.1 *GeneSIS for the specification and deployment of security components*

To better support DevSecOps, GeneSIS promotes specifying security mechanisms as explicit elements in the deployment model, instead of hidden (and thus tightly coupled) in the source code, so that developers can see and change the security mechanisms on the deployment model level. This includes specifying security requirements and capabilities, and supporting the deployment of security mechanisms as components reusable in different scenarios. Compared to the previous Genesis version reported in D2.2, we have first built a library of off-the-shelf security components that can be selected for instantiation in the deployment model. More importantly, we provide DevOps teams with mechanisms to configure security components and inject fined grained security policies into *InternalComponents* (without modifying their business logic), which in turn enables their seamless integration with third party security mechanisms (services, libraries, etc.). In particular, these mechanisms have been applied and demonstrated on the deployment of IoT platforms components (SMOOL KPs). These supports can ease the development, integration, and deployment of SIS with continuously enhanced security mechanisms.

In general, GeneSIS supports the deployment of security components as any other software components. Their deployment and configuration are managed using *Resources*, meaning they can be configured via exposed APIs and configuration files. The latest version of GeneSIS offers a library of off-the-shelf security components that can be selected for instantiation in the deployment model. For example, built-in cryptography components can be selected and configured to provide secure communication between IoT components using SSL/TLS (Section 4.4.3). Another example is the security API gateway presented in our motivating example, which is configured to secure the access to the smart applications. A security component to be deployed together with an IoT application can be declared in GeneSIS with *SecurityCapabilities* in a provided port. A required port of a *SoftwareComponent* that requires a matching *SecurityCapabilities* can be bound with the provided port of the security component that provides such *SecurityCapabilities*. Before enacting a deployment, the GeneSIS deployment engine validates the correctness of the provided deployment model. In particular, it ensures that all the required *SecurityCapabilities* match a provided *SecurityCapability*.

SIS are typically built on top of IoT platforms such as SMOOL [12] or FIWARE [13], which often act as an intermediary for the communications between the things within the IoT environment. They provide a proper ground for building mechanisms, in different applications and scenarios, to control and monitor these communications. GeneSIS first offers an approach to ease the integration of different security mechanisms with such IoT platforms, including the tool support for specifying reusable security components for IoT platforms.

GeneSIS provides a generic sub-type of *InternalComponent* for specifying the deployment of security mechanisms and policies built on top of such IoT platforms. We present here its application to the SMOOL IoT platform, which is used in our motivating example. Similar approach can be applied to other IoT platform. SMOOL is a SOFIA-based open-source middleware for IoT smart spaces, developed and maintained by Tecnalia. This platform consists of a server (Semantic Information Broker - SIB) and tools for creating clients (or Knowledge Processors - KPs) in Java, C++ and other languages. The created clients can communicate by using different protocols such as TCP, Bluetooth or WebSockets. The structure of the messages exchanged among clients follows the SMOOL OWL ontology. The SMOOL ontology embeds the necessary concepts to attach metadata to message payloads. SMOOL applications consist of a set of KPs that exchange data via a SIB. SMOOL comes with a KP generation wizard. The wizard can be used to generate a KP project in the Eclipse IDE, which includes all the code necessary to interact with a SIB. Developers can then add the application specific logic to the KPs according to the functional requirements of smart applications [12].

At the development phase (as well as at the deployment phase presented below), GeneSIS provides the support to relieve developers from manually specifying and maintaining security monitoring and control mechanisms in the code of a SMOOL client (a KP). Instead, a developer can define its own SMOOL client, focusing on its business logic. We integrated the SMOOL KP wizard with ThingML. As a result, a single Eclipse IDE can be used to generate the code of a KP, which can then be directly used as part of a ThingML program. The proper Maven manifests are automatically created facilitating the building and release of the desired application. This means that the DevOps team can quickly develop the business logic of the SIS based on the SMOOL platform, including necessary security mechanisms. DevOps teams can define SMOOL clients that leverage built-in security properties to check and enforce security concepts on messages requiring security controls.

The SMOOL's default security enforcement can be done with the SMOOL clients built-in security metadata checker to verify messages exchanged among them. In cases where a deeper control is needed, a specialised security metadata checker can be included in SMOOL clients, with additional privileges to watch and process the security metadata in messages exchanged, in the same way it is done with business logic concepts such as sensed temperature or gas values. This provides a fine-grained control on critical messages that may have significant security impact in the IoT system such as orders to actuators. More precisely, a client code can conduct security checks based on policies to be fulfilled by ontology concepts by using any of these options: *(i)* the default security metadata checker (for minimal configuration), *(ii)* a custom security metadata checker implemented in the development phase (for full control of security), and *(iii)* a custom security metadata checker for integration with external security services. Whatever security options, GeneSIS provides support for easily configuring the security mechanisms and how they should be integrated and deployed with the SIS. Thanks to ThingML, GeneSIS provides advanced support for the three options. To support the first options, GeneSIS enable the DevOps team to specify explicitly the default security policy that must be enforced by the Security checker (see code sample below). In the case of our motivating example, we control the roller shutters (via Windows Blind Position Actuators) to maximize the exploitation of daylights. Every actuation order must be accompanied by a valid security key before the actuation order can be executed as depicted below.

```
{
   "_type": "/internal/SMOOLSecurity",
   "name": "SecurityEnforcer",
    ...
    "security_Policy": [["BlindPositionActuator", "Authorization"]],
    ...
}
```

To support the second option, where the DevOps team can implement its own ad-hoc security checker, GeneSIS provides the means to automatically inject this security checker into the code of the *InternalComponent* to be deployed and to rebuild the component automatically. More precisely, when deploying this *SecurityEnforcer* component, the GeneSIS deployment engine injects the security policy into the ThingML code of the SMOOL client. This code injection is done before GeneSIS triggers the

compilation of this code to generate the actual implementation of the SMOOL client with the corresponding security policy.

To support the third option, GeneSIS not only inject the security checker code that integrates with a third party security solution (*e.g.*, Casbin [14] or the Context-aware Access Control mechanism in D4.3, or a "gatekeeper" in [15]) but it can also deploys the latter. We will elaborate more on these points in Section 4.4.2.

At the deployment phase, a SMOOL KP can be deployed by GeneSIS as any other software components. Once the SMOOL client has been developed, the developer can specify how to deploy it also indicating the security and monitoring mechanisms that should apply to its SMOOL client. GeneSIS will then inject within the SMOOL client the necessary code to perform the security checks before actually deploying it. To do so, we created a generic security component as a subtype of *InternalComponent* that represents a SMOOL client as a deployable artefact. This client can follow any of the security check options discussed above and is implemented with ThingML code, which integrates *(i)* the necessary SMOOL libraries, *(ii)* the SMOOL client business logic, *(iii)* and the security logic (see D4.3 for more details about how KP can be used to monitor and control security).

The main rationale behind this choice is the following. ThingML offers an extra abstraction layer that provides the ability to wrap the code and dependencies that compose a SMOOL client and to inject into it the necessary security code. In addition, it provides GeneSIS with a standard and platform-independent procedure to generate, compile, configure, and deploy the implementation of the security mechanisms. A similar approach could be applied to other IoT platforms. In this way, GeneSIS allows DevOps teams to reconfigure and update security mechanisms by design, in line with the evolution of IoT applications and the development of security and privacy risks. In the next section, we present more details on the DevSecOps support.

4.4.2 *The DevSecOps Support for the Continuous Enhancement of Security Mechanisms*

SIS typically expose a broad attack surface and their security must not be an afterthought [16-18]. The ability to continuously evolve and adapt these systems to their new environment is decisive to ensure and increase their trustworthiness, quality, and user experience. This includes security mechanisms, which must evolve along with the SIS, continuously fixing security defects and dealing with new security threats [19-21]. Following the DevSecOps principles [22], there is an urgent need for supporting the continuous deployment of SIS, including security mechanisms, over IoT, Edge, and Cloud infrastructures [23]. The DevOps movement promotes an iterative and incremental approach enabling the continuous evolution of software systems. As an evolution of the DevOps movement, DevSecOps promotes security as an aspect that must be carefully considered in all the development and operation phases for the continuous evolution of (IoT) systems that are secure. In this section, we present how GeneSIS, with the specification and deployment features described in Section 4.4.1, can enable the continuous enhancement of security controls in a DevSecOps cycle: from development to operation. GeneSIS also supports the adaptation of the system having enhanced security mechanisms or updated security policies with minimal impact on the already delivered and under operation (see D3.3).

Our approach [4, 8, 9] for the continuous deployment of SIS with enhanced security mechanisms can serve the DevOps team in both adaptation and evolution of the SIS. First, GeneSIS supports for evolving SIS with updated security mechanisms according to a new development cycle. Second, GeneSIS supports for adapting security enforcement to improve how the IoT system operates securely. This DevSecOps support leverages the GeneSIS' necessary mechanisms, interfaces, and abstractions to dynamically adapt the deployment and configuration of a SIS as presented earlier. We elaborate more on the two kinds of DevSecOps support in the following paragraphs.

GeneSIS supports for evolving SIS with updated security mechanisms according to a new development cycle. In this line of adaptation, the SIS in operation is evolving with new business logic components or even new physical devices being added resulting in the need for enhancing security mechanisms

accordingly. In the smart home system, there are IoT applications (*e.g.*, *UserComfortApp*) that get access to sensors' data (*e.g.*, temperature) from the smart home to make decisions and send commands to control the actuators, *e.g.*, window blinds. The applications interact with the smart home devices and services via the SMOOL platform (Figure 19). GeneSIS can easily support for the deployment of components that are either built on top of the existing IoT platforms like SMOOL or are independent of any IoT platform because of its generic approach for specifying deployment components. However, to make GeneSIS even more useful in practice, we have developed GeneSIS to ease the integration of IoT platform-specific components (*e.g.*, SMOOL clients) and IoT platform-independent components (*e.g.*, third-party security mechanisms like Casbin presented below) from development to operation.

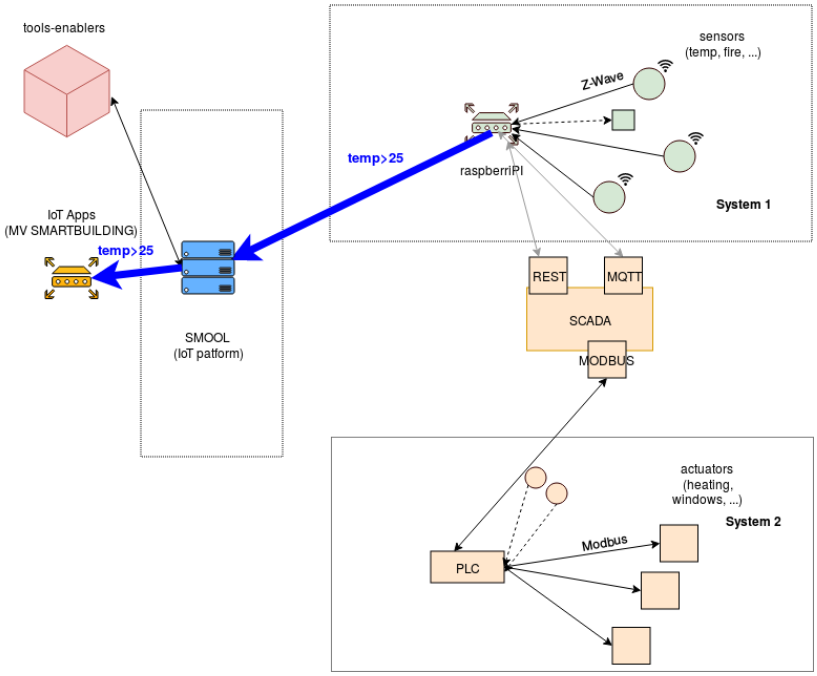


Figure 19. Tecnalia's Kubik smart building & SMOOL IoT platform

We present our approach based on GeneSIS and use the running example in Section 2 to demonstrate for the DevSecOps support by GeneSIS. In the initial version of the smart home system, there is the *UserComfortApp* application, which gets access to sensors' data to make decisions for energy efficiency and send commands to control the actuators, *e.g.*, window blinds. In particular, it maximizes the exploitation of daylights and regulates the in-door temperature whilst minimizing the energy consumption. If the room is bright because of daylight, it will switch off the LED-lights, and vice versa. On the other hand, if the room temperature is high, the application may need to close the window blinds to prevent sunlight heating the room. The *UserComfortApp* application interacts with the smart home devices via the SMOOL platform. There are two notable security mechanisms associated in this first version of the smart home (Section 2). The first one is a secure API gateway (Express Gateway [24]) that allows secure remote API access to the *UserComfortApp* application. The second one is a *SecurityEnforcer* by default of the SMOOL middleware that enforcing the security check for the data passing through, *e.g.*, only allowing genuine actuation commands to be sent to the actuators of the smart home.

The latest version of GeneSIS has provided a built-in support to ease the specification of the Express API Gateway in the deployment model. Adding a new instance of Express API Gateway is easy (see Section 4.6.3). The remaining work for the DevOps team is to specify the configuration files of the API gateway, and then configure how the API of the *UserComfortApp* application can be accessed. The first version of the configuration file *gateway.config.yml* defines how the securely exposed API that can be used for controlling remotely the *EnergyEfficiency* application.

In IoT platforms like SMOOL, there are often default security enforcements. For example, the actuation orders being sent to the physical actuators in System 2 of Figure 20 must be checked before they are actually sent to the actuators. This check makes sure only genuine actuation commands can be sent to the actuators. In other words, the SMOOL platform allows to check for actuation commands with valid security tokens. All the IoT apps must send actuation commands with valid security tokens.

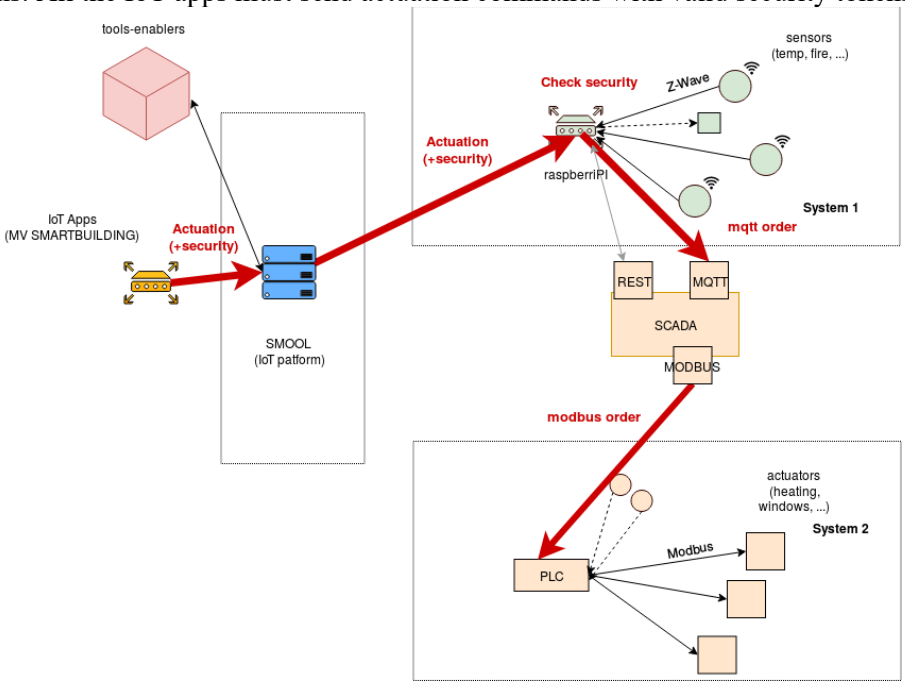


Figure 20. All actuation orders must be genuine!

However, during the evolution of the smart building system, new applications can be added, and new physical devices can also be added. New security requirements come up because the smart building system must control which apps can access which actuators. This means that more fine-grained security control must be introduced, which may not be available in the IoT platform. GeneSIS should support for seamlessly integrating new (third-party) security mechanisms into the IoT platforms.

In this new development cycle, not only that the secure API gateway must be updated with a new configuration file, but also the DevOps team needs to introduce a new security mechanism that can enhance the fine-grained control of how different applications can access to the sensors and actuators of the smart home system. GeneSIS has a generic support for seamlessly integrating and deploying any advanced security mechanism together with the IoT platform in use, *e.g.*, the SMOOL platform. More specifically, in this example, the DevOps team develop an access control mechanism, and then specify the integration point with the IoT platform in use (with GeneSIS support). During the deployment process, GeneSIS compiles the integration code before orchestrating the deployment of the integrated components. The DevOps team can choose to develop the required access control mechanism based on an open-source framework like Casbin [14], or the Context-Aware Access Control mechanism presented in D4.3, or any custom solution.

To enable such DevSecOps adaptation support, GeneSIS not only provides the modelling language embedded in a web UI for specifying the components of such IoT platforms, but also the reconfiguration and rebuild of these components (for integrating new security mechanisms with the IoT platform) before deployment (for adaptation or for a new development cycle). For example, in the SMOOL platform, each SMOOL producer or consumer is associated with a security checker for checking the security key of sensor data or actuation commands. GeneSIS allows updating the configuration of the security checker (*e.g.*, by injecting new configuration to overwrite the default one), and automatically rebuilding the SMOOL producer or consumer including the reconfigured security checker. By doing so, GeneSIS enables the DevOps team to make reconfiguration or redevelopment and redeployment easily for the evolution of SMOOL producers or consumers including security checkers. Figure 21 shows an example

of the GeneSIS's UI for extending the *SecurityChecker (in ENACTProducer)* to become a security enforcement point of the external access control service. Thanks to ThingML support within GeneSIS, the extended *SecurityChecker* is compiled in the *ENACTProducer* component for a new version of *ENACTProducer* to be deployed that works as a security enforcement point of the external access control service. More details about our experiments on the smart home system can be found in Section 4.6.3.

This approach is what we call the DevSecOps adaptation support for the co-evolution of business logic components and the security mechanisms. This means that when new business logic components require security mechanisms to evolve, GeneSIS can support for the adaptation, even including the integration of the IoT platform with other (third-party) security mechanisms. We present such an integration below for the SMOOL platform.

More advanced requirements may be also needed such as the constraints of the deployed security and privacy controls. Constraints for deploying security modules can be considered by GeneSIS such as how far the host of a security module is from the sensors or actuators. For example, the authorization module may need to be deployed and executed on a local node to the Building Control gateway or the gateways that host smart devices to ensure the performance of the authorization mechanism. The importance of having an authorization component close to the actuators is allowing actuation commands to be executed in a timely manner, without being hindered by the possible network delay. We have demonstrated this in Section 4.6.3.

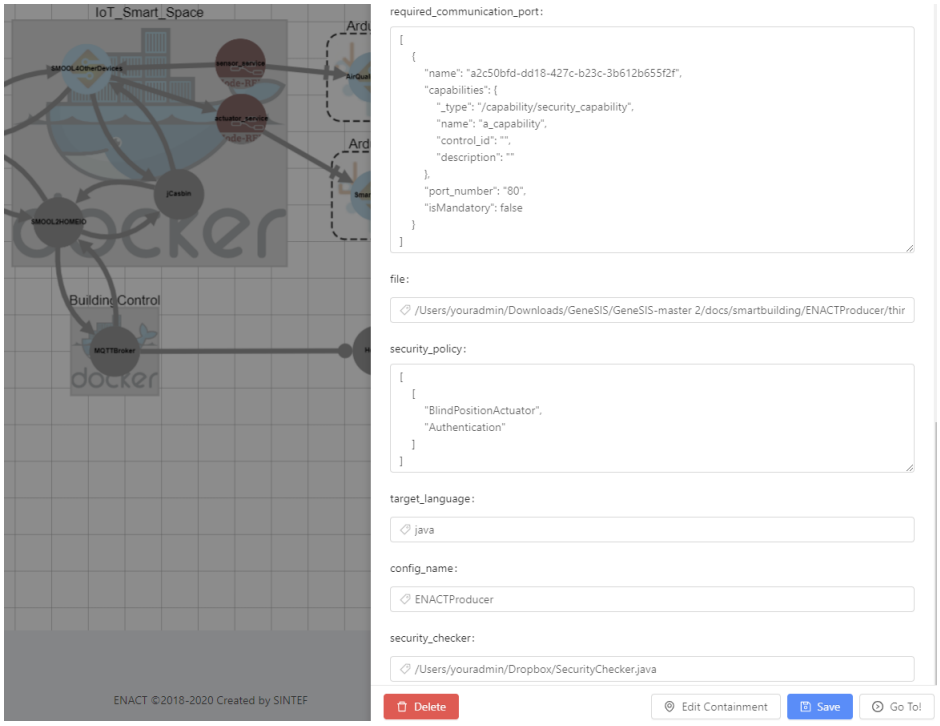


Figure 21. Extend the SecurityChecker.java as a security enforcement point of an external access control service

After the successful deployment, GeneSIS allows dynamic adaptations that can be triggered at any point, manually or automatically for adapting security enforcement to improve how the IoT system operates securely. In this line of adaptation, new security policies or configurations can be updated dynamically for the security mechanisms that are in operation. For example, the role-based access control policy can be easily updated according to new requirements. The trigger of such adaptation can be manually, but also can be automatically from a risk assessment process or after a reasoning process of actuation conflict management.

In summary, with the support from GeneSIS, the DevOps team can develop a new version of the smart building system together with enhanced security mechanisms according to its evolution. The

deployment of this new development cycle can be triggered manually from GeneSIS's GUI. After the successful deployment, GeneSIS also allows dynamic adaptations that can be triggered at any point, manually or automatically for adapting security enforcement to improve how the IoT system operates securely. In both ways presented so far, GeneSIS allows DevSecOps teams to reconfigure and update security mechanisms by design, in line with the evolution of IoT applications and the development of security and privacy risks. We have done experiments with our approach on the smart home system (Section 4.6.3). Besides, we have made a video demo using a smart building simulation called HomeIO. More details on the deployment demo using the HomeIO simulation can be found in this video[11].

Note that our approach for supporting DevSecOps with continuous enhancement of security mechanisms is generic. In this deliverable, we have showed the integration with Casbin because it is an open source well-known access control framework. In a similar way, GeneSIS can support for the integration of other security mechanisms, *e.g.*, the Context-Aware Access Control (CAAC) mechanism (see D4.3) into the deployment of the smart home/building scenario. In other words, a DepSecOps team can easily choose either Casbin component or a CAAC component from the drop-down list in the UI of GeneSIS and make corresponding configuration to use the mechanism and deploy it together with the whole SIS.

4.4.3 *GeneSIS for Enabling Secure Communications*

In SIS, an important security concern is to ensure confidentiality and integrity, for instance when sensors collect sensitive data, or when actuators operate critical hardware infrastructure. To this end, GeneSIS can inject additional software components to secure communications, as shown on Figure 22. GeneSIS equips both the client side and the server side with proxies. The one on the server side encrypts the communications and authenticates the client to the server, whereas the one the server side decrypts the communication.

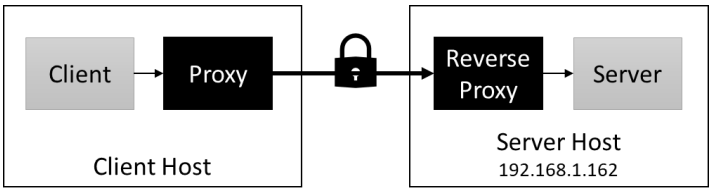


Figure 22 Securing communications by injecting proxies using GeneSIS

At the time of writing, the current implementation relies upon Docker to easily provision and connect client and server to their respective proxy, which are both instances of NGinx. The concepts however apply to other technologies. The communications between the two proxies are encrypted whereas the communications between the application components and their proxies are not, but these remains within the realm of Docker and are not exposed outside of the underlying Docker host.

Key Management. For communications to be secured, both client and server must have private and public keys. Both the client and server must disclose their public keys through certificates, which must be signed by a trusted 3rd party, so called certification authority (CA). By default, GeneSIS acts as a CA and signs the client and server certificates with its own private key, but existing CA can be used (and should be).

Server-side deployment. Consider the server component from Figure 22, which listens for HTTP requests on Port 5000, and which is given the hostname "app". GeneSIS first creates a dedicated Docker network on the host and deploys this server application in its own Docker container, connected to that network. GeneSIS then, generates a server private key and a server certificate, and deploys and configure an NGinx instance in a separate Docker container as shown by the code snippet below. Note that only this NGinx instance is accessible from the outside of the server host.

```
events {}
```

[11] https://youtu.be/yQ9XYWu-EZM

```
stream {
   server {
        listen 443 ssl;
        proxy_pass http://app:5000;
        ssl_certificate /etc/nginx/server.cer;
        ssl_certificate_key /etc/nginx/server.key;
        ssl_trusted_certificate /etc/nginx/ca-genesis.cer;

    }
}
```

The “proxy_pass” directive indicates the destination of the redirection. Here, it points to the container running the server application, which is exposed on the Docker network with the name “app”.

Client-side Deployment. On the client side, GeneSIS also creates a dedicated Docker network. It then generates a dedicated key and certificate for the client's authentication. GeneSIS then deploys another instance of NGinx, in a separate Docker container, configured as follows. Note that this proxy instance is not available outside of the client host, only from the internal Docker network.

```
events {}
stream {
   server {
      listen 5000;
      proxy_pass 192.168.1.162:443;
      proxy_ssl on;
      proxy_ssl_certificate /etc/nginx/client.cer;
      proxy_ssl_certificate_key /etc/nginx/client.key;
      proxy_ssl_trusted_certificate /etc/nginx/ca-genesis.cer;
   }
}
```

Here the “proxy_pass” directive points to IP address 192.168.1.162, that is to the host machine where the reverse proxy is running (so the port 443 dedicated to SSL communications). The “proxy_ssl” directive enables SSL/TLS for connections to a proxied server. The other directives indicate the client certificate, the client private key and the certificate of certification authority, respectively.

Finally, GeneSIS deploys the client application on the client host in a separate Docker container, and configures it so that it uses the local proxy instead of the application deployed on the server host.

4.5 Platform-independent Debugging of Resource Constrained Devices

We now turn to the issue of debugging SIS, and especially resource contrainted devices such as software components running on headless microcontrollers. We already addressed this issue in D2.2 where we described how we used model-driven engineering to inject logging instructions into ThingML programs. By contrast, we have now secured our ThingML logging framework using MQTTS and developed a more efficient alternative, based on binary encoding, which greatly improve its overhead. We refer the reader to Appendix C for comprehensive treatment.

The core of our approach is to reduce the effort needed to debug ThingML programs, using logging statement. Recall that ThingML programs are state-machines that react to and emit messages through well-defined ports. To improve logging, we first annotate ThingML elements, such as transitions, variables, functions and then use model-transformations to automatically get a new ThingML program where these annotation have been replaced with actual invocation of our logging framework. In D2.2, we describe a first framework where ThingML events are reified into messages (transition fired, property changed, function called, etc.). For instance, the change of a property thus triggers the emission of the "property_changed" message, which carries additional information as string arguments (*e.g.*, name of the component, name of the property, old value, new value).

By contrast with D2.2, we propose an alternative logging framework where all ThingML events are reified into a single binary message that captures the type of event, the source component, and all additional information. As our evaluation showed, this alternative has a much lower overhead and better fits the constraints that govern software development on embedded devices. Again, we refer the reader to Appendix C for a complete description of both approaches and their overhead evaluation.

4.6 Evaluation

In this section, we first highlight in Section 4.6.1 the key requirements that are addressed by this latest version of GeneSIS. Then, we show how we evaluated our approach from two different perspectives: its usability via an empirical study (Section 4.6.2); and its effectiveness in the context of a case study (Section 4.6.3). Finally, we synthesize the evaluation results in Section 4.6.4 to discuss how we have addressed the requirements.

4.6.1 *KPIs & Requirements*

The three pilots in the ENACT project give real-life requirements for GeneSIS in which some of the key requirements are presented in the example in Section 2. We highlight the following requirements that are addressed by GeneSIS:

- **Separation of concerns and reusability (R1):** A modular, loosely-coupled specification of the data flow and its deployment is required so that the modules can be seamlessly substituted and reused. Elements or tasks should be reusable across scenarios.
- **Abstraction and Infrastructure independence (R2):** It is a need to be able to specify the orchestration and deployment of SIS over IoT, Edge, and Cloud infrastructures in both a device- and platform-independent and -specific way (GeneSIS enables this through a domain-specific language). In addition, a continuously up-to-date, abstract representation of the running system is required to facilitate the reasoning, simulation, and validation of operation activities.
- **White- and black-box infrastructure (R3):** Support for white- and black-box devices is required to cope with various degrees of delegation of control over underlying infrastructures and platforms.
- **Automation and adaptation (R4):** A fully automated deployment of SIS over IoT, Edge, and Cloud resources is required. In addition, the deployment of a system should be dynamically adaptable with minimal impact over the running system (*i.e.*, only the necessary part of the system should be adapted). The deployment and adaptation API exposed to the users should be technology agnostic and, as much as possible, device- and platform-independent.
- **Specify security and privacy requirements (R5):** GeneSIS should support the specification of the security mechanisms required and offered by the different components that form the SIS.
- **Automatic deployment and enforcement of generic security mechanisms (R6):** Support for continuously deploying security mechanisms should be offered. In addition to deploying security mechanisms as any software components, generic off-the-shelf security components that can be deployed in different scenarios and context are required.
- **Deployment on devices with no Internet connection (R7)**: Deployment of software on tiny devices that do not always have direct access to the Internet or even the necessary facilities for remote access is required.

Abstraction and infrastructure independence (R2) and automation (R4) are justified by the need for deploying the system on an infrastructure leveraging the IoT (*e.g.*, Arduino boards), Edge (*e.g.*, Raspberry PIs), and Cloud (*e.g.*, SMOOL server, Amazon EC2) spaces. Separation of concerns (R1), automation and adaptation (R4), and automatic deployment of generic security mechanisms (R5) are justified by the need for dynamically *(i)* adding a new software component to manage the access to the applications and *(ii)* updating security policies, with minimal impact on the already running system. The

support for white- and black-box infrastructure (R3) is justified by the need to use, in the same system, a black-box device (*e.g.*, the Z-Wave transceiver, SmartBox) and white-box devices (*e.g.*, Raspberry PI). The support for security and privacy requirements (R5) and enforcement (R6) is justified by the involvement in the system of actuators whose access should be controlled, and private data must be protected. Finally, support for deploying software on devices with no Internet connection is justified by the need to deploy software on the Arduinos (R7).

In the following sections, we present different evaluations on GeneSIS and how it addresses these requirements. We evaluated our approach from two different perspectives: *(i)* its effectiveness in the context of a case study and *(ii)* its usability via an empirical study.

4.6.2 *Empirical Study*

In this section, we present an empirical study on the usability of our approach in supporting the continuous deployment of IoT system. Whilst validating the effectiveness of our tools is a must, it is also critical to evaluate their acceptance by end-user. In the following, we report on the evaluation of usability and acceptability of our approach by end-users. We first detail the set-up before we dig into the results of the evaluation of the GeneSIS language.

4.6.2.1 Evaluation set up

We exploited a hackathon organized at Université Côte d'Azur to evaluate our domain-specific languages for the deployment and management of actuation conflicts as well as the integration of our two tools.

For each tool, two evaluation sessions were conducted as depicted in Table 3. Before any contact with the tool and associated language, each participant had to fill a first questionnaire about the language. This provided us with feedback about the acceptability of both the graphical concrete syntax and the concepts of our languages, which are non-biased *(i)* by prior knowledge of the language or *(ii)* by any extra knowledge that could be introduced by manipulating the tool. In a second stage, after a tutorial where the participants had the opportunity to use the tools, they were asked to fill again the exact same questionnaire. This provided us with the ability to evaluate how the understanding of the languages evolved after utilization of the tools. The questionnaires also included generic questions about the participants and their former experiences.

Table 3. Agenda of the Hackathon

8:00 – 9:00	Other hackathon activities
9:30 - 10:00	First questionnaire about GeneSIS
10:00 – 11:30	Tutorial and hands on
11:30 – 12:00	Second questionnaire about GeneSIS
13:00 – 13:30	First questionnaire about Actuation Conflict Management
13:30 – 14:30	Tutorial and hands on
14:30 – 15:00	Second questionnaire about Actuation Conflict Management
15:00 – 18:00	Other hackathon activities

4.6.2.2 Hands on

For the hand on, participants were organized in groups of two and each group was provided with: 1 Raspberry Pi 3B+ (including 1 SD card and a power plug), a set of RFID tags, 2 Arduino Uno (including 2 cables for serial connection and power supply). Each Arduino was wired to: 1 button with a led, 1 RGB led, 1 RFID reader. Participants were requested to deploy and manage actuation conflicts using GeneSIS and the Actuation Conflict Manager.

Participants were proposed to consider the case of an Access Control System (ACS) that manages access to specific areas. For the sake of simplicity, we considered only two rooms. The objective was to incrementally build the ACS, extending it with new features and evolving it as the purpose of the rooms change over the time.

In the first stage of the scenario, there is no ACS, both doors can be opened using the button or a badge (see Figure 23).

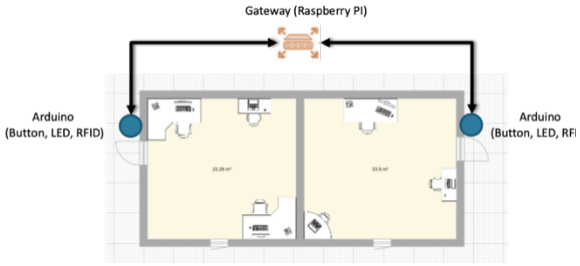


Figure 23. First stage of the exercise.

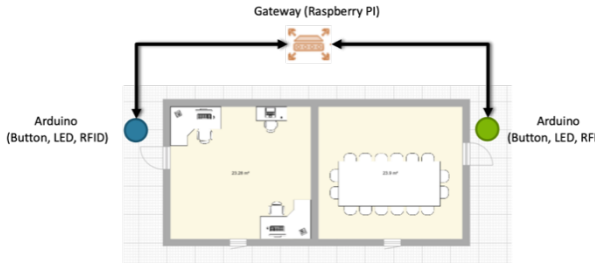


Figure 24. Second stage of the exercise

Looking at the physical infrastructure, the RFID reader and the button provided to the participants were both connected to an Arduino board that was itself connected to a Raspberry Pi. The state of the door (*i.e.*, open or closed) materialized as a LED. From the software perspective, the objective was to deliver a system that reacts to the button being pressed or to the identification of the badge by the RFID reader and to request the opening of the door. The control logic, which decides whether to open the door or not, was implemented in JavaScript using Node-RED and executed on the Raspberry Pi. By contrast, the software logic relaying the signals from the button and the RFID reader to the door was implemented in C and executed on the Arduino board.

In the second stage, it is decided to transform one of the offices into a meeting room (see Figure 24). The access policy for the office does not change but, for the meeting room, the system must react to the button being pressed or to a badge being read and requests the opening of the door only during office hours (*e.g.*, 8:00 – 18:00). Otherwise, users must present their badge to the RFID reader, which then signals and records their presence.

In the third stage, the logic to control the access to the meeting room is further improved. The meeting room can only be opened with the badge of a staff member invited to the meeting booked at the time of opening the door. The control logic deployed on the Raspberry Pi must then interact with the Agenda service used by the staff to book the meeting room.

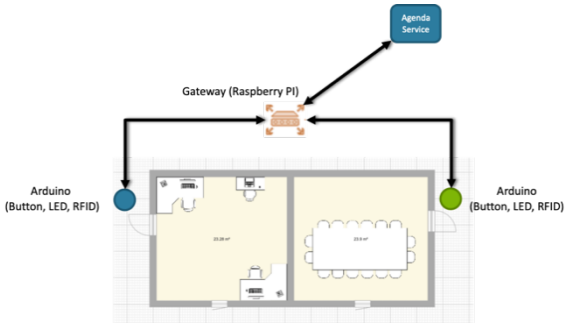


Figure 25. Third stage of the exercise

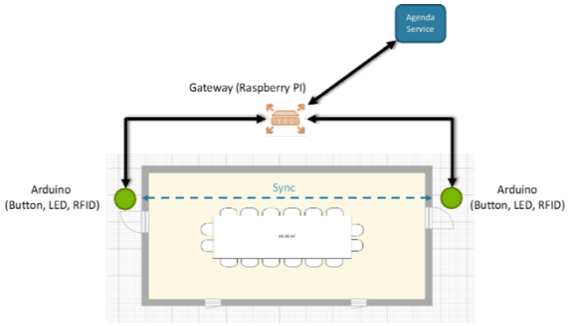


Figure 26. Fourth stage of the exercise.

Finally, the meeting room being too small, it is decided to transform the two rooms into one single meeting room (see Figure 26). Both doors are kept, making it possible to enter or leave by any of them. Whilst same access control policy as before for the meeting room is adopted, it is decided that both doors should be synchronised – *i.e.*, opening one door with a badge also result in the opening of the other door.

During this empirical evaluation, at each stage of the scenario, the participants used GeneSIS and the ACM tool. GeneSIS was used to perform first the initial deployment of the whole system and at each new stage, to incrementally deploy only the updated part of the system. The ACM tool was used at each stage to check for the existence of direct or indirect actuation conflicts, identifying and managing them with off-the-shelf ACMs or by writing their own customised ACM in case the components supplied were not suitable.

To carry out the first stage, the participants used GeneSIS to create the initial deployment model of the entire system. Before deploying it, they had to use the ACM tool to identify any potential conflicts. In the first stage, a direct actuation conflict was present between the use of the button and the RFID reader to open the door. An off-the-shelf component to manage this simple conflict was provided. A deployment was performed by sending the updated system by the ACM tool to GeneSIS.

In the second stage, with the transformation of an office into a meeting room, the participants had to modify the ACM into a new ACM to deal with direct conflict on the meeting room door. The updated model was sent to GeneSIS for performing the update.

At the third stage, the logic was a little more complex to manage access to the meeting room door, requiring the deactivation of the button, and the use of the RFID reader to access an agenda service validating the access. The participants had to write a custom ACM with the help of the ACM Designer. With the introduction of a new ACM, the updated application model is sent by the ACM tool to GeneSIS to update the system. As usual, only the modified parts of the system are updated by GeneSIS.

In the final stage, the modification of the room configuration and the synchronisation of the two access doors to the meeting room induces a change in the environment model in the ACM tool. This change leads to the detection of an indirect conflict between the two doors. The management of this conflict follows the same logic as in the previous step, the same component can be used, but on both doors.

4.6.2.3 Participants

A total of 38 participants from 10 different nationalities (see Figure 27) joined the hackathon, out of which, 25 already hold a relevant master degree. All participants were proficient in reading and writing English. Figure 27 also depicts the main domain of expertise of the participants. It is worth noting that each candidate could provide several domains, explaining a total of vote higher than 38. To sum up, three main domains were represented: IoT/Ambient Intelligence/embedded systems, Software engineering, and Telecom.

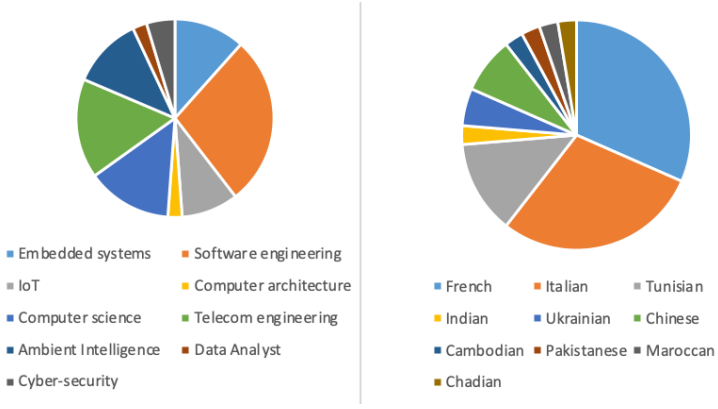


Figure 27. Background (left) and Nationality (right) of the participants

Finally, participants had little experience of use modelling languages (see Figure 28) and there was a good balance between participants used to design, develop and operate IoT systems and participants that never contributed to such projects (see Figure 28).

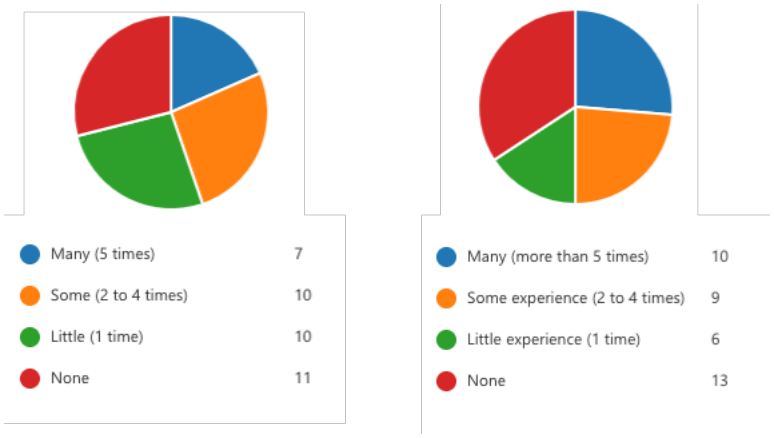


Figure 28. Experience of the participants in using modelling language (left) and in design, develop, or operate IoT systems (right)

Regarding participants' experience in using automatic deployment and software configuration management tools, almost all participants were experienced in using Docker and/or Kubernetes in particular for the deployment of Cloud applications (see Figure 29).

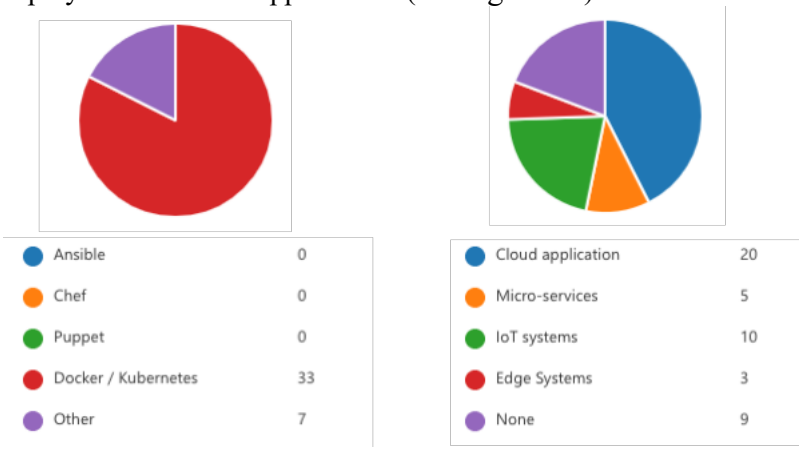


Figure 29. Experience of hte participant in using deployment tools (left) and the type of systems on which they apply the tools (right).

4.6.2.4 Results of the Evaluation

The first questionnaire about the GeneSIS modelling language started with an introduction to the main concepts of GeneSIS, *i.e.*, the types of components and links as well as their core properties. The average time answering the question was 25:36 minutes.

Figure 30 provides an overview of participants opinion about acceptability of the GeneSIS concepts. All 38 participants answered and, in general, the concepts are considered as intuitive. However, specific concepts such as the *isDeployer* and *isControler* properties appeared to be more complex to apprehend than others. It is also worth noting that the concept of *hosting* appears as slightly less intuitive than the concept of *communication*.

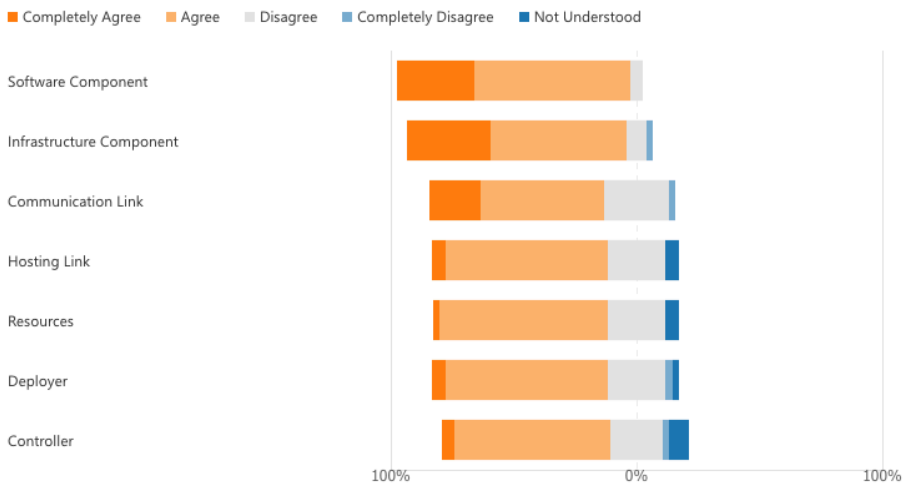


Figure 30. Acceptability of the GeneSIS concepts.

In the first question, the participants were asked to identify the types of all the components involved the deployment model shown in Figure 31, written using the graphical concrete syntax.

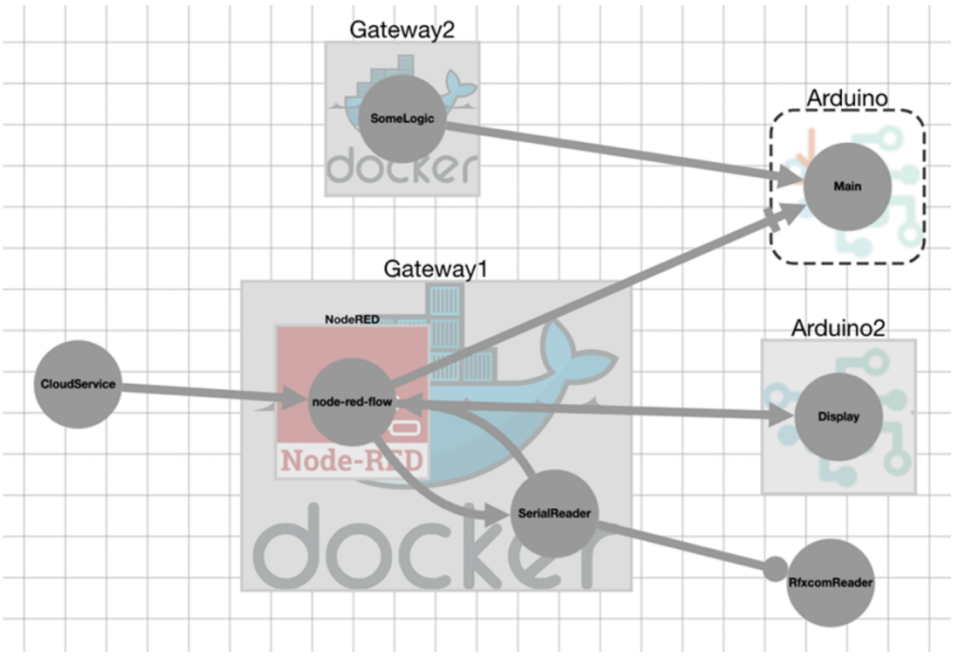


Figure 31. Deployment model analysed by the participants.

As depicted in Figure 32, in average, *InfrastructureComponents* were successfully identified by 86% of the participants (average of the percentage of good answers for all infrastructure components with a standard deviation between the components of 5.8). In average, *InternalComponents* were successfully identified by 76% of the participants (average of the percentage of good answers for all internal components with a standard deviation between the components of 5,8). Finally, in average, *ExternalComponents*, were successfully identified by 78,9% of the participants (average of the percentage of good answers for all external components with a standard deviation between the components of 0).

In the following questions, participants were asked to identify the type relationships between specific components: *communication* and *hosting*. 92% of the participants successfully identified the communication and 81% the hosting. It is worth noting that we do not provide standard deviation here

as the participants were only asked to identify one instance of each type of link. Overall, these results confirm the acceptability of the GeneSIS concepts and their representation in the graphical syntax.

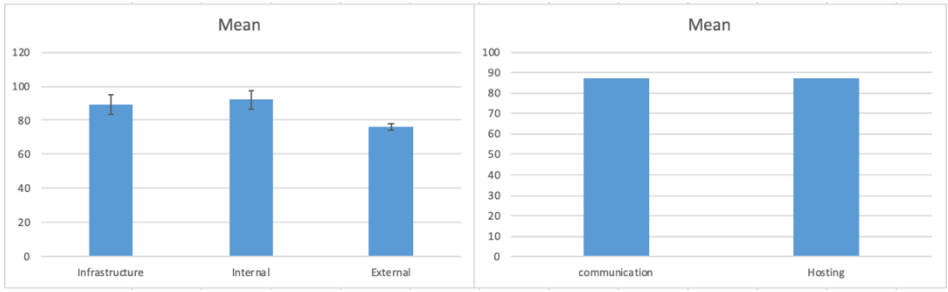


Figure 32. Successful identification of the GeneSIS concepts

However, the other questions helped us identify the following shortcomings. Only 36% of the participants were able to identify that the component *main* will be deployed via a deployment agent hosted on *Gateway1*. We believe this is mainly due to the fact that the graphical notation associated to the *isDeployer* (x arrow) and the *needDeployer* properties are not easy to interpret and thus need to be refined.

We also observed that only 65% of the participants answered properly to the question *"Who can communicate with the component main?"*. Considering the answers, we believe the main source of confusion are the following: *(i)* it is not clear if a hosting can also be used to communicate with a component, and *(ii)* it is not clear if a communication with the *isDeployer* property can be used for communication.

After the GeneSIS hands on, the participants were asked to fill the same questionnaire once more. The objective was to understand if the use of the tool helps assimilating the GeneSIS concepts. In general, we only observed a minor improvement in the participants' ability to identify the GeneSIS concepts in the deployment model depicted in Figure 31, which was already pretty high. Similarly, the opinion of the participants on the acceptability of the GeneSIS concepts improved slightly (see Figure 33). By contrast, we observe a consequent increase in the identification and understanding of the *isDeployer* property, moving from 36% to 67% of correct answers.

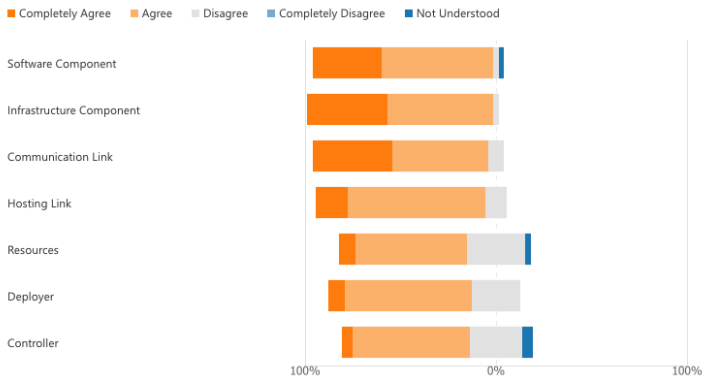


Figure 33. Acceptability of the GeneSIS concepts after the hands on.

4.6.3 *Evaluation of the DevSecOps Support for the Continuous Enhancement of Security Mechanisms*

In this section, we show how our approach has been applied to the smart home scenario in Section 2 inspired from the Tecnalia's Smart Building Kubik use case and served as a baseline for the evaluation

of GeneSIS by the use case provider. The deployment model of the smart home system has been specified by CNRS[12] as shown in Figure 34.

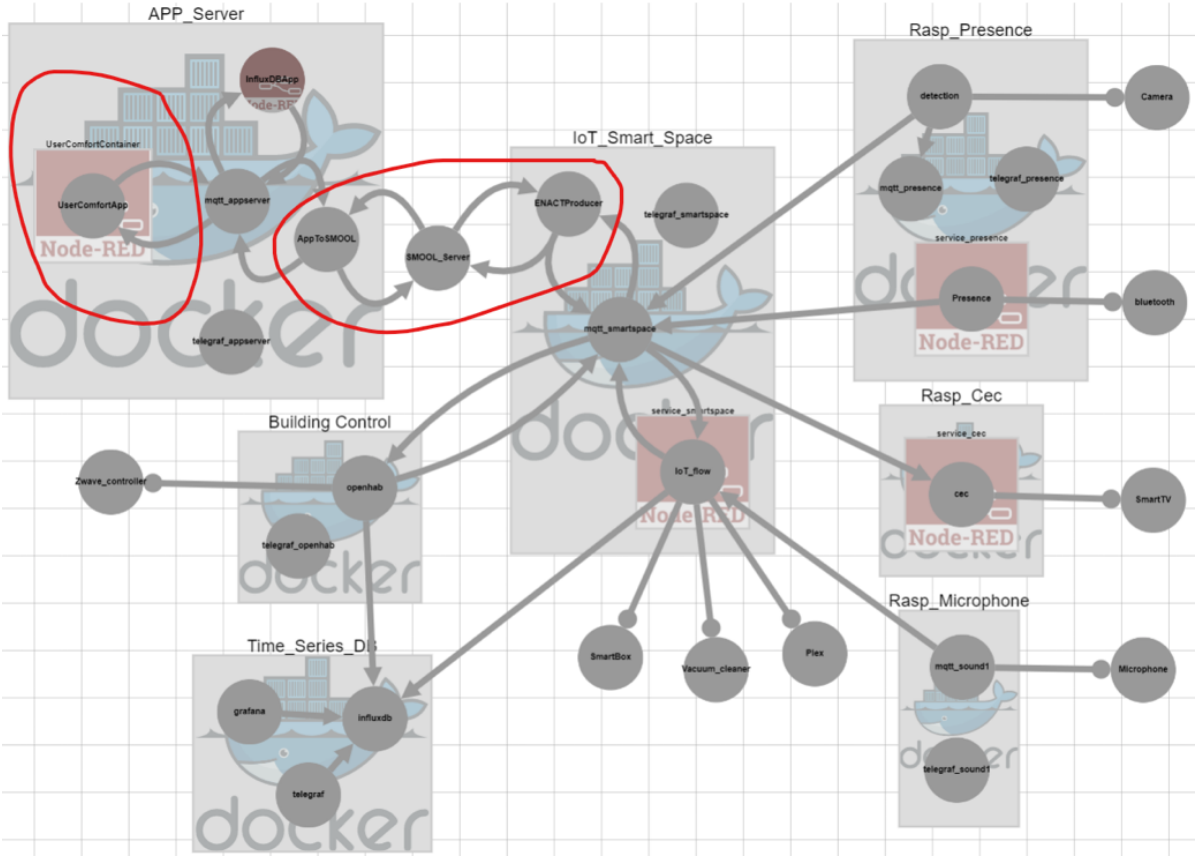


Figure 34. The initial version of the smart home system (deployment view)

In the initial version of the smart home system, there is an *UserComfortApp* application (on *App_Server*), which gets access to sensors' data to make decisions and send commands to control the actuators, *e.g.*, window blinds, for user comfort purposes. The *UserComfortApp* interacts with the smart home devices via the SMOOL platform (in the middle). The smart home devices send sensor data to the *UserComfortApp* and receive actuation commands from the *UserComfortApp* via MQTT protocol and the SMOOL middleware (*AppToSMOOL* and *ENACProducer* are two instances of SMOOL clients). More details on the deployment can be found in this video[13].

There are two notable security mechanisms associated in this version of the smart home system. The first one is a secure API gateway (Express Gateway [24]) that allows secure remote API access to the *UserComfortApp* application. The second one is a security checker by default of the SMOOL middleware that enforcing the security check for the data passing through, *e.g.*, *ENACTProducer* only allowing genuine actuation commands to be sent to the devices of the smart home.

The latest version of GeneSIS has provided a built-in support to ease the specification of the Express API Gateway in the deployment model. Adding a new instance of Express API Gateway is easy as shown in Figure 35.

[12] https://gitlab.com/enact/actuation_conflict_manager/-/tree/master/demos/demo-openhab
[13] https://gitlab.com/enact/actuation_conflict_manager/-/blob/master/demos/demo-openhab/2020-09%20ACM%20Smart%20Home.mkv

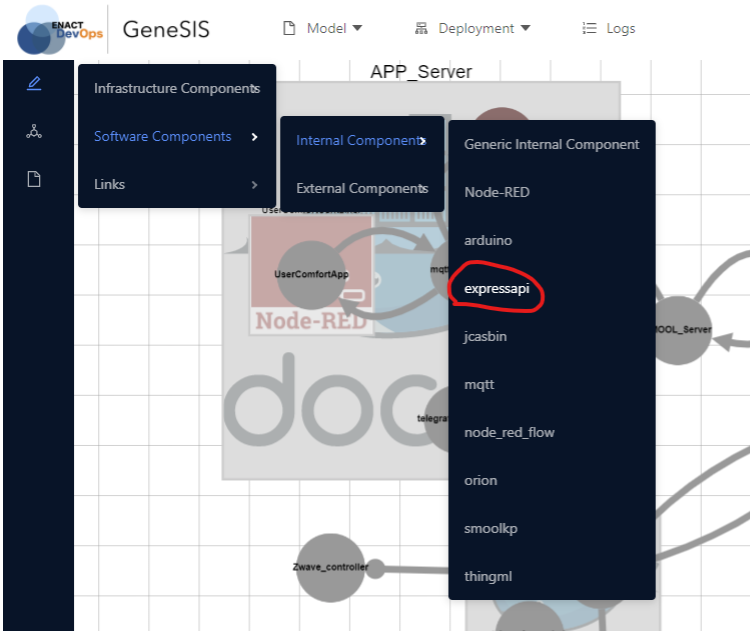


Figure 35. Specify an API gateway can be easy with a built-in ExpressAPI

The remaining work for the DevOps team is to specify the configuration files of the API gateway, and then configure how the API of the *UserComfortApp* application can be accessed. The configuration file *gateway.config.yml* in Figure 36 shows how the securely exposed API that can be used for controlling remotely the *UserComfortApp* application.

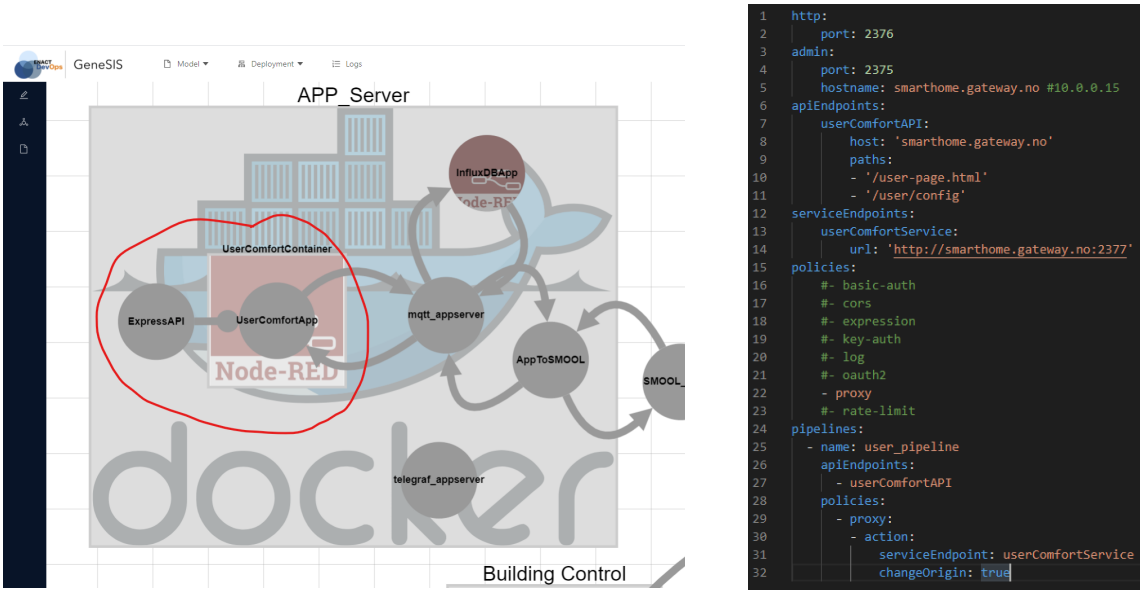


Figure 36. An instance of ExpressAPI has been added with the configuration file

GeneSIS has also provided support for configuring the security checker (enforcer) in the *ENACTProducer* component that makes sure only genuine actuation commands can be sent to the actuators as specified in the deployment model below.

```
{
   "_type": "/internal/SMOOLSecurity",
   "name": "SecurityEnforcer",
    ...
    "security_Policy": [["BlindPositionActuator", "Authorization"]],
    ...
}
```

With the *ENACTProducer* (*SecurityEnforcer*) is specified with this policy, all actuation commands (*e.g.*, *BlindPositionActuator*) sent from smart applications via *AppToSMOOL* must be accompanied with valid security tokens. The default security checker in *ENACTProducer* must validate security tokens successfully before the actuation commands can be sent to the smart objects (Figure 37).

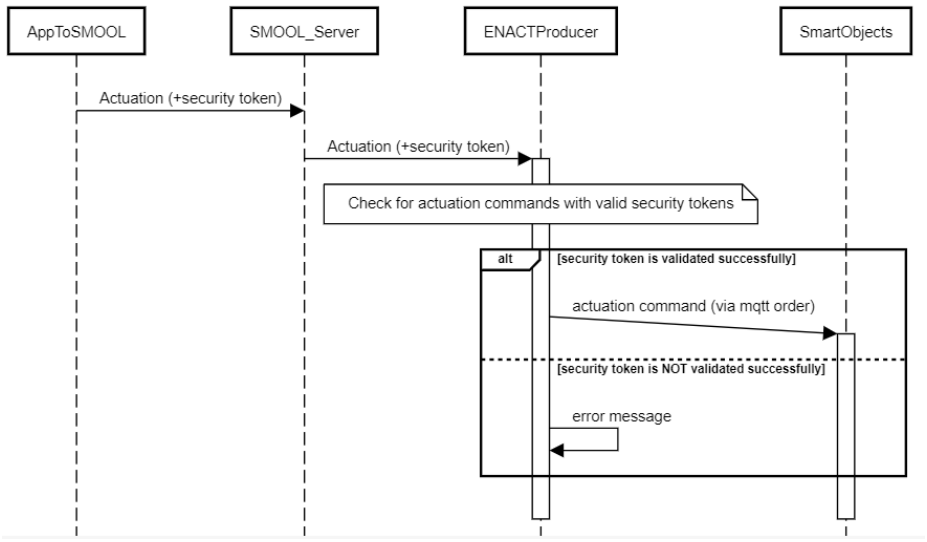


Figure 37. SMOOL security checker (in ENACTProducer) checks for actuation commands with valid security tokens

As said, during the evolution of the smart home system, new applications can be added, and new physical devices can also be added. New security requirements come up because the smart home system must control which apps can access which actuators. This means that more fine-grained security control must be introduced, which may not be available by default in IoT platforms. GeneSIS should support for seamlessly integrating new (third-party) security mechanisms into IoT platforms.

In the subsequent development cycle, two other applications called *ComCenterApp* and *DaikinACApp* have been added to the smart home system. Moreover, the smart home system can also have new IoT devices such as *MultiSensors* (and the corresponding software services) as shown in Figure 38.

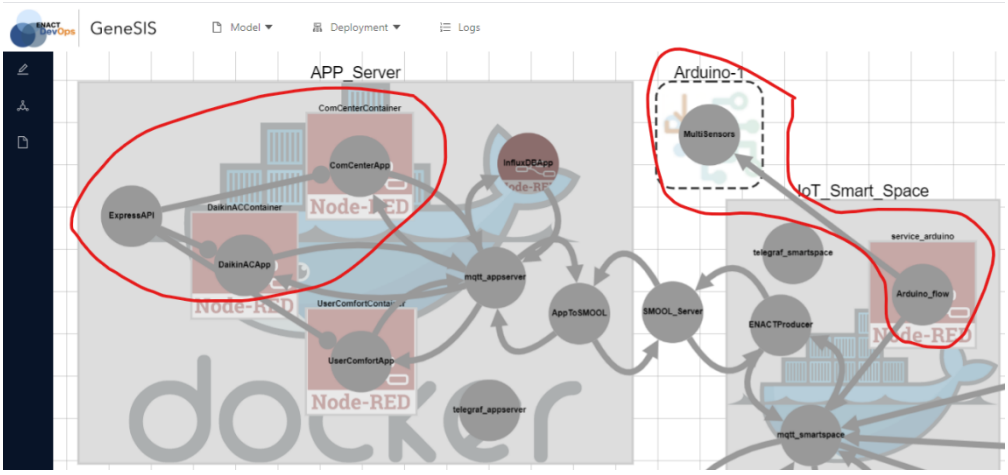


Figure 38. New applications and even new physical devices have been added

In this new development cycle, not only the secure API gateway must be updated with a new configuration file according to the two new apps, but also the DevOps team needs to introduce a new security mechanism that can enhance the fine-grained control of how the different applications can

access to the sensors and actuators of the smart home system. The DevOps team choose to develop the required access control mechanism based on an open-source framework like Casbin [14].

Updating the new configuration file for the API gateway is straightforward to enable secure remote API access to applications *EnergyEfficiency* and *UserComfortApp* as shown in Figure 39. However, to address the requirement of providing more fine-grained control of how the two applications can access to the sensors and actuators of the smart building, the DevOps team need to develop and deploy an advanced access control mechanism.

```yaml
http:
    port: 2376
admin:
    port: 2375
    hostname: smarthome.gateway.no #10.0.0.15
apiEndpoints:
    userComfortAPI:
        host: 'smarthome.gateway.no'
        paths:
            - '/user-page.html'
            - '/user/config'
    comCenterAppAPI:
        host: 'smarthome.gateway.no'
        paths:
            - '/ComCenterApp-page.html'
            - '/ComCenterApp/control'
    daikinACAppAPI:
        host: 'smarthome.gateway.no'
        paths:
            - '/DaikinACApp-page.html'
            - '/DaikinACApp/control'
serviceEndpoints:
    userComfortService:
        url: 'http://smarthome.gateway.no:2377'
    comCenterAppService:
        url: 'http://smarthome.gateway.no:2378'
    daikinACAppService:
        url: 'http://smarthome.gateway.no:2379'
policies:
```

```yaml
policies:
    #- basic-auth
    #- cors
    #- expression
    #- key-auth
    #- log
    #- oauth2
    - proxy
    #- rate-limit
pipelines:
    - name: user_pipeline
      apiEndpoints:
          - userComfortAPI
      policies:
          - proxy:
              - action:
                  serviceEndpoint: userComfortService
                  changeOrigin: true
    - name: comCenterApp_pipeline
      apiEndpoints:
          - comCenterAppAPI
      policies:
          - proxy:
              - action:
                  serviceEndpoint: comCenterAppService
                  changeOrigin: true
    - name: daikinACApp_pipeline
      apiEndpoints:
          - daikinACAppAPI
      policies:
          - proxy:
              - action:
                  serviceEndpoint: daikinACAppService
                  changeOrigin: true
```

Figure 39. The configuration file of ExpressAPI has been updated

The DevOps team can choose to develop the required access control mechanism based on an open-source framework like Casbin [14], or the Context-based Access Control mechanism presented in D4.3, or an in-house solution. Using GeneSIS, the DevOps team can specify the access control service to be deployed, *e.g.*, *jCasbin* (Figure 40). The integration of *jCasbin* into the existing SMOOL platform is easy thanks to GeneSIS' support for dynamically extending the security checker of the SMOOL platform to become a security enforcement point of the *jCasbin* service. More specifically, the DevOps team extends the security checker of the SMOOL platform with the code that passes on the security token in the call to jCasbin service (for access control decisions). Thanks to ThingML support within GeneSIS, the extended SecurityChecker is compiled and deployed that works as a security enforcement point of the *jCasbin* service. This seamless integration means that not only the actuation commands passing through the checker must be genuine, but also, they must conform to the access control policy defined in the *jCasbin* service (after being deployed by GeneSIS). In other words, *jCasbin* provides a fine-grained security control. For example, defining a role-based access control (RBAC) policy for the smart home system will make sure specifically what users of what apps can be allowed to take what actions on the objects of the smart home (*e.g.*, vacuum cleaner, TV). We use *jCasbin* as an example here but this way of enhancing security controls is applicable to other access control mechanisms like the Context-based Access Control mechanism in WP4.

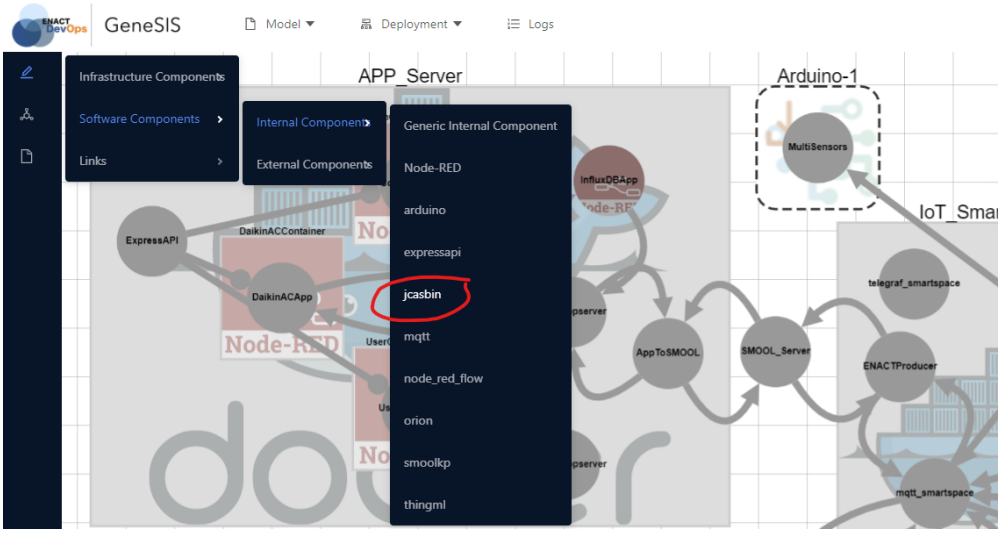


Figure 40. Specify an access control mechanism using a built-in jcasbin

Figure 41 shows how the *jCasbin* service has been integrated into the existing smart home system. We now have enhanced the default SMOOL security checker (in *ENACTProducer*) to become a security enforcement point (PEP) of the *jCasbin* service. The architecture between these two components is following the well-known PDP-PEP model, in which the *jCasbin* service is the policy decision point (PDP). The enhanced SMOOL security checker is now not only checking security tokens by default. After validating the security token successfully, the security checker also sends the requested action (inside the token) to the *jCasbin* for checking against the fine-grained access control policy (a predefined policy deployed together with *jCasbin*). Figure 42 shows the RBAC policy model of *jCasbin* and an excerpt of the RBAC policy instance defined for the smart home system.

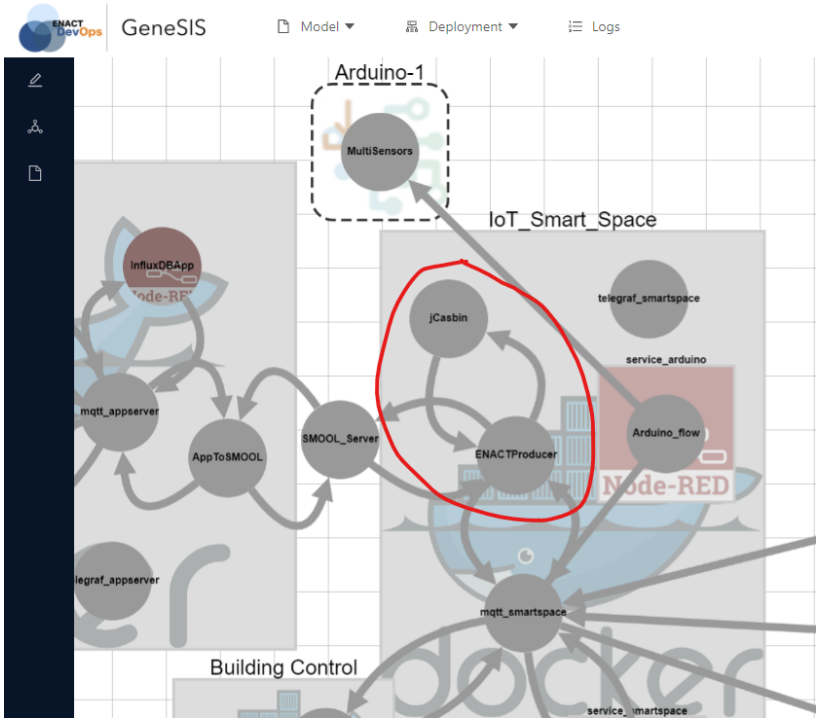


Figure 41. The jCasbin instance has become the policy decision point for the ENACTProducer

```
[request_definition]
r = sub, obj, act

[policy_definition]
p = sub, obj, act

[role_definition]
g = _, _

[policy_effect]
e = some(where (p.eft == allow))

[matchers]
m = g(r.sub, p.sub) && r.obj == p.obj && r.act == p.act
```

```
p, DaikinDevicesControl_W, Vacuum_cleaner, write
p, DaikinDevicesControl_R, Vacuum_cleaner, read
g, DaikinACApp, DaikinDevicesControl_R
g, DaikinACApp, DaikinDevicesControl_W
g, UserComfortApp, DaikinDevicesControl_R
g, ComCenterApp, DaikinDevicesControl_R

p, SmartTV_W, SmartTV, write
p, SmartTV_R, SmartTV, read
g, UserComfortApp, SmartTV_W
g, ComCenterApp, SmartTV_W
g, UserComfortApp, SmartTV_R
g, ComCenterApp, SmartTV_R

p, SmartBox_W, SmartBox, write
p, SmartBox_R, SmartBox, read
g, UserComfortApp, EntertainmentDevicesControl_W
g, ComCenterApp, EntertainmentDevicesControl_R

p, BlindPositionActuator_W, BlindPositionActuator, write
p, BlindPositionActuator_R, BlindPositionActuator, read
g, UserComfortApp, BlindPositionActuator_W
g, UserComfortApp, BlindPositionActuator_R
```

Figure 42. The RBAC policy model (left) and an excerpt of the RBAC policy defined for the smart home

Figure 43 shows the GeneSIS's GUI for extending the security checker in the *ENACTProducer* component to become a security enforcement point of the *jCasbin* service. With ThingML support in GeneSIS, the extended *SecurityChecker* is compiled in the *ENACTProducer* component for a new version of *ENACTProducer* to be deployed that works as a security enforcement point of the *jCasbin* service. In other words, GeneSIS eases the configuration and deploys the newly enhanced *ENACTProducer* together with the *jCasbin* service according to the PDP-PEP model discussed above.

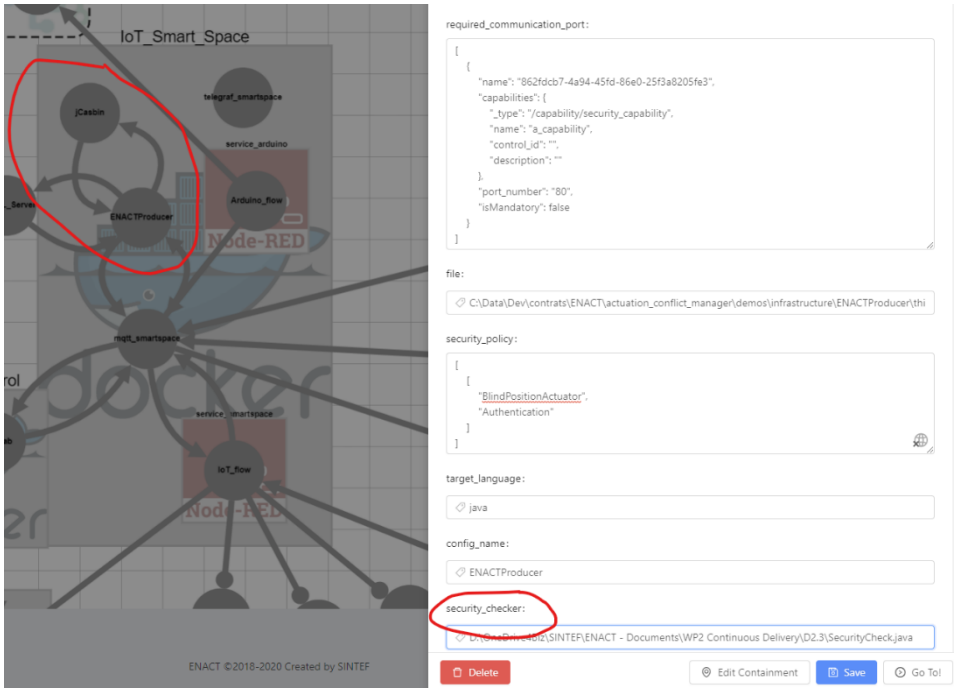


Figure 43. The GeneSIS's GUI for extending the security checker in the ENACTProducer component

After the successful deployment of this enhanced security control, the behaviour of the smart home system regarding the actuations can be seen in Figure 44. All actuation commands (*e.g.*, *BlindPositionActuator*) sent from smart applications via *AppToSMOOL* must be accompanied with valid security tokens. The default security checker in *ENACTProducer* must validate security tokens. If the security token is validated successfully, the checker will forward the requested action (the actuation command) to the *jCasbin* service to check against the access control policy deployed there. The *jCasbin* service will return the decision to the checker. The actuation command will be sent to the smart object(s) only if the permission has been granted.

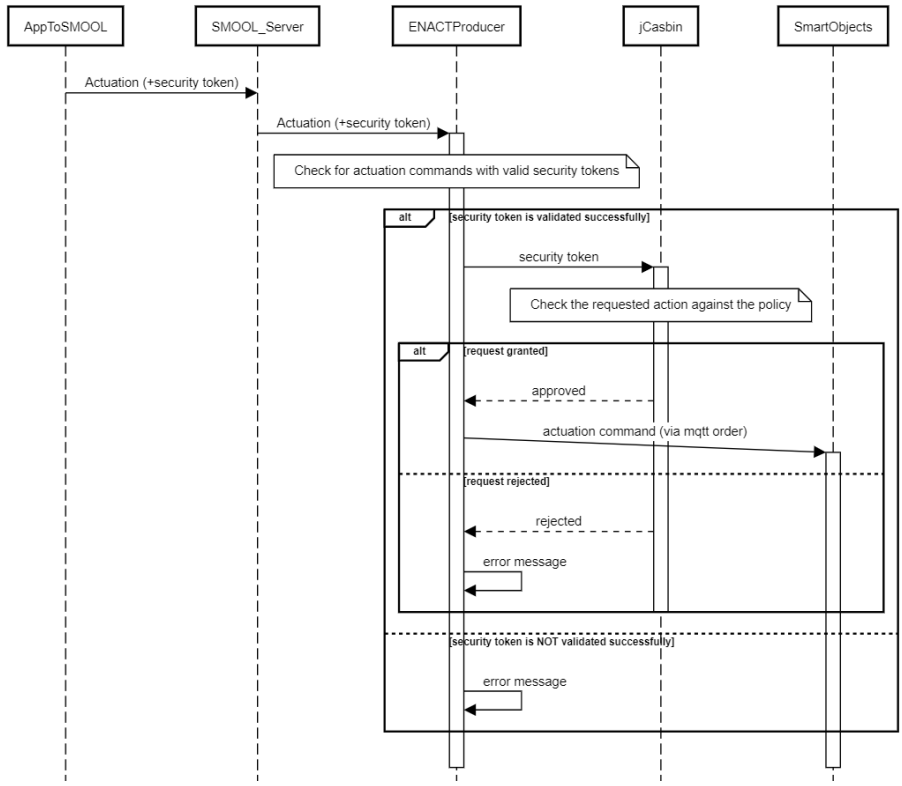


Figure 44. The jCasbin service is the security decision point to grant permissions according to the policy

More advanced requirements may be also needed such as the constraints of the deployed security and privacy controls. Constraints for deploying security modules can be considered by GeneSIS such as how far the host of a security module is from the sensors or actuators. For example, the authorization module may need to be deployed and executed on a local node to the Building Control gateway or the gateways that host smart devices to ensure the performance of the authorization mechanism. The importance of having a PDP close to the actuators is allowing actuation commands to be executed in a timely manner, without being hindered by the possible network delay.

In our experiment, GeneSIS has provided a convenient tool support for extending and integrating an access control mechanism (*e.g.*, Casbin) to work locally with the default security mechanisms of an IoT platform (*e.g.*, SMOOL) that is used for the development and operation of the SIS (the smart home system). GeneSIS enables the DevOps support by allowing a smooth security integration from development phase, to the operation phase, via automated deployment and orchestration. Figure 45 shows the full deployment model of the smart home system with the newly integrated security mechanisms. In this latest version of the smart home system, the three main applications on the *APP_Server* can be securely accessed via the API gateway. Moreover, a fine-grained security control has been enforced to control how the users/processes of these applications can get access to the smart services and devices.

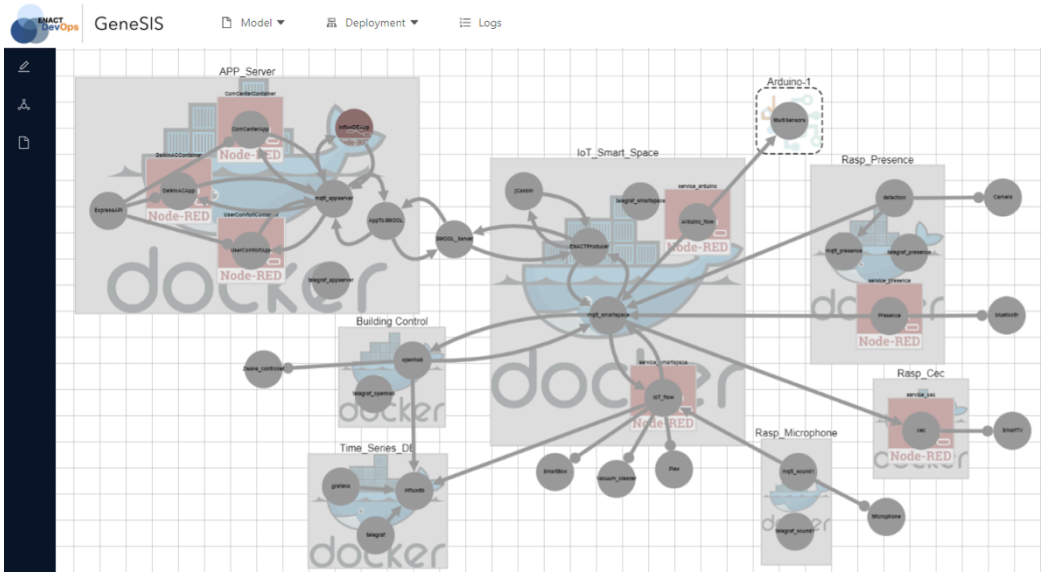


Figure 45. The full deployment model of the smart home system with the newly integrated security mechanisms

Note that after a successful deployment, during the operation phase, GeneSIS supports for adapting security enforcement to improve how the IoT system operates securely at runtime. In this line of adaptation, new security policies or configurations can be updated dynamically for the security mechanisms that are in operation. The trigger of such adaptation can be manually, but also can be automatically, for example, after a reasoning process of actuation conflict management.

In both ways presented so far, GeneSIS allows DevSecOps teams to reconfigure and update security mechanisms by design, in line with the evolution of IoT applications and the development of security and privacy risks.

4.6.4 *Synthesis*

In the following, we evaluate how our approach addresses the requirements defined in *(i)* Section 3.3.1 and *(ii)* the current status with respect to the use case requirements as defined in D1.1. Based on our experiments, we have shown that GeneSIS is capable of supporting for:

- Separation of concerns and reusability (R1): A modular, loosely-coupled specification of the data flow and its deployment is required so that the modules can be seamlessly substituted and reused. In both scenarios, several component types were reused (*e.g.*, Node-RED components) in the deployment model. In addition, new component types were created and integrated to the GeneSIS component repository (*e.g.*, *ExpressGateway*, *MQTTBroker*, *jCasbin*). These components are typical candidates for reuse in other scenarios. The component-based design of the GeneSIS modelling language ensures that the provisioning and deployment models are modular and loosely-coupled. In addition, the type-object pattern in the metamodel of the GeneSIS modelling language ensures that component types can be reused across several models.
- Abstraction and infrastructure independence (R2): There is a need to be able to specify the orchestration and deployment of SIS over IoT, Edge, and Cloud infrastructures in both a device- and platform-independent and -specific way. In addition, a continuously up-to-date, abstract representation of the running system is required to facilitate the reasoning, simulation, and validation of operation activities. The GeneSIS modelling language offers a single domain-specific modelling language and abstraction that enables the management of application deployed on IoT, Edge, and Cloud infrastructure. Independently of IoT layers, these resources as well as the software components can be abstracted in a homogeneous way as components. In addition, by applying the Models@run.time pattern, the GeneSIS execution environment provides an abstract and up-to-date representation of the running system that can be dynamically manipulated.

- White- and black-box infrastructure (R3): Support for white- and black- box devices is required to cope with various degrees of delegation of control over underlying infrastructures and platforms. In the context of Kubik, both the SMOOL server and the PLC are considered as *ExternalComponent* not managed by GeneSIS. Nevertheless, it is possible to seamlessly orchestrate these components with all the other *InternalComponents*. The GeneSIS modelling language embeds the necessary concepts for the GeneSIS execution environment to distinguish and orchestrate white-box (*i.e.*, resources on top of which GeneSIS can manage a software stack) and black-box resources (*i.e.*, resources coming with a software stack that cannot be manipulated). More specifically, we refer here to the concept of *InternalComponent* and *ExternalComponent*, respectively.
- Automation and adaptation (R4): From a deployment model, GeneSIS supports the fully automated deployment of a SIS. By applying the Models@Run-time pattern GeneSIS provides developers with the means to adapt the deployment of a SIS. In addition, the deployment of a system should be dynamically adaptable with minimal impact over the running system (*i.e.*, only the necessary part of the system should be adapted). In our evaluation, we updated the deployment by adding the *UserComfort* application, the *ExpressGateway*, and the *SecurityEnforcer* component. Thanks to the adoption of the Models@run.time pattern, only these parts of the application were adapted.
- Specify Security Requirements (R5): GeneSIS should support the specification of the security mechanisms required and offered by the different components that form the SIS. As presented in both evaluation scenarios, when modelling the deployment of the Kubik smart IoT system, several required and provided *SecurityCapabilities* were specified. Related to the need for the *ExpressGateway* and the *SecurityEnforcer*, GeneSIS provides mechanisms for specifying security and privacy capabilities provided and required by a component.
- Automatic Deployment and Enforcement of Generic Security Mechanisms (R6): GeneSIS offers support for the deployment of security mechanisms. In particular, a specific component type can be used for the generic deployment of security monitoring and control mechanisms on IoT platforms. In the context of the smart building, the Building Control and the actuators cannot be accessed without proper authorization. The smart building apps are built on top of the SMOOL platform and uses clients implemented in Java. The GeneSIS-SMOOL security component was used to deploy the *SecurityEnforcer* component, automatically injecting this security policy into the SMOOL client. The *SecurityEnforcer* will allow passing only messages of type "*BlindPositionActuator*" containing some specific data in the Authorization concept. The *security_policy* property of the *SecurityEnforcer* (*InternalComponent)* was thus specified from the GeneSIS' web UI. The *security_policy* property of the *SecurityEnforcer* (*InternalComponent)* was thus specified from the GeneSIS' web UI. The security policy was injected in the ThingML code of the SMOOL client and compiled to Java, in turn. The proper Maven manifest was automatically created and used to build the application before deployment.
- Devices with no Internet Connection (R7): By leveraging the concept of deployment agent, the GeneSIS execution environment can be extended to Edge devices, which, in turn, can deploy software components on devices only accessible locally. In the first scenario, the Arduinos such as Arduino 1 in the smart home (Section 2), do not offer remote access and can only be accessed via their serial connections with RPI-2. The deployment of the software components on these devices must be enacted by a deployment agent.

By addressing all these requirements, and in particular R6 and R4, GeneSIS provides support for the DevSecOps of SIS. Our case study shows how GeneSIS enables the security-by-design of SIS that can modulate a variety of security features at operation depending on real needs. The system requirements may change during its lifetime and still the re-deployment by GeneSIS of enhanced security features into already running applications makes it possible that SIS adapts to evolving conditions and threats while keeping their trustworthiness.

Table 4. Requirements from D2.2

ReqID	Requirement	Description	Status at M35
UC1-3 **R1**	Scalability	GeneSIS should be able to deploy SIS involving hundred sensors/actuators.	**Available.** The integration of GeneSIS with DivENACT tool (see D4.3) allows to leverage Microsoft IoT Hub has helped fulfilling this requirement.
UC1-3 **R2**	Scalability	The GeneSIS modelling language should be able to represent deployments involving hundred sensors/actuators.	**Available.** The integration of GeneSIS with DivENACT tool (see D4.3) has helped to fulfil this requirement.
UC1-3 **R3**	Trustworthiness and Agility	GeneSIS should support the re-deployment (*e.g.*, moving one software node from one host to another), re-configuration, and update (install new version of a software node) of software components.	**Available.** The full version of the "Diff" in the GeneSIS models@runtime engine has been developed. Support for re-deployment is available.
UC3 **R4**	Trustworthiness	GeneSIS should help identifying direct actuation conflicts (*i.e.*, concurrent accesses to a same component).	**Available.** GeneSIS provides the controller mechanisms and can be consumed by the actuation conflict management enabler.
UC1-3 **R5**	Scope	GeneSIS should be able to deploy SIS involving IoT, edge and cloud infrastructures.	**Available.** GeneSIS can successfully deploy over IoT, Edge, and Cloud infrastructure. Test has been made against deployment models involving: Arduino, RaspberryPI, regular laptops, and AWS clouds resources
UC1-3 **R6**	Scope	The GeneSIS modelling language should be able to represent deployment over IoT, Edge, and cloud infrastructure	**Available.** GeneSIS can successfully represent deployment involving IoT, Edge, and Cloud resources. Test has been made with deployment models involving: Arduino, RaspberryPI, regular laptops, and AWS clouds resources
UC1-3 **R7**	Trustworthiness	The GeneSIS language will support the specification *(i)* of the security mechanisms to be deployed and *(ii)* of metadata (*e.g.*, software version) for each of the elements in a model.	**Available.** The GeneSIS modelling language has been extended with concepts to specify required and provided security and privacy capabilities and to how the required capabilities can be fulfilled. Moreover, the language has supported the specification for integrating security components together that promotes the DevSecOps practices.
UC1,3 **R8**	Integration	GeneSIS should properly integrate with classical IoT middleware (*e.g.*, SMOOL, SOFIA2)	**Available.** Integration of GeneSIS via ThingML has been done and tested.
UC1-3 **R9**	Elasticity	GeneSIS should support the provisioning of cloud resources.	**Available.** Integration with the CloudML provisioning engine is available. In addition, provisioning of docker resources is also available.
UC1-3 **R10**	Elasticity	The GeneSIS modelling language should provide the necessary concept for specifying the cloud resources to be provisioned.	**Available.** GeneSIS leverage the CloudML approach to specify the provisioning of multi-cloud resources.
UC2 **R11**	Monitoring	The GeneSIS language should include the necessary concepts to reflect directly in the language run-time data. In particular information about the deployment and	**Available.** GeneSIS implements the models@runtime pattern, which provides a mean to enhance deployment models with runtime informations. At the current

		infrastructure status as well as about the execution flow of ThingML programs.	moment GeneSIS monitors information about the status of a deployment. The full version of the mechanisms to monitor the flow of ThingML programs is available.

Table 5. Requirements from D1.1

ReqID	Requirement	Description	Status at M15
TO1.1	ITS use case	**Real Time Traffic Management Plan updates on On-Board Systems.** Demonstrate the remote and continuous deployment of, at least, two cabins with OTI and a Plan update during the operation part. i.Including at least 2 gateways and 2 IoT devices i.Including at least 1 cloud resource i.Including at least 2 deployments v.Including at least 1 updated software component	**Done.** We demonstrated the deployment update of 20 software components on 20 devices provided by INDRA.
TO1.2	ITS use case	**SW development for the infrastructure deployed.** Agile Software deployment on the CMWs Demonstrate a valid deployment with lack of human interaction in a reduced amount of time (limited by the device) increasing the efficiency in operation time and workload. Demonstrate the remote deployment of the CMWs: *(i)* Including at least 2 gateways and IoT devices, *(ii)* Including at least 1 cloud resource	**Done.** We demonstrated deployment of 20 software components on 20 devices provided by INDRA. Deployment was trigger without human intervention. More precisely, deployment was triggered when 2 conditions were satisfied: *(i)* a new version of the software is available in the repository, and *(ii)* message has been received that the train is ready for update.
TO4.1	ITS use case	**Demonstrate the integration of the ITS SIS with the FiWARE (Orion Context Broker)**	**Done.** Integration with the Orion context broker has been demonstrating in a lab setting. Demonstration in the context of the use case will be done in collaboration with WP1
TO1.1	eHealth	**Continuous deployment across the IoT, edge and Cloud space.** *(i)* Include at least 2 IoT&Edge nodes and 2 cloud nodes. *(ii)* 10 Multiple deployments *(iii)* Includes orchestration, setting up interoperation with "no downtime"	**Under evaluation by use case.** Deployment in combination with DivENACT on a set of 15 nodes (including arduinos, and Tellu gateways), with different deployments depending on each device capability. Deployment with or without blue/green deployment for zero downtime.
TO1.2	eHealth	**Automatic change or upgrade.** Demonstrate 5 changes or upgrades of 5 independent Gateways. Performed automatically and without need of physical intervention (or minimize physical intervention)	**Done.** Update demonstrated with and without blue/green deployment. Deployment performed in collaboration between DivENACT and GeneSIS
TO1.3	eHealth	**Automatic Pairing of devices with Gateway after reset.** At least 5 different devices automatically paired with Gateway after reset	**Done.** The software developed for the automatic pairing of Bluetooth devices is automatically deployed using DivENACT and GeneSIS.

TO1.1	Smart building	**Interface with DSS.** The Orchestration enabler interacts with the DSS enabler to provide it with the list of devices selected as part of the SIS.	**Done.** Risk management is able to consume GeneSIS deployment models and can thus retrieve the list of devices involved in a deployment.
TO1.2	Smart building	**Integration with SOFIA/SMOOL.** Demonstrate the integration of the Orchestration and deployment enabler with the SMOOL platform: *(i)* Demonstrate the continuous deployment of SMOOL client and their automatic integration with SMOOL broker. *(ii)* Demonstrate data exchange of the deployed components via SMOOL.	**Done.** Integration has been achieved between GeneSIS, ThingML and SMOOL. This includes the deployment of security mechanisms.
TO1.3	Smart building	**Deployment of the use case applications** Demonstrate the continuous deployment of the two smart building applications. Including actuation conflict managers and S&P monitoring probes.	**Done.** A full DevOps scenario has been tested, deploying all the applications presented in Section 2. The scenario validate the integration between ACM, S&P monitoring and control, and GeneSIS. GeneSIS deploys actuation conflict managers and S&P monitoring and controls mechanisms at the IoT platform level (SMOOL)

4.7 Beyond ENACT

In the future we will investigate extending GeneSIS and its deployment engine in the following directions. First, we plan to explore new deployment strategies such as the A/B and canary strategies as part of our rolling deployment approach, providing a way to only introduce a new version of an application to a limited number of replicas providing a mean to evaluate this new version in production but with limited impact on the running system. We will also investigate how to extend our rolling deployment strategy to resource constrained devices that do not offer support for dynamic loading (*i.e.*, deploying a software component requires flashing the whole firmware of the device). In such case other hybrid strategies leveraging Edge devices can be envisioned. For instance, we can foresee deployment patterns where a degraded but high-quality version of the software is deployed on the resource constrained devices whist execution of the latest version is to an Edge device (if both are connected to each other) and can thus benefit from rolling deployment. The degraded version on the resource constrained devices can be used as a backup in case the latest version of the service crashes. Whilst response time would be degraded, availability would be improved.

5 Detecting, Analysing, and Managing Actuation Conflicts

5.1 Overview and Main Achievements

5.1.1 *Overall approach*

Within a Smart IoT System (SIS), several applications may concurrently access actuators. These competing accesses may entail *conflicts* resulting in indeterministic, inconsistent and unexpected behaviours. A *direct actuation conflict* occurs when at least two applications concurrently access a

software resource, in front of an actuation system, with antagonist purposes. For instance, let us consider the following two applications. The first one turns the light on when a person is detected in the room while the second turns the light off when the TV is in use. This situation may cause the light to blink, yet this is even not sure. A SIS may also face *indirect actuation conflicts*. This type of conflict is more insidious as it appears when two applications act on separate actuators whose effects interfere through the physical environment. The effects resulting from these actions are typically unpredictable and difficult to anticipate from the software application models.

Traditionally, the management of actuation conflicts is handled at the code level, often in an ad-hoc fashion. The detection and resolution of conflicts is then a one shot and time-consuming activity that requires a deep understanding of the SIS code, preventing its integration as part of an agile and continuous delivery process. As a result, to the best of our knowledge, there is currently no solution for managing actuation conflicts within DevOps.

The actuation conflict management enabler [25] aims at addressing this issue, supporting the detection, analysis and resolution of actuation conflicts as part of a typical DevOps process thanks to the following core features:

- Modelling: The overall conflict management process applies over and abstract representation of the SIS that is decoupled from its detailed code.
- Automation: The tool is conceived to *(i)* automate as much as possible the conflict management process and *(ii)* to integrate with other continuous delivery and deployment solutions for maximum agility.
- Verification: Verifications mechanisms ensure that the conflict management solution to be injected in the SIS satisfies temporal and logical properties making DevOps teams confident to place the new version in the system.

The enabler first extracts and abstracts a WIMAC model (Workflow and Interaction Model for Actuation Conflict management) from the application implementation, its deployment model (GeneSIS model) and a model of the environment. This model contains the necessary information for the actuation conflict management enabler to automatically identify direct and indirect actuation conflicts. Based on this model and predefined conflict patterns, actuation conflicts are identified and resolved by automatically injecting generic off-the-shelf actuation conflict management solutions, hereafter called Actuation Conflict Managers (ACM). At this stage, two options are offered to the DevOps team: *(i)* approving the off-the-shelf ACM automatically selected and moving to the next stage, *(ii)* replacing the selected ACM by another one (either off-the-shelf or not). Because the generic off-the-shelf ACMs are not necessarily tailored or optimized to the very type of actuation conflict under consideration, DevOps teams are provided with tool support for the design of specific actuation conflict managers, denoted by Custom ACMs in the sequel. More precisely, Custom ACMs are specified using a domain-specific modelling language, providing the ability to conduct simulations and verifications of logical and temporal properties, also considering the specificities of the targeted platforms. Custom ACMs can in turn be added to the library of off-the-shelf ACMs. Once the ACMs are integrated into the application implementation, the deployment and application models are updated accordingly.

To achieve all these, the actuation conflict management enabler consists in a set of tools and transformations (workflow is detailed in Figure 47). In the following section, we detail the WIMAC language (Section 5.2.1) and the chain of transformation that drives the detection of conflicts, their resolution, and the update of the application and of its deployment (Sections 5.2.2 and 5.3.2).

5.1.2 *Main achievements and innovations*

As detailed in our systematic mapping study [26] (and also see sections 4.1 and 4.2 in D2.1), while many solutions exist for managing concurrent accesses to shared software resources, very little effort has been devoted to providing solutions tailored to DevOps and the IoT domain [27]. A first reason is the lack of abstraction and languages that decouples the actuation conflict

management from the low level code whilst still providing all the necessary concepts. A second reason is the lack of automation and of integration with solutions for the continuous deployment of SIS (which is offered in ENACT by GeneSIS). To identify and resolve actuation conflicts, it is necessary to understand how the different software components and actuators of a SIS are orchestrated and deployed. Yet, very little effort has been spent on providing solutions tailored to the delivery and deployment of applications across the whole IoT, Edge, and Cloud spaces [1], preventing such integration. A third reason is the lack of tools and mechanisms for specifying safe ACMs.

To address these limitations, we use a model driven engineering approach with reasoning engines and models transformations for detecting and resolving actuation conflict in the overall SIS. We also use finite state models, model-checking algorithms, and discrete event systems simulation for designing and for validating some custom solutions to resolve some local actuation conflict.

The main contributions of the actuation conflict management enabler are then the following:

- A unique SIS model "Workflow and Interaction Model for Actuation Conflict Management" (WIMAC) to that includes all the necessary concepts to for actuation conflict management.
    - This model is instantiated from the deployment model, IoT Application models, and the physical environment model of the SIS.
    - By leveraging this model some automatic or semi-automatic tools assist the DevOps team in actuation conflict detection and resolution.
- A tool to automatically detect and resolve conflict by applying a set of transformation rules on the WIMAC model. The application of these rules results in the injection of off-the-shelf ACMs into the WIMAC model.
- A semi-automatic tool for designing new custom and safe ACMs , which can be used to replace automatically selected off-the-shelf ACMs.

At the end of the process, new or updated deployment and IoT Application models are produced from the WIMAC model. The implementation of the custom ACM, when needed, is also provided as a deployable artefact for the target platforms.

Table 6: actuation conflict management enabler main achievements since M22

Feature	Description	Status at M22	Status at M35
DevOps-oriented features (main features)			
Analyse development and deployment models aiming to identify actuation conflicts	See D2.2	Completed	Completed
Specify environment model (physical and logical) to help identifying indirect *conflicts*	See D2.2	Completed	Completed
Detect direct and indirect actuation conflicts	See D2.2	Completed	Completed
Adapt policies rules to identify/resolve conflicts	See D2.2	Completed	Completed
Select off-the-shelf generic actuation manager to resolve identified conflicts	See D2.2	Completed	Completed

Specify new dedicated custom action manager using a user-friendly formalism (ECA rules)	See section 5.2.2.5	Ongoing	Completed
Deploy updated applications with managed action conflicts	See section 5.2.2.6	Ongoing	Completed
Trustworthiness-oriented features			
Reliability: Validate logical and temporal properties of custom actuation manager	See section 5.2.2.5	Ongoing	Completed
Automatic conflict detection and end-to-end design of safe solution	Thanks to the last improvements in actuation conflict management Enabler (see section 5.1.3)	Ongoing	Completed
Interaction with other enablers			
Node-RED, MQTT	See D2.2	Completed	Completed
GeneSIS (deployment model) used as input model to identify conflict and as output to deploy updated applications without conflicts	See D2.2	Completed	Completed
ThingML: development model of embedded applications on things, with behaviour models	See D2.2	Completed	Completed

5.1.3 *Improvements over D2.2*

Compared to the initial release of the enabler in D2.2, the main improvements of the actuation conflict management enabler can be summarized as follows:

- **New version (V2) of the WIMAC modelling language**. The language has been extended with the concepts of *Composite Applications*. It also provides developers with the ability to define a set of *Physical Properties* associated to *Physical Environment*, as well as a set of *Properties* associated with all named elements, allowing to enrich the model with metadata.
- **Enhancement of the reasoning engine for automatic conflict detection and resolution.** A first enhancement of this reasoning engine results from the new version of the WIMAC metamodel. The transformation rules can be more expressive leading to more accurate and fine-grained transformations. In addition, a new mechanism has been added to facilitate the management of these rules, in particular, by breaking down the overall process into functional steps and organizing the rules accordingly.
- **New semantic-based reasoning engine for actuation conflict detection.** Elements of the new WIMAC metamodel can now be characterized by metadata. Thanks to these metadata, one can store semantic information based on an ontology which, along with a semantic model of the physical environment, can be used by a semantic reasoning engine to automatically detect indirect conflicts.
- **Enhancement of the tool for designing new custom and safe** actuation conflict managers. A domain specific language is now available to specify custom ACM and the properties it must comply with to be safe. The ACM description then includes:
    - The description of the behaviour logic of the ACM with a language close to ECA (Even-condition - Action rules).

- The description of the synchroniser (execution engine with temporal properties verifications) required to implement this logical behaviour (*i.e.*, specific configuration of this engine).
- The description of the logical properties the ACM has to comply to, which are further checked and evaluated by a state-of-the-art model checker.
- The description of the temporal properties the actuation conflict manager (synchroniser and logical behaviour) in its operational context has to comply with. The description uses a discrete event model for temporal simulations called DEVS [28].
- Finally, the tool supports the implementation of actuation conflict managers. From the specification of a custom ACM, a DEVS model is generated and accompanied by the supporting DEVS kernel (*i.e.*, the execution engine).

5.2 Actuation Conflict Management Enabler

In this section, we present the actuation conflict management enabler workflow and tools for automatic and semi-automatic actuation conflicts detection and resolution in SIS.

5.2.1 *New WIMAC Modelling Language V2*

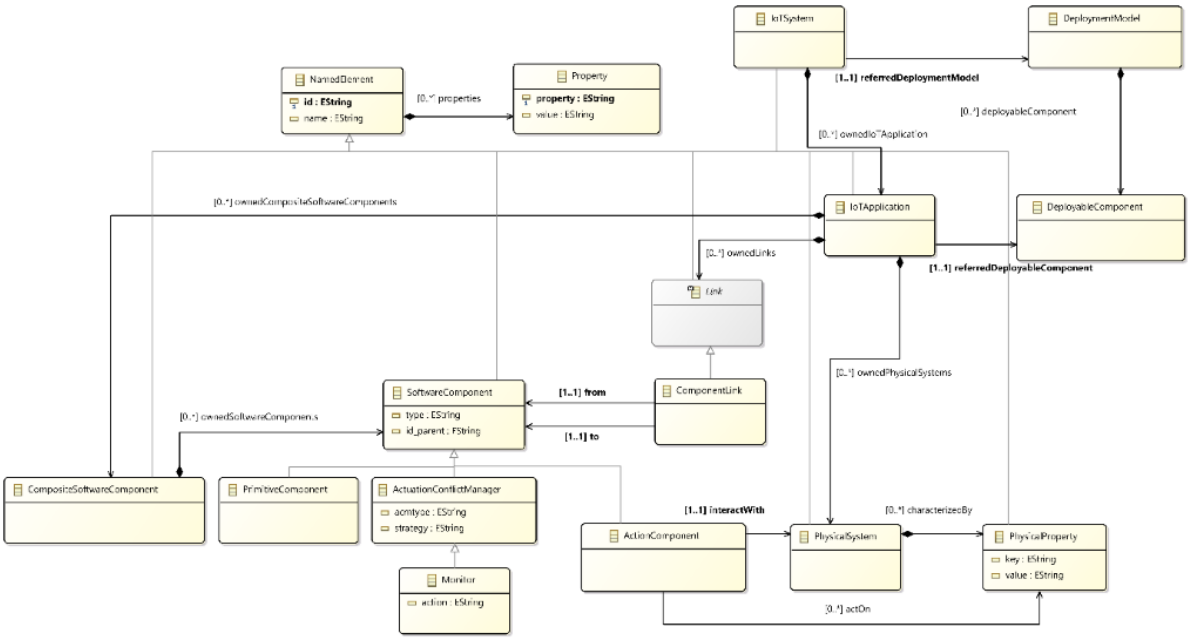


Figure 46: Meta-model of the WIMAC modelling language

The objective of the Actuation Conflicts Management tool is to support designers in *(i)* identifying actuation conflicts, *(ii)* designing custom ACMs or selecting off-the-shelf ACMs to solve these conflicts, and *(iii)* injecting these ACMs into the existing application. The WIMAC domain specific modelling language has been conceived as an abstraction, aggregating only the necessary concepts and facilitating all these activities. This includes details about the architecture and the deployment of the SIS (*i.e.*, relationships between software components, distinguishing those that interact with actuators and including where they are deployed) as well as their effects on the physical environment. It is worth noting that the WIMAC modelling language is technology and platform independent, meaning that applications are not bound to a specific programming language (at the time of writing, transformations from Node-RED and ThingML are implemented demonstrating this aspect). The same applies to the deployment model (*i.e.*, WIMAC is not bound to GeneSIS). In the following we describe the core concepts of the WIMAC meta-model.

An *IoTSystem* is composed of *IoT-Application*s. Each *IoTSystem* is associated with a *DeploymentModel,* which contains *DeployableComponent*s. By contrast, an *IoTApplication* is only associated with a single

DeployableComponent, mapping the WIMAC and GeneSIS components and providing details about where and how the *IoTApplication* is deployed. This information is important as an *IoTApplication* may evolve during the actuation conflict resolution process and may thus be redeployed. An *IoT-Application* is composed of *Physical-System*s and *Composite-SoftwareComponent*s, which encapsulates *SoftwareComponent*s and *Link*s.

SoftwareComponent represents a black-box piece of software (*e.g.*, a node in the Node-RED application). Contrary to the GeneSIS modelling language, the WIMAC *SoftwareComponent*s can be finer-grained as they do not necessarily need to represent deployable artefacts but may also represent their internal modules. A *Software-Component* can be an Actuation-ConflictManager representing a software module used to manage an actuation conflict. A *SoftwareComponent* can also be an *ActionComponent* in charge of controlling an actuator.

An *ActionComponent* can affect *PhysicalProperties*, which represents a physical phenomenon that may evolve over the time (*e.g.*, temperature in the kitchen). All the *PhysicalProperties* affected by an *ActionComponent* are logically grouped into a *PhysicalSystem*. This indicates that any *ActuationConflictManager* managing a *PhysicalProperty* needs to consider and manage the interactions with all the other properties in the *PhysicalSystem*.

It is worth noting that an *ActionComponent* can only interact with a single *PhysicalSystem*. The rationale for this design is the following. Even if one *ActionComponent* could interact with several *PhysicalSystem*s, only a single ACM can be added to the system to manage all of these interactions. For instance, a light may affect both the luminosity and the temperature of a room, generating one ACM for each property would not make sense as, in all cases, switching on or off the light always affect both properties. Instead, for devices equipped with several actuators, it is thus preferable to represent them with several *ActionComponents*.

A *link* represents a relationship between two *SoftwareComponent*s. L*ink*s are directed and are attached to one input and one output *SoftwareComponent*.

All elements of the WIMAC model are *NameElement*s except those dedicated to the deployment part. They are then associated with an *id*, a *name* and a set of *Properties*. These properties can provide metadata about any concept in a WIMAC model, which, for instance, can be useful during its different transformations and manipulations. More precisely, they can help to distinguish and classify more finely the instances of a same concept (*e.g.*, identifying whether or not a *SoftwareComponent* is part of a dataflow accessing an action component via the “conflict” tag metadata, see Section 5.2.2.4.

5.2.2 *Workflow with the new Tools of the Actuation Conflict Management Enabler V2*

In the following we present the transformations chain that drives the actuation conflict management process as part of a DevOps cycle. It consists of seven stages starting with the construction of a WIMAC model and ending with the redeployment of the SIS (see Figure 47).

Stages A to C aim at producing a WIMAC model that contains all the necessary information for the conflict detection and resolution. Stage D detects actuation conflicts and automatically resolves them, inserting ACMs automatically selected; stage E provides developers with the ability to design new ACMs to replace them, if needed. Finally, on the basis of the WIMAC model enriched with ACMs, stage F produces updated versions of the deployment and application models that are, in turn, used to trigger a (re)deployment. In the following subsections, we detail each transformation and illustrate them with the Section 2 motivating example.

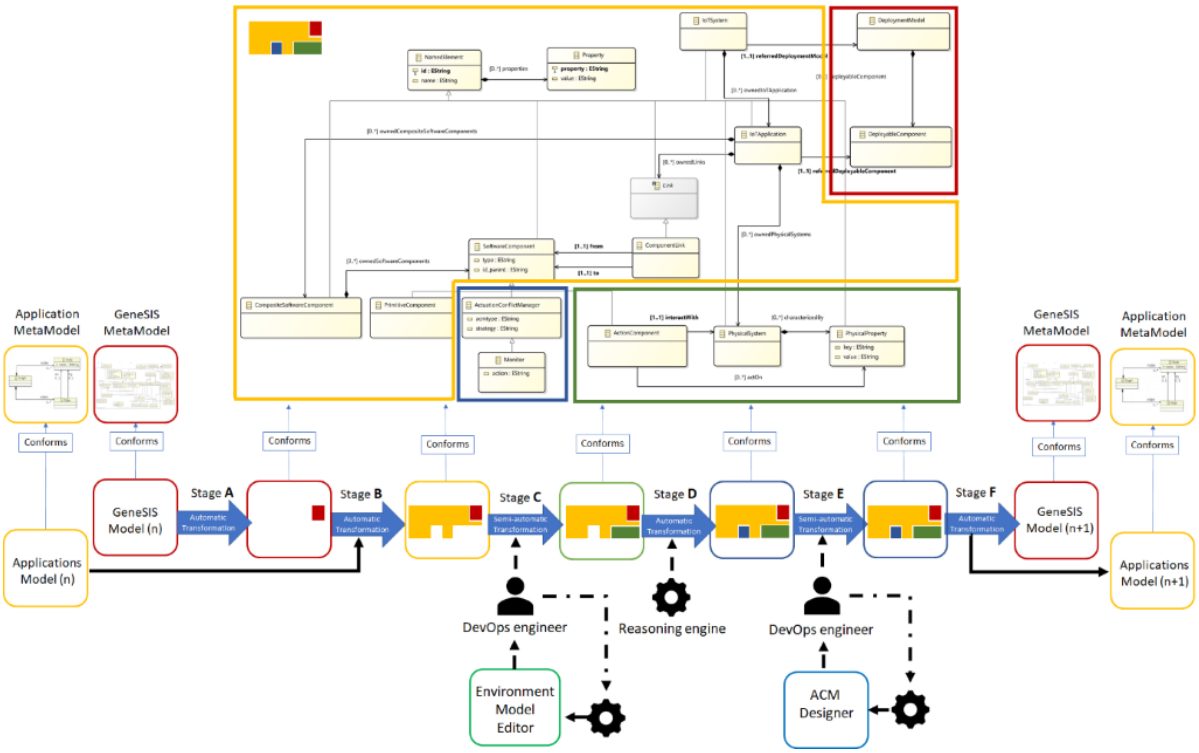


Figure 47: Workflow of the actuation conflict management enabler

5.2.2.1 Stage A: Genesis to WIMAC

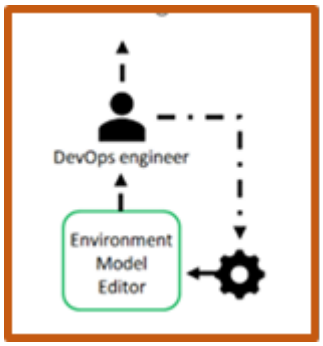
The objective of the first transformation is twofold: *(i)* bootstrapping the WIMAC model and *(ii)* adding to the model data related to the deployment of the system under consideration. These will be used at the end of the process (Stage F) to (re)deploy the SIS updated with ACMs. More precisely, the transformation results in the creation of an instance of *DeploymentModel*, which is composed of instances of *DeployableComponents* (cf. red area in the WIMAC meta-model depicted in Figure 47) in to the WIMAC model. As WIMAC follows a component-oriented approach, since this is the approach adopted by most of the tools for continuous deployment (including GeneSIS) [6], this transformation is rathert straight-forward and is fully automatic.

5.2.2.2 Stage B: Application Models to WIMAC

The deployment model also provides references to the different applications that compose the SIS (*e.g.*, from the deployment model, the endpoint of an application using Node-RED can be found and further used to retrieve the application model, a.k.a. the Node-RED flows, dynamically). More precisely, the transformation results in the creation of an instance of *ApplicationModel*, which is mainly composed of instances of *SoftwareComponents* (cf. part of yellow area in the WIMAC meta-model in Figure 46). Here again, as WIMAC follows a software component-oriented approach (including Node-RED and ThingML), this transformation is rather straight-forward and is fully automatic.

Deployment and Application Models will be used at the end of the process (Stage F) to (re)deploy the system evolved to manage actuation conflicts.

5.2.2.3 Stage C: Model of the indirect interactions between actuators through the physical environment



While direct actuation conflicts can be identified on the mere basis of the structure of an application, indirect actuation conflicts can be more subtle. They typically involve physical aspects making them difficult to identify. For instance, a ventilation system, by acting on the air flow, indirectly acts on temperature and humidity factors. A TV, understood primarily as an entertainment device, can also be understood as an actuator that impacts the acoustic atmosphere, luminosity and, to a lesser extent, the temperature in their physical environment. Thus, in addition to the detection of the concurrent accesses of several applications to a same actuator, the actuation conflict may appear by considering the interactions between devices through the physical environment.

In the WIMAC model *SoftwareComponent*(s) that drive an actuator, are *ActionComponent*(s). *ActionComponent*(s) are also linked to *PhysicalSystem*(s). They act on some associated *PhysicalPropert*(ies).

This stage then extends the WIMAC with the details about *(i)* the *PhysicalSystem* and the *PhysicalPropert*ies on which the system can act, as well as *(ii)* the relationship between the actuators and impacted physical properties.

In a first simple approach, this transformation is semi-automatic, and developers are provided with a tool to specify relationships between instances of *PhysicalSystem*(s) and existing *ActionComponent*(s) (see WIMAC Model in Figure 46).

These new relationships between *ActionComponent*(s) will be used to detect indirect actuation conflicts in the next stage.

5.2.2.4 Stage D: Automatic reasoning on WIMAC for conflict detection and resolution

The objective of this transformation is twofold: *(i)* detecting the actuation conflicts and *(ii),* for each conflict instantiating a default ACM into the WIMAC model (cf. blue coloured area in Figure 47). This transformation consists in the rule-driven graph rewriting of the WIMAC model. Inspired by the concept of pointcut and advice in Aspect-Oriented Programming [29], each rule consists in *(i)* a pattern representing a typical actuation conflict, which is used to identify the conflict, and *(ii)* a pattern specifying how to inject an ACM into the WIMAC model.

This transformation is fully automatic and is implemented using a graph rewriting engine called Attributed Graph Grammar (AGG[14]), which supports an algebraic approach for applying graph transformations and provides the ability to define graph patterns and their associated transformation rule in the form of an attributed graph (for example respectively left and right hand sides of the rule Figure 49a). The main rationale behind the choice of AGG is the following: *(i)* the AGG grammar used to define new transformation/rewriting rules is simple and the set of rules can be easily extended to handle new actuation conflict patterns, and *(ii)* it has demonstrated to be scalable, meaning transformations can be applied on large models.
However, AGG is limited by its strategy of applying the rewriting rules. Indeed, each rule is applied according to the order in the list of rules. Each rule modifies the graph and transmit it to the next rule for the next transformation. The AGG rules application engine is therefore very dependent to the order of the rules. As soon as the rules are numerous, transformations become difficult to manage. For ENACT, we have therefore implemented a step-by-step approach according to intermediate objectives in the graph transformations. This has the primary advantage of managing only a small number of rules

[14] https://www.user.tu-berlin.de/o.runge/agg/

for each step. We also developed a language (see Table 7) that allows to configure a more flexible application rules engine.
The new AGG rewriting process works then as follows. Rules are considered one after the other. Each rule <*ruleName*$_i$> is applied iteratively a maximum number of times <*iterationNumber*$_i$> or until it cannot anymore (<*iterationNumber*$_i$>="∞"). In the later no pattern is found in the WIMAC model. Each application of an AGG rule is then identified by the name of the rule and the maximum number of times it must be applied: (<*ruleName*>, <*iterationNumber*>). Sequences of application (→) of rules (<*rulesSequence*>) can also be applied a maximum number of times (<*iterationNumber*>) (See language in Table 7).

A simple grammar to describe the sequence of application of rules in a rewriting cycle is describe in Table 7.

<ruleName>::-<string>
<iterationNumber>::=<uint>| “∞”
<rule> = (<ruleName>,<iterationNumber>)
<rulesSequence>::= <rule>|<rule>→<rulesSequence>
<rulesFactor>::= <rulesSequence>|((<rulesFactor>), <iterationNumber>)

Table 7: Simple grammar to describe the sequence of application of rules in a rewriting cycle

At the time of writing, the detection process is composed of six steps of rewriting, each of them applies its own set of rules. This decomposition provides the following benefits: *(i)* rules with the same objective are grouped together and thus reused in different context (*e.g.*, rules for direct conflicts detection), and *(ii)* it ensures a proper order between the set of rules for a same.

The first step allows to initialize the current WIMAC model with tags that will allow to directly understand, which part of which application (*SoftwareComponents*) can have, close or not, an impact on an Action Component (*ActionComponent)*. The two rules used in this step do not modify the WIMAC model. However, they tag the Software Components by *(i)* finding Software Components that are directly impacting Action Component (see Figure …) *(ii)* and by spreading this tag to Software Components upstream, that have, a close or not, impact on Action Component (see Figure 48).

$$((tagSoftFromAction,1) \rightarrow (spreadSoftFromAction, \infty),\infty)$$

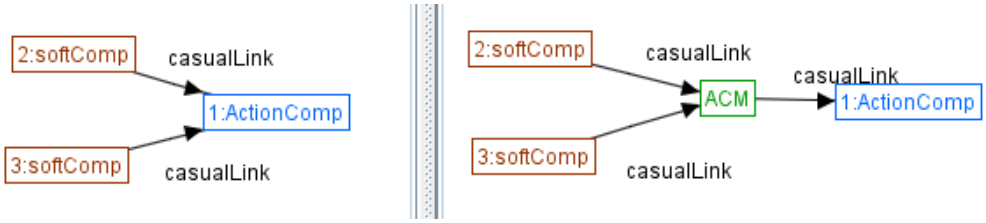
(a) Rule tagSoftFromAction to tag software components with direct impact on physical system

(b) Rule spreadSoftFromAction to tag software components with a farwer away impact on physical system

Figure 48: Rules for handling impact of software components to a close or not physical system.

The second step handles direct conflicts (Figure 49) and insert ACM when various software component access to a single *ActionComponent*. This step is composed by the following rules:

$$((\text{findDirectConflict},1) \rightarrow (\textit{multipleDirectConflicts},\infty),\infty)$$



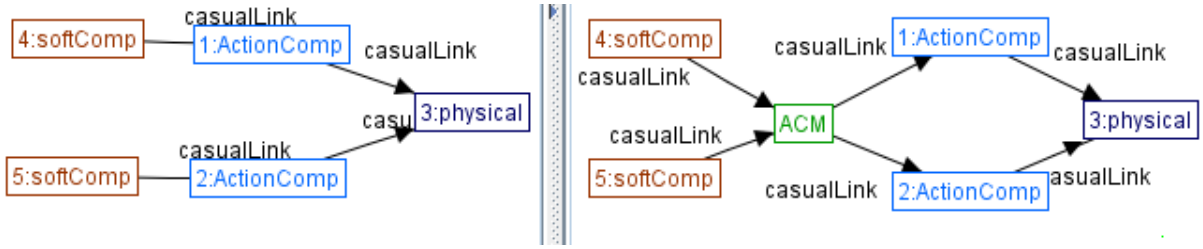
(a) Rule findDirectConflict for the direct conflict detection

(b) Rule multipleDirectConflicts for the direct conflict detection

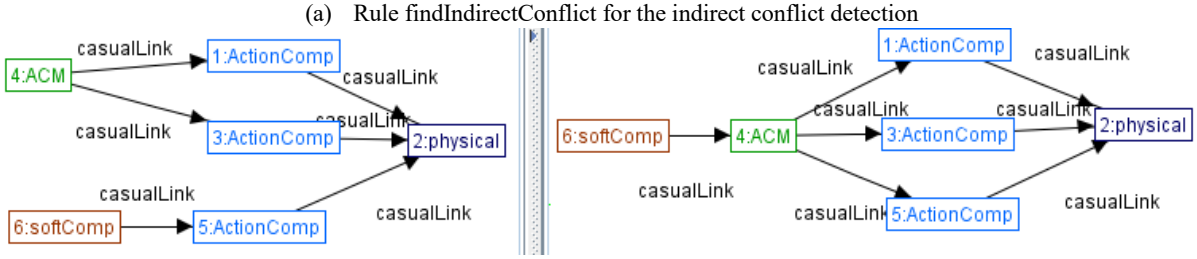
Figure 49: Rules for handling direct actuation conflicts

The third step handles indirect conflicts (see Figure 50) and insert ACM before various *ActionComponents*, when they act on a single *physical* system. This step is composed by the following rules:

$$((\textit{findIndirectConflict},1) \rightarrow (\textit{multipleIndirectConflicts},\infty),\infty)$$



(a) Rule findIndirectConflict for the indirect conflict detection

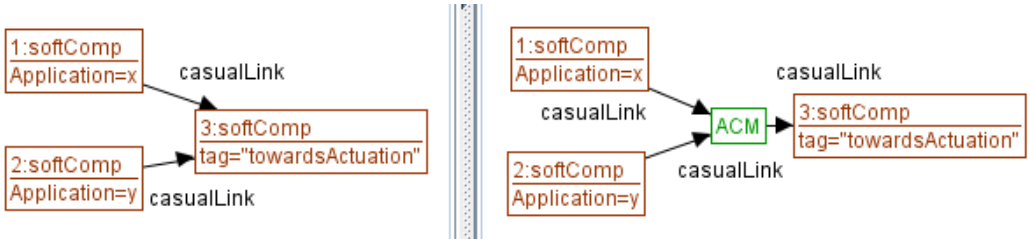


(b) Rule multipleIndirectConflicts for the indirect conflict detection

Figure 50: Rules for handling indirect actuation conflicts

The fourth step handles direct actuation conflict. Actuation conflicts can occur very early between application flows, as soon as at least two of them compete for access, close or not, to one or more actuators. We therefore introduce an attribute with the name of the application to which the software component belongs. This attribute then makes it possible to distinguish potentially contradictory impacts on the actuation from different application flows. We therefore define the following rules (Figure 51) which allow an ACM to be inserted when two or more software components of two different applications access a single software component tagged "towardsActuation".

$$((\text{findSoftConflict},1) \rightarrow (\text{multipleSoftConflicts},\infty),\infty)$$



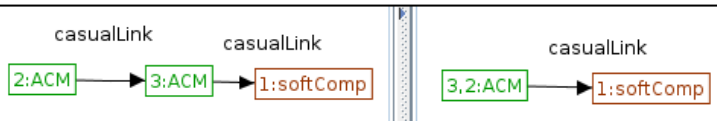
(a) Rule findSoftConflict for of conflict between separated application flows.

(a) Rule multipleSoftConflicts detection of conflict between separated application flows.

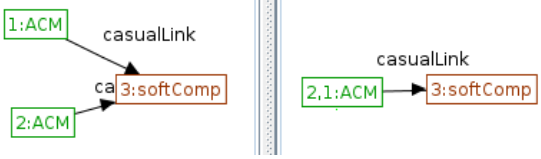
Figure 51: Rules for handling conflicts conflict between separated application flows.

The **fifth step merges close ACMs (**Figure 52**)**.
In the first transformation steps of the WIMAC model, the detection phase handles all types of conflicts instantiating numerous ACMs. In this step, all the ACMs that are sequenced or in parallel on a single single flow, are merged. This step is composed by the following rules:

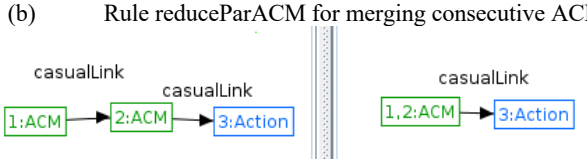
$$((reduceSeqACM, \infty) \rightarrow (reduceParACM, \infty) \rightarrow (reduceParActACM, \infty) \rightarrow (reducePaActACM, \infty)), \infty)$$



(a) Rule reduceSeqACM for merging consecutive ACM



(b) Rule reduceParACM for merging consecutive ACM



(c) Rule reduceSeqActACM for merging consecutive ACM

(d) Rule reduceParActACM for merging concurrent ACM

Figure 52:Rules for merging close ACMs

Finally, ***the sixth step replaces abstract ACMs by Monitors*** (see Figure 53). Monitors are a concrete default ACMs. They log the exchanged messages for further investigations at the actuation conflict points. For this step, other rules may insert other kinds of ACM. Developpers can also replace monitor by other off-the-shelf ACM.These rules are the following:

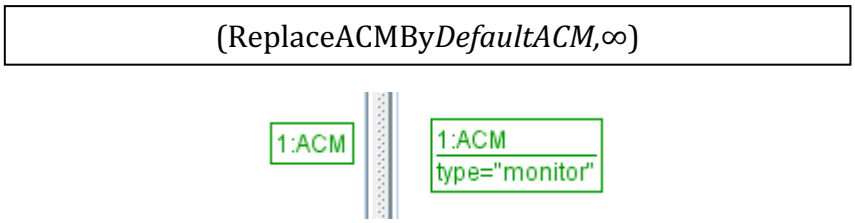
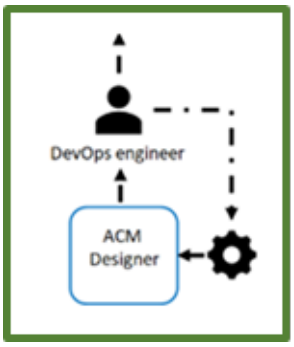


Figure 53: Rules ReplaceACMByDefaultACM for replacing abstract ACMs by concrete ones (Monitors by default)

5.2.2.5 Stage E: Designing new custom and safe ACMs



The former stages have resulted in the introduction of ACMs into the WIMAC model. At any time DevOps teams are provided with the ability to replace these ACMs by other ones. When none of the off-the-shelf ACMs fits a specific conflict, DevOps teams are provided with a tool for the design of custom and safe ACMs.

This tool offers a domain specific language called ECA+, that gathers all the information necessary for the design and the verification of a new ACM. This language can be used to describe: *(i)* the logical behaviour of the ACM, *(ii)* the logical and temporal properties to be satisfied and, *(iii)* an implementation strategy.

The workflow in Figure 54, details the different steps required when designing and checking a custom and safe ACM. Each step is detailed hereafter.

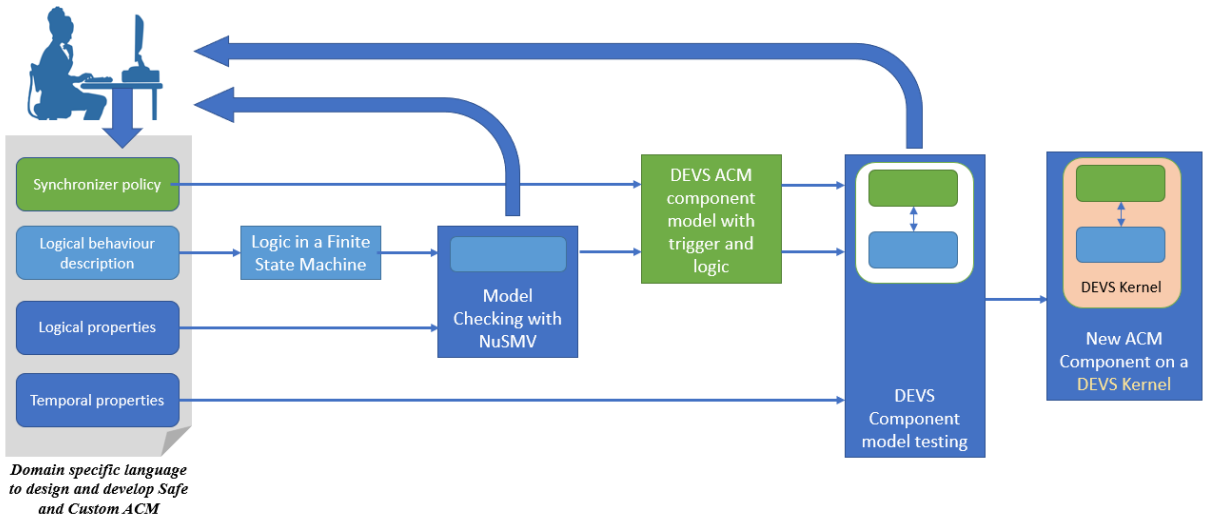


Figure 54: Workflow for designing and checking a custom and safe ACM.

ECA+: A domain specific languages to design and develop Safe and Custom ACM

The ECA+ domain specific language is then broken down into four parts.

The first part defines the synchronisation and triggering strategy of the logical behaviour of the ACM (see Figure 54 for the corresponding grammar[15]). Indeed, ACM logic consists in rules and the execution of these rules requires an execution engine and at least some conditions to trigger their execution.

[15] ECA Grammar using Pegjs: *https://gitlab.com/enact/actuationconflictmanager/-/blob/master/src/acm-custom-design/grammar/eca.pegjs*

```
SynchroConfiguration = "IF" _ SynchroCondition? _
    SynchroDelay? _ "TRIGGER"
SynchroCondition = InputCondition _ "THEN"
SynchroDelay = "DELAY" <delay> _ "ms"_ "THEN"
InputCondition = Input '||' InputCondition
               / Input '&&' InputCondition
               / '(' condition:InputCondition ')'
               / Input
```

Figure 55: Grammar for describing synchronisation and triggering strategy

The second part of the language describes the logical behaviour of the ACM (see Figure 55 for the corresponding grammar). Three sections describe the three main parts of the logical behaviour of the ACM. The first section (Managed Inputs Outputs in the Figure 55) defines the *inputs* and the *outputs* of the logical model. The second section defines and names a set of *predicates* to be evaluated on the inputs. The last section describes and names *actions* that correspond to specific combinations of outputs. Thanks to the set of predefined *predicates* and *actions* and their semantic the rules are simplified without using *inputs* and *outputs* references (see illustration in 5.2.3 section). The rules for describing the logical behaviour of the ACM are then easier to understand with a syntax close to the ECA (Event – Condition – Action) language (see illustration in 5.2.3 section). The main improvement is the addition of rules to describe not only output actions on conditions but also states changes. The logical behaviour description is then equivalent to a finite state machine (FSM).

```
// Managed Inputs Outputs
Input = "Input(" <inputPort> "," <index> ")"
Output = "Output(" <outputPort> "," <index> ")

// Predicates
PredicateConfiguration = <predicate> "=" predicateCondition
predicateCondition = InputCondition '||'
     right:predicateCondition
           / left:InputCondition '&&'
                right:predicateCondition
           / '(' condition:predicateCondition ')'
           / condition:InputCondition

InputCondition = (IntegerInput / StringInput)
IntegerInput = Input ("<="/">="/"=="/"!="/"<"/">")
     <integer>
StringInput = Input ("=="/"!=") <string

// Behavioural Rules
ECARules = "ON" <predicate> "IF" VariableCondition "DO"
     VariableAssignement* Action* ";"
VariableCondition = <variable> ("=="/"!=") <value>
                  / "True"
VariableAssignement = <variable> "=" <value>
Action = <action>

// Output Actions
ActionDefinition = <actionName> ":" OutputAssignement*
OutputAssignement = Output "=" ActionOperation
ActionOperation = Input ("+"/"-"/"*"/"/") ActionOperation
                / Input
                / <value>
```

Figure 56: Grammar for describing the logical behaviour of ACM

The third section describes a list of logical properties to be verified by state-of-the-art model-checkers such as NuSMV.

```
Properties = "ASSERT" "!(" PropertyCondition ")"

PropertyCondition = PropertyConditionMember '||'
    PropertyCondition
                  / PropertyConditionMember '&&'
                      PropertyCondition
                  / '(' PropertyCondition ')'
                  / PropertyConditionMember
PropertyConditionMember= (Input/Output)
    ("=="/"!="/"<"/">"/"<="/">=") (<integer>/<string>)
```

Figure 57: Grammar for describing the logical properties to verify

The fourth and the last sections describe some temporal properties to be verified in a simulated operation context. There is no specific grammar for this part as it only consists in choosing different controllers for detecting some defects during simulation.

ACM logical behaviour verification

Since the logical behaviour of the ACM is defined and equivalent to a finite state machine (FSM), a model checker allows to validate some logical properties on the behaviour of a custom ACM. The main goal of the model checker is to verify that these properties are always verified in every states of the ACM. Two main type of properties are classically verified:

- Safety: this kind of property ensures that something (bad) will never happen
- Liveness: this kind of property ensures that something (good) will eventually happen.

In our case, safety properties are verified to ensure that a conflicting state will never occur. The actuation conflict management enabler leverages NuSMV, a state-of-the-art model checker, to perform the model checking. NuSMV [30] allows for the representation of synchronous and asynchronous finite state systems as well as the analysis of specifications expressed in Computation Tree Logic (CTL) and Linear Temporal Logic (LTL), using BDD-based and SAT-based model checking techniques.
The model checking can provide two feedbacks: either properties are always true which means that there is no inputs sequence that leads to a conflict, or it stops on a sequence that invalidates a property.

ACM Synchroniser

The logical behaviour of an ACM is specified through a FSM. A FSM is only defined by two functions: *(i)* one to compute the new state (called state transition function) and *(ii)*one to compute the outputs (called output transition function), from the current state and current inputs . So, to be executed, a FSM needs what we call an execution engine used for triggering these functions after receiving inputs. In ECA+ there is a specific description for the trigger policy (see specific grammar in Figure 55). The ACM then consists in two entities: *(i)* a FSM to implement the logical behaviour of the ACM and *(ii)* a synchroniser that receives events from the application flows and triggers FSM execution with synchronised inputs. (see Figure 58). The synchronisation policy then has a great impact on the properties of the ACM (for example some events may be lost between synchroniser inputs and outputs and then not taken into account by the logical behaviour FSM).

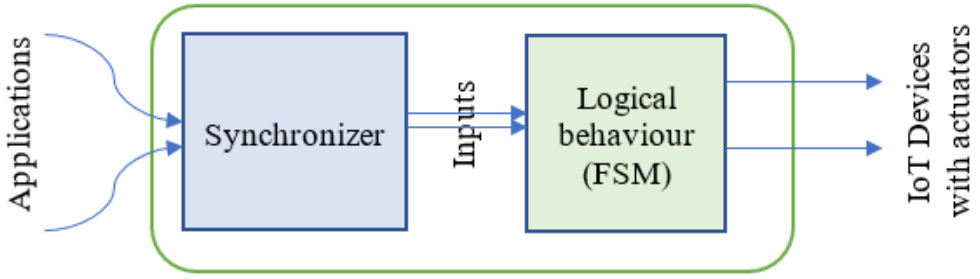


Figure 58: ACM model with a synchroniser

Temporal verification of the ACM and generation for various target platform.

Because of the impact of the synchroniser in the ACM behaviour, we must introduce a second validation step that is not only based on the logic of the ACM but also on its temporal behaviour. For investigating that we have chosen a discrete event modelling approach with the DEVS formalism. DEVS offers both a simulation environment with rigorous simulated time management but also great flexibility for implementing complex models that are free of any formalism [31].

- DEVS formalism:

DEVS defines two kinds of models: atomic models and coupled models. An atomic model specifies the dynamic of a component. It describes the behaviour of a component, which is indivisible, in a timed state transition level. Coupled models describe how to couple several models together to form a new model. This kind of model can be employed as a component in a larger coupled model, thus giving rise to the construction of complex models in a hierarchical fashion. As in general systems theory, a DEVS model contains a set of states and transition functions that are triggered by the simulator.

A DEVS atomic model AM with the behaviour is represented by the following structure:

$$AM = < X, Y, S, \delta_{int}, \delta_{ext}, \lambda, t_a >$$

where:

- $X : \{(p,v) | (p \in inputports,\ v \in Xph)\}$ is the set of input ports and values.
- $Y : \{(p,v) | (p \in outputports,\ v \in Y_{ph})\}$ is the set of output ports and values.
- S: is the set of states.
- $\delta_{int} : S \rightarrow S$ is the internal transition function that will move the system to the next state after the time returned by the time advance function.
- $\delta_{ext} : Q \times X \rightarrow S$ is the external transition function that will schedule the states changes in reaction to an external input event.
- $\lambda : S \rightarrow Y$ is the output function that will generate external events just before the internal transition takes places.
- $ta: S \rightarrow \mathbb{R}_o^+$ is the time advance function, that will give the lifetime of the current state.

The interpretation is the following:

- $Q = \{(s,e) | s \in S_h, 0 < e < t_a(s)\}$ is the total state set,
- e is the elapsed time since last transition, and s the partial set of states for the duration of $t_a(s)$ if no external event occurs.
- δ_{int}: the model being in a state s at t_i, it will go into s^0, $s^0 = \delta_{int}(s)$, if no external events occurs before $t_i + t_a(s)$.
- δ_{ext}: when an external event occurs, the model being in the state s since the elapsed time e goes in s^0, The next state depends on the elapsed time in the present state. At every state change, e is reset to 0.
- λ: the output function is executed before an internal transition, before emitting an output event the model remains in a transient state.
- A state with an infinite lifetime is a passive state (steady state), else, it is an active state (transient state). If the state s is passive, the model can evolve only with an input event occurrence.

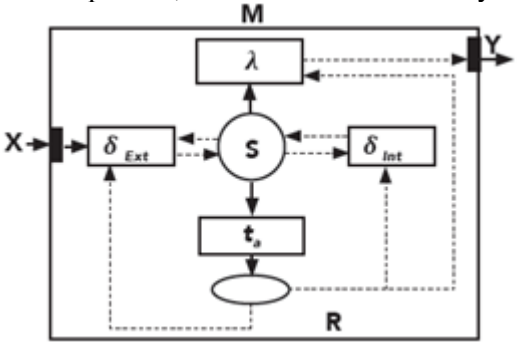


Figure 59: DEVS atomic model

Figure 59 (DEVS atomic model) describes the behaviour of a discrete-event system as a sequence of deterministic transitions between sequential states (S). The atomic model AM reacts depending on two types of events: external and internal events. When an input event occurs (X), an external event (coming from another model) triggers the external transition function $\delta_{ext}(X,S)$ of the atomic model in order to update its state. If no input event occurs, an internal event triggers the internal transition $\delta_{int}(S)$ of the atomic modelling order to update its state. Then, the output function $\lambda(S)$ is executed to generate the outputs (Y). ta(S) is the time advance function which determine the lifetime of a state.

The DEVS coupled model (CM) is a structure:

$$CM =< X,Y,D,\{M_d \in D\},EIC,EOC,IC >$$

where:

- X is the set of input ports for the reception of external events.
- Y is the set of output ports for the emission of external events.
- D is the set of components (coupled or basic models).
- M_d is the DEVS model for each $d \in D$.
- EIC is the set of input links, that connects the inputs of the coupled model to one or more of the inputs of the components that it contains.
- EOC is the set of output links, that connects the outputs of one or more of the contained components to the output of the coupled model.
- IC is the set of internal links, that connects the output ports of the components to the input ports of the components in the coupled models.

In a coupled model, an output port from a model $M_d \in D$ can be connected to the input of another $M_d \in D$ but cannot be connected directly to itself. The Figure 60 depicts the metamodel that allow to describe a DEVS coupled model from Atomic Models.

A simulator, called DEVSimPy [32], is associated with the DEVS formalism in order to execute instructions of coupled model to actually generate its behaviour.

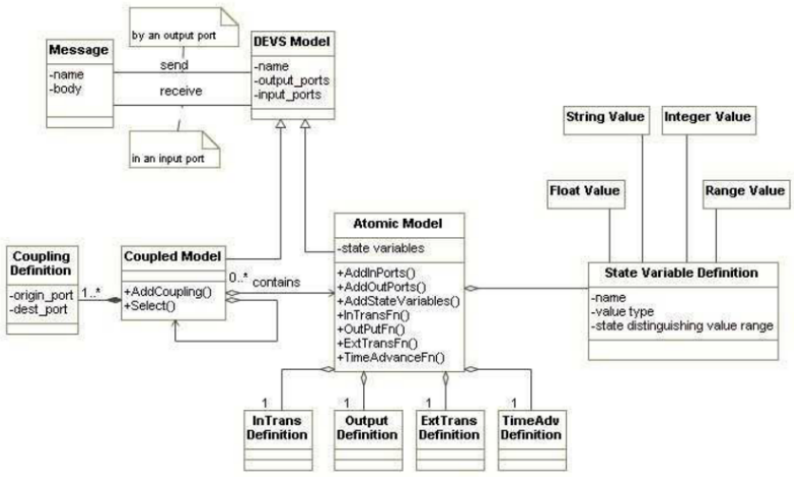


Figure 60: Metamodel to describe a coupled DEVS model

- Why DEVS:

Numerous works allow the projection of well-known formalisms (Moore and Mealy FSM, Petri Net, State Charts, SyncML etc. [33]) in DEVS formalism. For instance, if $t_a=\infty$, δ_{int} is inactive, and the DEVS atomic model is equivalent to a Moore finite state machine, where δ_{ext} is the Moore transition function and λ is Moore output function.

Nevertheless, DEVS model describe a timed event system that may be more elaborated. Because δ_{int}, ta, δ_{ext}, λ are functions in which any kind of algorithms can be implemented, DEVS may model more complex systems, at the risk of making unusable formal model checking approach.

In our example of a vacuum cleaner robot, fixing the duration of "cleaning state" at 100 simulated time steps is not a realistic hypothesis. Actually, to start with, it is better considering this duration as purely aleatory because it depends on an a priori unknown operating context (surface of the apartment, battery wear and autonomy, etc.). But numerous algorithms may be implemented in DEVS t_a function (even a learning algorithm) to compute a predictive mode [34] from successive experiments in a same apartment.

Finally, coupled DEVS models can be changed during simulation. Some atomic models may be added and removed. This property offers many opportunities for our actuation conflict management enabler that are not exploited for ENACT.

For all these reasons DEVS is the right formalism for testing ACMs before deployment.

- Overall DEVS model for ACM in an operating context:

An overall DEVS model for testing ACM consists in a set of atomic models.
The test of an ACM is then based on the description of its logical behaviour, a synchroniser to manage its logical behaviour model from the asynchronous messages received by the component:

1. The first atomic model is the *FSM DEVS model* which specifies the logical behaviour of the ACM. This FSM model is described by the developer in the previous step using ECA+. Its behavioural logic has been validated using model checkers such as NuSMV -*i.e.*, a set of logical properties have been checked. Converting a Moore machine model into an atomic DEVS model is quite straightforward. DEVS model of a Moore machine corresponds to ta=∞ which makes δ_{int} inactive. δ_{ext} is then the state transition function and λ the output transition function of the FSM.

2. The second atomic model corresponds to the *DEVS model of the synchronizer* which generates the FSM inputs according to asynchronous input events received by the ACM. Different synchronisers correspond to different strategies and therefore different models. At the time of writing, we distinguish between two synchroniser models. The first one is based on a strategy called ASAP (as soon as possible). It triggers an execution cycle of the FSM as soon as an input event is received. The second is based on conditional inputs and delayed triggering (ECDR). It triggers a FSM execution cycle, with the value of the received event, if a delay has elapsed without any other events occurred before. If another event occurs in the meantime, the FSM execution cycle is triggered with another value as input to the FSM. This strategy is typically implemented for cases such as the "double click" problem. The strategy presented here makes it possible to distinguish between a click and a double click of a mouse according to the time elapsed between two clicks.

For testing purposes, we need to simulate the operating context of the ACM implementation.

3. Events emitted by the different applicative flows correspond to *random events generators models*. Each random generator replaced each conflicting applicative flow. Because of the lack of a priori knowledge about the applications behaviour, random event emission is a reasonable hypothesis. Random generation is parameterized by variable start date, random occurrence within a given time range and range of random values.

4. Other DEVS atomic models may be added to complete the overall operating context. For example, *IoT devices* can be modelled for explicating the ACM impact on the devices behaviour and to verify some properties on their state changes. Physical Environment can also be modelled for verifying some physical properties evolution.

Finally, two addition al Atomic DEVS models are dedicated to the evaluation of the temporal behaviour of the ACM.

5. First of all, atomic models called message collectors, record the various messages exchanged between any atomic models of the global DEVS model.

6. Secondly, atomic models called controllers, receive and process various messages extracted throughout the global DEVS model to detect defects. A number of patterns and their DEVS

atomic models are defined a priori as the synchroniser models. Each pattern is defined in the ACM description.

A DEVS simulator called DevSimPy is then used to test the overall model.

ACM generation

Once the ACM model has been validated in a simulated operational context, the two coupled DEVS models (synchroniser and logical behaviour) can be directly used to implement the ACM. Indeed, the DEVS simulator is based on a lightweight DEVS kernel. This kernel is completely portable on the lightweight targets and platforms targeted by the project. For instance, we developed a DEVS kernel in Node.js allowing to implement an atomic DEVS model as a Node-RED node. When the simulated time of the DEVS kernel is driven by a timer, the execution of an atomic model corresponds to its execution with the system clock. Our ACM will thus be composed of two connected DEVS nodes, the synchroniser, and the logical behaviour with their atomic models.

5.2.2.6 Stage F: GeneSIS and Application models from WIMAC

This stage aims first at updating the application and deployment models on the basis of the changes introduced in the WIMAC model transformed throughout the previous stages. It consists in the reverse of Stage A and B transformations. These models are then used to trigger a (re)deployment of the SIS, only redeploying parts of the systems that have been modified.

The update of the applications basically lies in injecting the ACMs into the application and adapting the structure of the applications accordingly, by adding or removing links. To enable such an injection, ACMs must be implemented, or at least integrated, using the programming language of the applications. While off-the-shelf ACMs can be implemented in specific programming languages, custom ACMs are modelled with DEVS and implemented thanks to a DEVS kernel implementation in the programming languages of the targeted applications (*e.g.*, in the context of our motivating example, the ACMs models are executed on various instances of the DEVS kernel node in NodeRed). Thus, custom ACMs can be reused in multiple applications, as long as the DEVS kernel has been implemented on each target. Once the application is updated with ACMs and its structure adapted accordingly, its deployment model is updated. For each *IoTApplication* updated in the WIMAC model, the corresponding *InternalComponent* in the GeneSIS deployment model is marked as changed and its associated Resources are updated to point to the implementation of the application. Once this is completed, a new deployment is triggered.

5.2.3 *Smart Home Use Case Implementation*

In this section we detail and explain how the actuation conflict management enabler [35] was used as part of the Smart Home Scenario with “Communication Center” and “User Confort” applications interacting with a connected TV and a vacuum cleaner robot.

Context and naming convention

A new vacuum cleaner robot is added into the IoT Smart Space. A *UserComfort application* is modified with the objective to trigger cleanings at scheduled times. The modification of the *UserComfort application* results in the introduction of indirect actuation conflicts: indeed, the vacuum cleaner is quite noisy and may prevent the *CommunicationCenter application* from achieving its goal, for instance when the vacuum cleaner is running in a room holding a phone/conference call.

Smart Home use-case consists then in managing three applicative flows:

- *App_Rob of the UserComfort application*
- *App_Phone_TV* and *App_RemoteControl_TV* of the *CommunicationCenter application*

App_Rob controls the vacuum cleaner.
App_Phone_TV manages the ambient noise during a phone call by stopping the noise sources.
App_RemoteControl_TV controls the connected TV to insure the user comfort from a tablet PC.

The first one sends four commands: ROB_STOP, ROB_START, ROB_RESUME, ROB_PAUSE. The two others send the commands: TV_MUTE, TV_UNMUTE.

Then, we can illustrate two types of conflict here:

- a direct conflict on the commands sent to the Smart TV from the last two applicative flows.
- an indirect conflict on the "ambient noise level" property of the physical system between the three applicative flows, the robot and the TV being sources of noise.

To make our enabler usable in any DevOps ecosystem, we cannot assume that we have advanced knowledge of the semantics of the applications. Indeed, this would require specific annotations in the application models that are not necessarily provided by the DevOps team. Thus, only the name of the applications can allow the developer to identify their semantics and therefore to distinguish conflicts between different flows acting on the same device. In our illustration, for instance, only the name of the applicative flows allows the developer to differentiate the commands between the two conflicting applications, *App_Phone_TV* and *App_RemoteControl_TV* on the smart TV.

The ACM set up, after detection and resolution of conflicts (with fusion of two ACMs), manages 3 applicative flows that transmit TV_MUTE, TV_UNMUTE, ROB_START, ROB_STOP, ROB_SUMMARY, ROB_PAUSE, ROB_DOCK.

To distinguish events occurring in different flows, we prefix them with the name of the emitting application. For example, TV_MUTE events sent by the App_Phone_TV and App_RemoteControl_TV applications are denoted as App_Phone_TV.TV_MUTE or App_Remote_Control_TV.TV_MUTE.

Once the default ACM is instantiated at the point of conflict, the developer can replace it by a more specific off-the-shelf component or decide to design one using ECA+. Note that subsequently, device states will be named with "ing" when the state is active (*e.g.* ROB_CLEANING) and "ed" when the state is passive (*e.g.* TV_MUTED).

Step by step implementation of the Actuation Conflict Management Enabler

This use-case serves as an example to show the complete conflict management workflow.

- **Stage A: From Genesis model to WIMAC**

The first WIMAC model results in the creation of an instance of *DeploymentModel* which is composed of instances of *DeployableComponents*. The GeneSIS deployment model of the SIS is depicted in Figure 34. The initial version of the smart home system (deployment view) using the graphical syntax of GeneSIS. The whole deployment model can be found in the ENACT code repository[16].

- **Stage B: From Application Model to WIMAC**

Figure 61 represents the WIMAC model after the completion of stage B. Considering, for instance, the *UserComfort* application. The corresponding Node-RED flows in the *UserComfort* application are extracted to refine the WIMAC model. Each flow results in the creation of an instance of *CompositeSoftwareComponent* (see grey boxes in Figure 61) and, for each node, an instance of

[16] *https://gitlab.com/enact/actuation_conflict_manager/-/tree/master/demos/demo-openhab*

SoftwareComponents is created (see orange boxes in Figure 61). Finally, an instance of WIMAC Link is created for each Node-RED link.

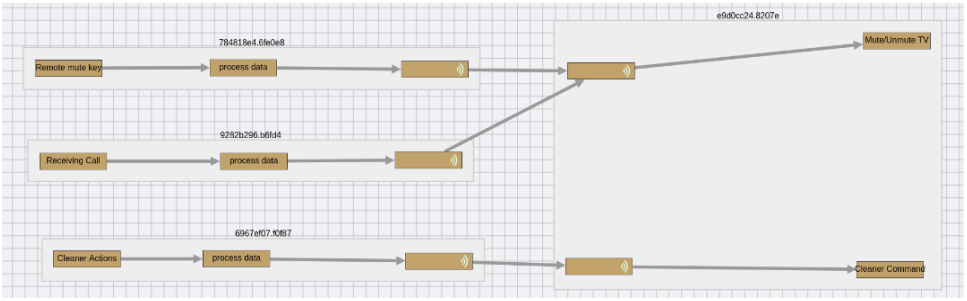


Figure 61: Extract from the WIMAC model after the instances of GeneSIS and Node-RED flows in the model.

- **Stage C: WIMAC and physical environment modelling**

In this third step, the *UserComfort* application is extended to manage the vacuum cleaner robot. The DevOps team extends the environment model by introducing *ActionComp*(s) and linking them to the concerned *PhysicalProperty* in the *PhysicalSystem* so as to enable indirect conflict detection. In the latter case, since the vacuum cleaner robot generates a lot of noise, its associated *ActionComp* is marked as acting on the Audio Physical Property in the Living Room.

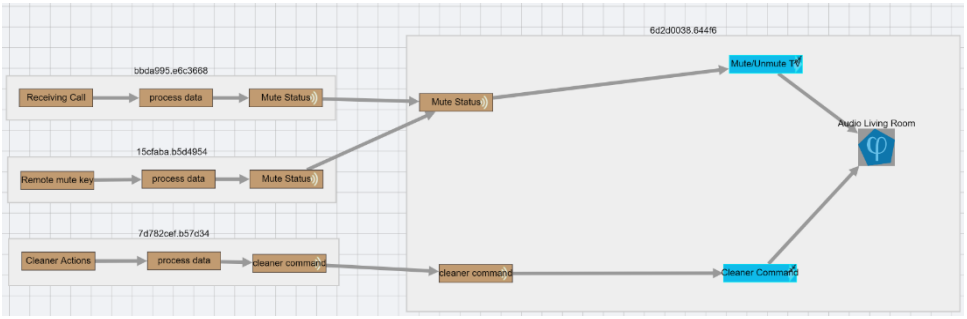


Figure 62: Extract from the WIMAC model after the instances of the physical systems in the model

- **Stage D: WIMACS transformations for actuation conflict resolution**

Considering the use-case, the vacuum cleaner robot and the Smart TV are indirectly conflicting since both impact the sound level in the living room. An excerpt of the WIMAC model representing this interaction is depicted in Figure 63.

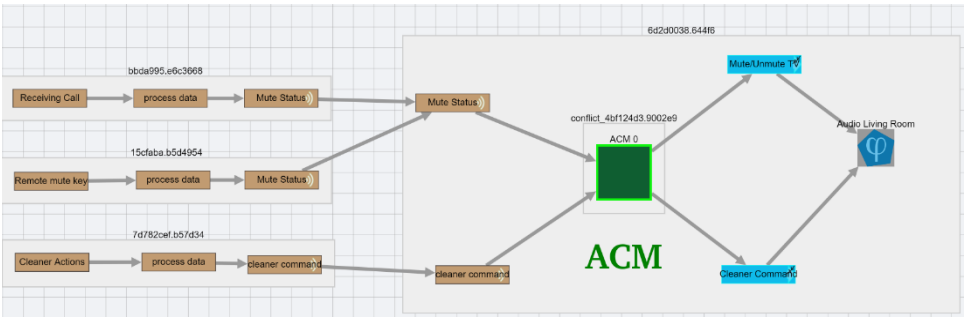


Figure 63: Excerpt of the WIMAC model after automatic conflict resolution

In this case, the transformation is performed as follows. All the rules are applied in parallel in order to find all the possible conflicts. The rule for the direct conflict on *ActionComponent*s does not find any conflict since the pattern is not matching. However, a compelling direct conflict was found between two applications and an indirect conflict was detected through the sound environment. Once all the conflict detection rules have been processed, the different *ActuationConflictManager* instanciated are merged

into only one resulting *ActuationConflictManage*. The final model for this general conflict detection is depicted in Figure 63.

- **Stage E: Custom Actuation Conflict component design and injection in WIMAC model**

When none of the off-the-shelf ACM can be used to manage the detected conflict, this stage allows to design of a custom and safe ACM.

As explained in Section 5.2.2.5, this tool relies on a domain specific language called ECA+, that gathers all the information necessary for the design and the verification of a new ACM. Considering the use-case one must introduce a new custom ACM that controls the smart TV and the vacuum cleaner robot, preventing an indirect conflict to occur through the sound physical property and this, whatever the commands coming from the different applicative flows of the *UserComfort* and *CommunicationCenter* applications.

```
// Synchroniser Strategy
TriggerCondition = Input(1,0) != null && Input (2,0) != null && Input (3,0) != null
IF TriggerCondition THEN TRIGGER
```

```
// Logical Behaviour Description

// Inputs definition
Input (1) = {"Clean" "Stop" "Resume" "Pause"}
Input (2) = {"mute" "unmute"}
Input (3) = {"mute" "unmute"}

// Inputs sequences definition
App_Rob.ROB_START = Input (1,0) == "Clean"
App_Rob.ROB_PAUSE = Input (1,0) == "Pause"
App_Rob.ROB_RESUME = Input (1,0) == "Resume"
App_Rob.ROB_STOP = Input (1,0) == "Stop"
App_Phone_TV.TV_MUTE = Input (2,0) == "mute"
App_Phone_TV.TV_UNMUTE = Input (2,0) == "unmute"
App_RemoteControl_TV.TV_MUTE = Input (3,0) == "mute"
App_RemoteControl_TV.TV_UNMUTE = Input(3,0) == "unmute"

// Predicates on Inputs

// ECA like rules

// States variables are State_TV and State_ROB where
// State_TV values are : TV_Muted, TV_Unmuted,
// State_ROB values are : Rob_cleaning, Rob_Stopped, Rob_Paused, Rob_Resumed
// Here some of ECA rules

ON App_Rob.ROB_START IF TV=="Muted" DO Rob="Cleaning" doROB_START;
ON App_Rob.ROB_STOP IF True DO Rob = "Stopped" doROB_STOP;
ON App_Rob.ROB_PAUSE IF Rob == "Cleaning" DO Rob = "Paused" doROB_PAUSE;
ON App_Rob.ROB_RESUME IF Rob == "Paused" DO Rob = "Resumed" doROB_RESUME;
ON App_Phone_TV.TV_UNMUTE IF Rob =="Cleaning" || Rob =="Resumed" DO Rob = "Paused"
doROB_PAUSE;
ON App_Phone_TV.TV_UNMUTE IF True DO tv="Unmuted" doTV_UNMUTE;
ON App_Phone_TV.TV_MUTE IF True DO phone="Muted" doTV_MUTE;
```

```
//Action Definition and corresponding Outputs
        ROB_START : Output (1) = "Clean"
        ROB_STOP : Output (1) = "Stop"
        ROB_PAUSE : Output (1) = "Pause"
        ROB_RESUME : Output (1) = "Resume"
        TV_MUTE : Output (2) = "mute"
        TV_UNMUTE : Output (2) = "unmute"
        default: Output (1) = "Stop", Output (2) = "mute"
```

```
// Logical Properties
ASSERT!( Output (1) == "Clean" && Output(2) == "unmute" )
```

```
// Temporal Properties
LivenessInTime (10,20)
NoLostEvent ()
```

Figure 64: Custom Actuation Conflict Description with ECA+

Verification of the ACM logical behaviour

The logical behaviour of the ACM is described in the green-coloured part of the ECA+ description. Input(i) are events coming from the different applicative flows *i*. These inputs (Inputs(i,j)) are renamed according to the rank *j* of their occurrence on a given flow. For instance, App_Phone_TV.TV_MUTE = Input (2,0) == "mute", corresponds to the first occurrence (counter 0) of the "mute" event from the applicative flow 2, called App_Phone_TV.

A more simple syntax than App_Phone_TV.TV_MUTE, will be used to facilitate ECA+ rules expressions. An example can be illustrated by the contradiction between App_Phone_TV.TV_MUTE and App_RemoteControl_TV.TV_UNMUTE commands. In this case, the resolution must provide a single command TV_MUTE or TV_UNMUTE to the smart TV.

"ON…IF …DO" rules are dedicated to the description of the resolution logic. They use state variables internally defined to memorize the current state.

For instance, the following rule means that when the input event ROB_START is transmitted from an applicative flow called App_Rob, and if the current state of the robot is ROB_STOPPED=on and the current state of the TV is TV MUTED=on then the new state of the robot is ROB_CLEANING=on, the TV state remains TV MUTED=on and a unique command ROB_START is sent to the robot (see ECA + rule in Figure 65).

```
ON App_Rob.ROB_START IF TV_MUTED=on ROB_STOPPED=on
    DO ROB_CLEANING=on TV MUTED=on  ROB_START;
```

Figure 65: Example of ECA+ rule in the Smart Home use-case ACM

Actions are finally translated by corresponding outputs. For instance, the ROB_START action corresponds to Output (1) = "Clean", further sent to the vacuum cleaner robot. This set of ECA+ rules are equivalent to the following Finite State Machine (Figure 66). As for a Moore machine each state sends its own outputs events, here corresponding to actions.

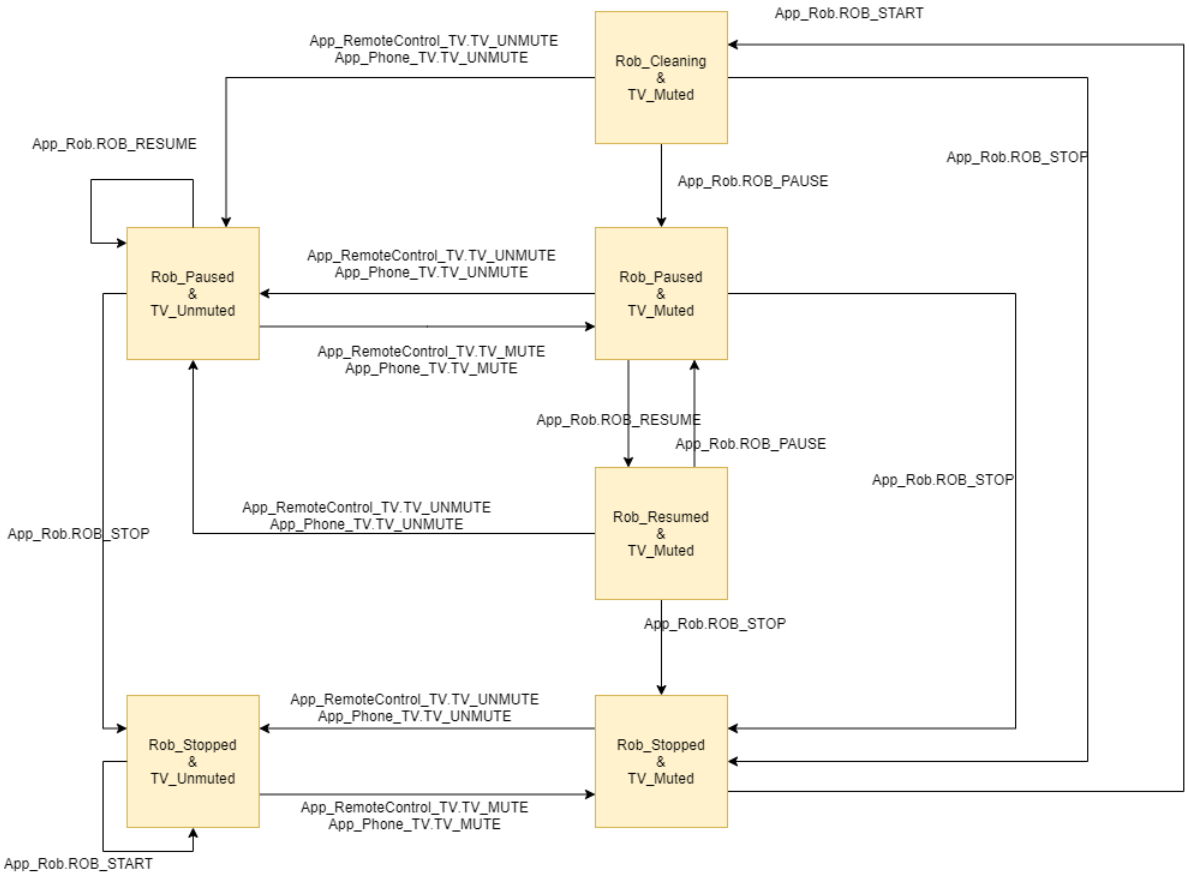


Figure 66: Finite State Model of the ACM logical behaviour

For the logical verification of the ACM, the corresponding FSM is used through NuSMV, a state-of-the-art model checker. In our example, the main property must comply with is "I don't want to send ROB_START when the last event sent to the TV is TV_UNMUTE" (see Figure 66).

```
// Logical Properties
ASSERT( Output (1) == "Clean" && Output(2) == "unmuted" )
```

ACM Synchroniser

As explained in section 5.2.2.5, finite state machine functions need to be computed on a regular basis. A synchroniser must contain condition on input events for triggering such a process. Considering the use-case, these conditions correspond to the blue-coloured part of the ECA+ description. Here a FSM cycle is triggered when at least one event occurred from each applicative flows (see Figure 67).

```
// Synchroniser Strategy
TriggerCondition = Input(1,0) != null && Input (2,0) != null && Input (3,0) != null
IF TriggerCondition THEN TRIGGER
```

Figure 67: Synchroniser Policy in Smart Use case ACM

Temporal verification of an ACM and generation for a target platform

A DEVS coupled model consists in a set of linked atomic models. Considering the use-case atomic DEVS models correspond to:

- The 3 applicative flows (App_Rob, App_Phone_TV, App_RemoteControl_TV) that are replaced by *random event generator models* for the simulation.

- A *synchroniser model* with a chosen policy for the ACM implementation (see an example of synchroniser that sends an input as soon as (ASAP) an event occurs in Figure 70). In the Smart Home use-case, the policy is more complex, as we described in section 2.
- The ACM *Logical behaviour model* (see Figure 71).
- The two *device models*: Smart TV and Vacuum Cleaner Robot (see Figure 68 and Figure 69).
- Some *message collector models* to collect data during simulation.

Various *controller models* to test temporal property defects during simulation.

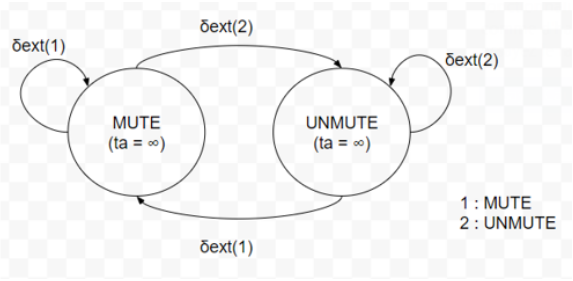


Figure 68: DEVS atomic model of the Smart TV

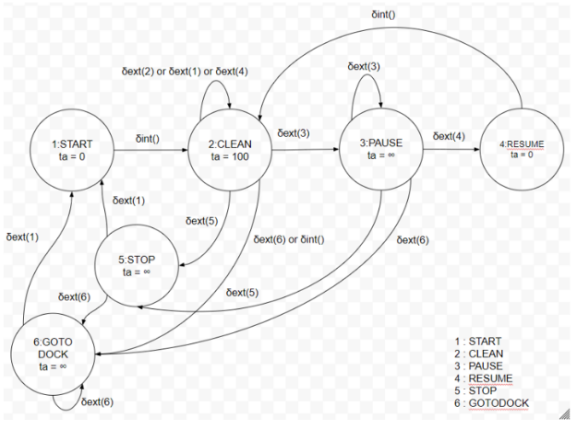


Figure 69: DEVS atomic model of the vacuum cleaner robot

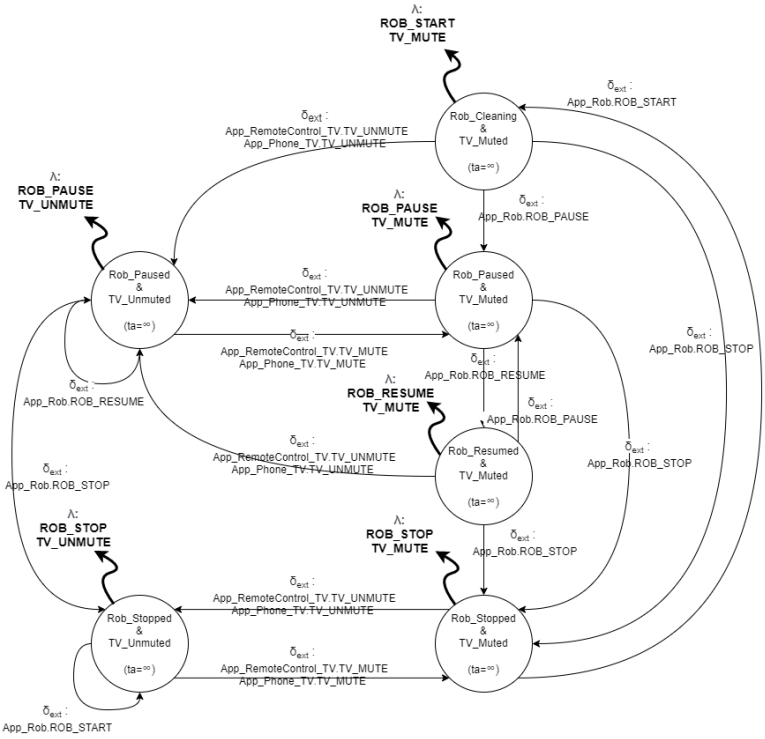


Figure 70: DEVS atomic model of an ASAP synchroniser

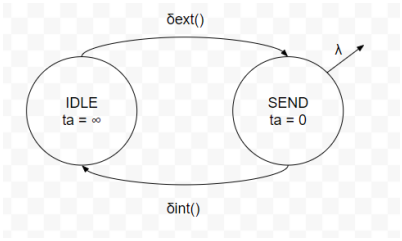


Figure 71: DEVS atomic model of the ACM logical behaviour

These atomic models are coupled into a coupled DEVS model for simulation in DevSimPy (see Figure 72). The first message collector models are used to track the information exchanged between the models. In our case, the first message collector model records the events emitted by the three random event generator models, called App_Rob_Gen, App_Phone_TV_Gen and App_RemoteControl_TV_Gen. In

our simulation, the message collector receives the outputs from App_RemoteControl_TV_Gen and provides the table and graph in the Figure 73. The table and graph the Figure 74 show the same thing with App_Rob_Gen.

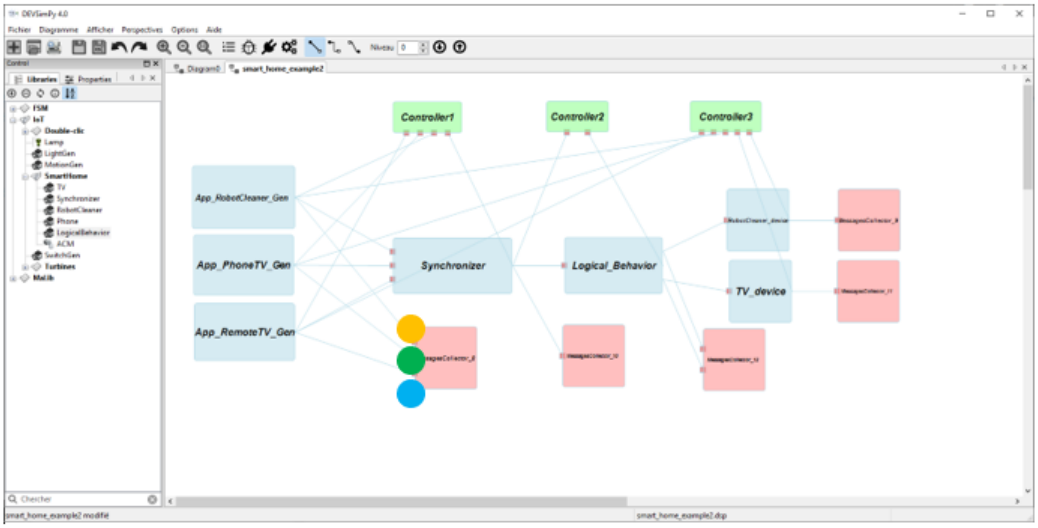
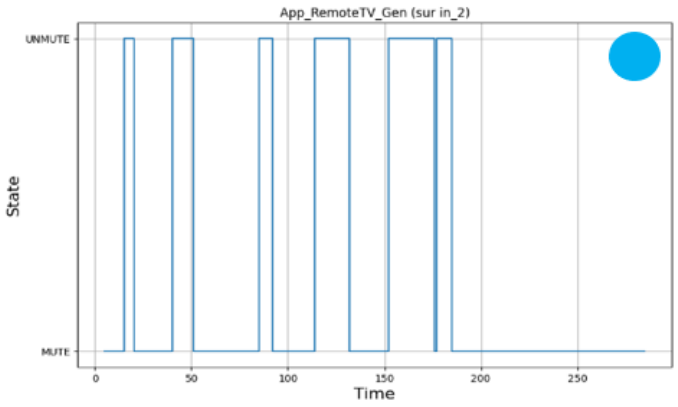


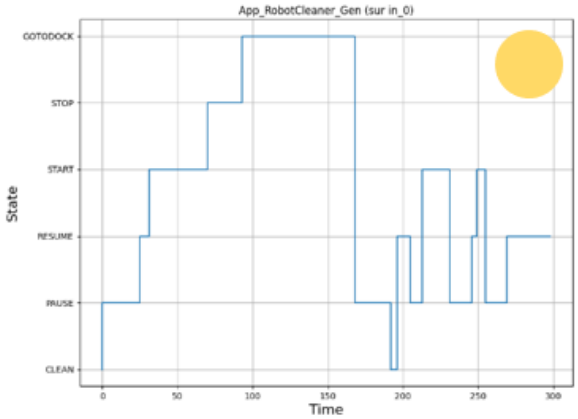
Figure 72: Overall DEVS model in DevSimPy in the Smart Home scenario



TV (sur in_0) | TV_Gen (sur in_1)

	Evénement	Message
1	0	<< value = MUTE; time = 9.0>>
2	1	<< value = UNMUTE; time = 21.0>>
3	2	<< value = MUTE; time = 23.0>>
4	3	<< value = UNMUTE; time = 32.0>>
5	4	<< value = MUTE; time = 45.0>>
6	5	<< value = UNMUTE; time = 91.0>>
7	6	<< value = MUTE; time = 121.0>>
8	7	<< value = UNMUTE; time = 126.0>>
9	8	<< value = MUTE; time = 144.0>>
10	9	<< value = UNMUTE; time = 160.0>>
11	10	<< value = MUTE; time = 177.0>>
12	11	<< value = UNMUTE; time = 205.0>>
13	12	<< value = MUTE; time = 270.0>>
14	13	<< value = UNMUTE; time = 274.0>>
15	14	<< value = MUTE; time = 312.0>>
16	15	<< value = UNMUTE; time = 361.0>>
17	16	<< value = MUTE; time = 377.0>>
18	17	<< value = UNMUTE; time = 386.0>>

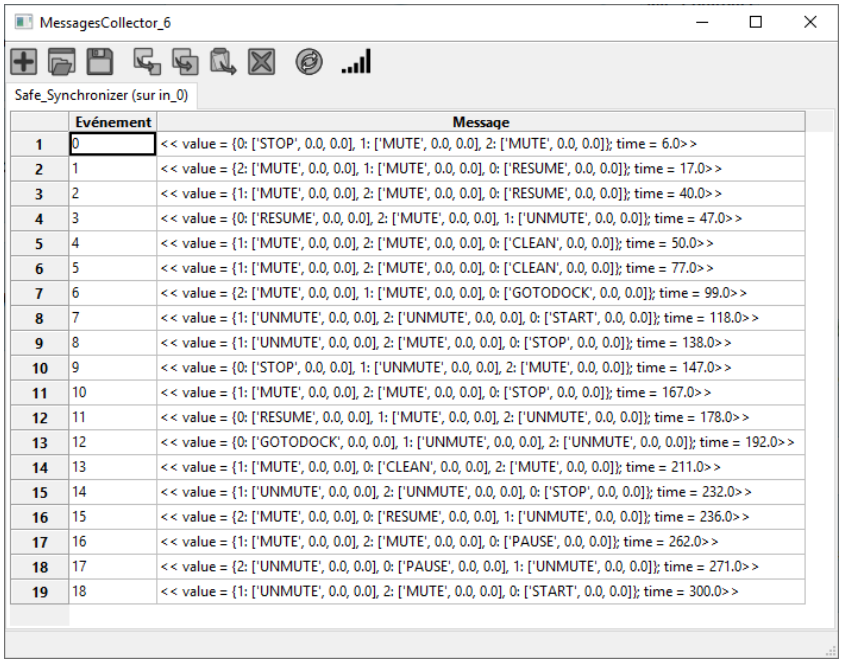
Figure 73: App_RemoteControl_TV_Gen outputs



RobotCleaner (sur in_0) | RobotCleaner_Gen (sur in_1)

	Evénement	Message
1	0	<< value = START; time = 111.0>>
2	1	<< value = CLEAN; time = 121.0>>
3	2	<< value = GOTODOCK; time = 121.0>>
4	3	<< value = START; time = 132.0>>
5	4	<< value = CLEAN; time = 142.0>>
6	5	<< value = GOTODOCK; time = 142.0>>
7	6	<< value = START; time = 166.0>>
8	7	<< value = CLEAN; time = 176.0>>
9	8	<< value = GOTODOCK; time = 176.0>>
10	9	<< value = START; time = 203.0>>
11	10	<< value = CLEAN; time = 213.0>>
12	11	<< value = GOTODOCK; time = 213.0>>
13	12	<< value = START; time = 289.0>>
14	13	<< value = STOP; time = 297.0>>

Figure 74: App_Rob_Gen outputs


MessagesCollector_6

Safe_Synchronizer (sur in_0)

	Evénement	Message
1	0	<< value = {0: ['STOP', 0.0, 0.0], 1: ['MUTE', 0.0, 0.0], 2: ['MUTE', 0.0, 0.0]}; time = 6.0>>
2	1	<< value = {2: ['MUTE', 0.0, 0.0], 1: ['MUTE', 0.0, 0.0], 0: ['RESUME', 0.0, 0.0]}; time = 17.0>>
3	2	<< value = {1: ['MUTE', 0.0, 0.0], 2: ['MUTE', 0.0, 0.0], 0: ['RESUME', 0.0, 0.0]}; time = 40.0>>
4	3	<< value = {0: ['RESUME', 0.0, 0.0], 2: ['MUTE', 0.0, 0.0], 1: ['UNMUTE', 0.0, 0.0]}; time = 47.0>>
5	4	<< value = {1: ['MUTE', 0.0, 0.0], 2: ['MUTE', 0.0, 0.0], 0: ['CLEAN', 0.0, 0.0]}; time = 50.0>>
6	5	<< value = {1: ['MUTE', 0.0, 0.0], 2: ['MUTE', 0.0, 0.0], 0: ['CLEAN', 0.0, 0.0]}; time = 77.0>>
7	6	<< value = {2: ['MUTE', 0.0, 0.0], 1: ['MUTE', 0.0, 0.0], 0: ['GOTODOCK', 0.0, 0.0]}; time = 99.0>>
8	7	<< value = {1: ['UNMUTE', 0.0, 0.0], 2: ['UNMUTE', 0.0, 0.0], 0: ['START', 0.0, 0.0]}; time = 118.0>>
9	8	<< value = {1: ['UNMUTE', 0.0, 0.0], 2: ['MUTE', 0.0, 0.0], 0: ['STOP', 0.0, 0.0]}; time = 138.0>>
10	9	<< value = {0: ['STOP', 0.0, 0.0], 1: ['UNMUTE', 0.0, 0.0], 2: ['MUTE', 0.0, 0.0]}; time = 147.0>>
11	10	<< value = {1: ['MUTE', 0.0, 0.0], 2: ['MUTE', 0.0, 0.0], 0: ['STOP', 0.0, 0.0]}; time = 167.0>>
12	11	<< value = {0: ['RESUME', 0.0, 0.0], 1: ['MUTE', 0.0, 0.0], 2: ['UNMUTE', 0.0, 0.0]}; time = 178.0>>
13	12	<< value = {0: ['GOTODOCK', 0.0, 0.0], 1: ['UNMUTE', 0.0, 0.0], 2: ['UNMUTE', 0.0, 0.0]}; time = 192.0>>
14	13	<< value = {1: ['MUTE', 0.0, 0.0], 0: ['CLEAN', 0.0, 0.0], 2: ['MUTE', 0.0, 0.0]}; time = 211.0>>
15	14	<< value = {1: ['UNMUTE', 0.0, 0.0], 2: ['UNMUTE', 0.0, 0.0], 0: ['STOP', 0.0, 0.0]}; time = 232.0>>
16	15	<< value = {2: ['MUTE', 0.0, 0.0], 0: ['RESUME', 0.0, 0.0], 1: ['UNMUTE', 0.0, 0.0]}; time = 236.0>>
17	16	<< value = {1: ['MUTE', 0.0, 0.0], 2: ['MUTE', 0.0, 0.0], 0: ['PAUSE', 0.0, 0.0]}; time = 262.0>>
18	17	<< value = {2: ['UNMUTE', 0.0, 0.0], 0: ['PAUSE', 0.0, 0.0], 1: ['UNMUTE', 0.0, 0.0]}; time = 271.0>>
19	18	<< value = {1: ['UNMUTE', 0.0, 0.0], 2: ['MUTE', 0.0, 0.0], 0: ['START', 0.0, 0.0]}; time = 300.0>>

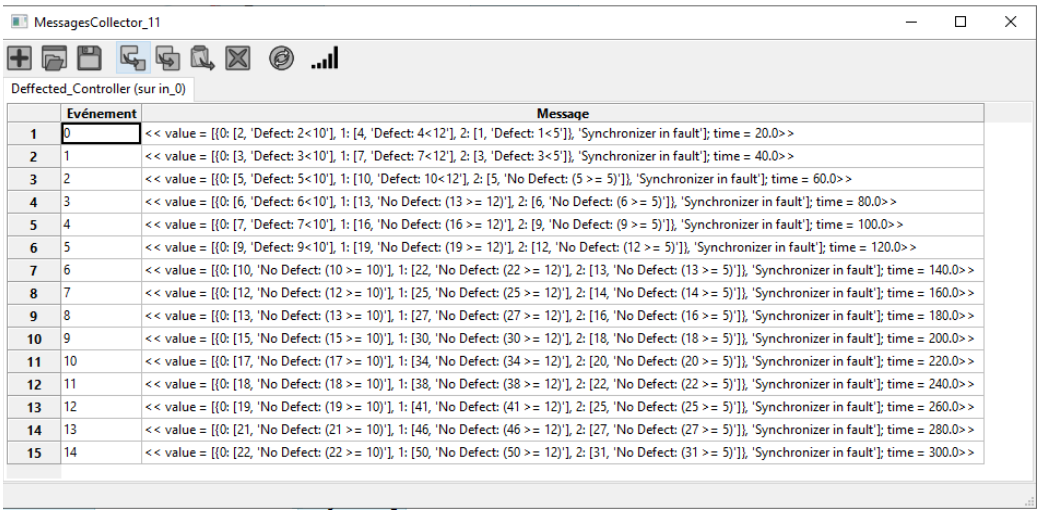
Figure 75: Synchroniser outputs

The properties can then be checked in the simulation with controllers that must detect defects. The first controller must check the applicative flows liveliness to make sure that they will never be temporarily blocked. We call this controller "LivenessInTime". Two parameters are needed. The <1st parameter> is the minimum number of events that each applicative flow must emit during a number (<2nd parameter>) of simulated time steps. The purpose of the second controller is to check that the synchroniser does not lose events by generating its outputs. In the ECA+ description of the Smart Home use-case, we monitor the LivenessInTime(10,20) and NoLostEvent () defects (see Figure 76).

```
LivenessInTime (10,20)
NoLostEvent ()
```

Figure 76: LivenessInTime and NoLostEvent properties in ECA+

We have tested different synchroniser to highlight some LivenessInTime and NoLostEvent defects that may appear. Indeed, without changing the logic of the ACM behaviour we can see that obviously the implementation of the ACM is also related to the synchroniser. The Figure 77 shows an example of synchroniser with defects.


MessagesCollector_11

Deffected_Controller (sur in_0)

	Evénement	Message
1	0	<< value = [{0: [2, 'Defect: 2<10'], 1: [4, 'Defect: 4<12'], 2: [1, 'Defect: 1<5']}, 'Synchronizer in fault']; time = 20.0>>
2	1	<< value = [{0: [3, 'Defect: 3<10'], 1: [7, 'Defect: 7<12'], 2: [3, 'Defect: 3<5']}, 'Synchronizer in fault']; time = 40.0>>
3	2	<< value = [{0: [5, 'Defect: 5<10'], 1: [10, 'Defect: 10<12'], 2: [5, 'No Defect: (5 >= 5)']}, 'Synchronizer in fault']; time = 60.0>>
4	3	<< value = [{0: [6, 'Defect: 6<10'], 1: [13, 'No Defect: (13 >= 12)'], 2: [6, 'No Defect: (6 >= 5)']}, 'Synchronizer in fault']; time = 80.0>>
5	4	<< value = [{0: [7, 'Defect: 7<10'], 1: [16, 'No Defect: (16 >= 12)'], 2: [9, 'No Defect: (9 >= 5)']}, 'Synchronizer in fault']; time = 100.0>>
6	5	<< value = [{0: [9, 'Defect: 9<10'], 1: [19, 'No Defect: (19 >= 12)'], 2: [12, 'No Defect: (12 >= 5)']}, 'Synchronizer in fault']; time = 120.0>>
7	6	<< value = [{0: [10, 'No Defect: (10 >= 10)'], 1: [22, 'No Defect: (22 >= 12)'], 2: [13, 'No Defect: (13 >= 5)']}, 'Synchronizer in fault']; time = 140.0>>
8	7	<< value = [{0: [12, 'No Defect: (12 >= 10)'], 1: [25, 'No Defect: (25 >= 12)'], 2: [14, 'No Defect: (14 >= 5)']}, 'Synchronizer in fault']; time = 160.0>>
9	8	<< value = [{0: [13, 'No Defect: (13 >= 10)'], 1: [27, 'No Defect: (27 >= 12)'], 2: [16, 'No Defect: (16 >= 5)']}, 'Synchronizer in fault']; time = 180.0>>
10	9	<< value = [{0: [15, 'No Defect: (15 >= 10)'], 1: [30, 'No Defect: (30 >= 12)'], 2: [18, 'No Defect: (18 >= 5)']}, 'Synchronizer in fault']; time = 200.0>>
11	10	<< value = [{0: [17, 'No Defect: (17 >= 10)'], 1: [34, 'No Defect: (34 >= 12)'], 2: [20, 'No Defect: (20 >= 5)']}, 'Synchronizer in fault']; time = 220.0>>
12	11	<< value = [{0: [18, 'No Defect: (18 >= 10)'], 1: [38, 'No Defect: (38 >= 12)'], 2: [22, 'No Defect: (22 >= 5)']}, 'Synchronizer in fault']; time = 240.0>>
13	12	<< value = [{0: [19, 'No Defect: (19 >= 10)'], 1: [41, 'No Defect: (41 >= 12)'], 2: [25, 'No Defect: (25 >= 5)']}, 'Synchronizer in fault']; time = 260.0>>
14	13	<< value = [{0: [21, 'No Defect: (21 >= 10)'], 1: [46, 'No Defect: (46 >= 12)'], 2: [27, 'No Defect: (27 >= 5)']}, 'Synchronizer in fault']; time = 280.0>>
15	14	<< value = [{0: [22, 'No Defect: (22 >= 10)'], 1: [50, 'No Defect: (50 >= 12)'], 2: [31, 'No Defect: (31 >= 5)']}, 'Synchronizer in fault']; time = 300.0>>

Figure 77: Synchroniser defects

ACM generation for a target platform

As explained in section 5.2.2.5, the DEVS kernel used for running an atomic model is lightweight and easy to implement on various targets. Considering the use-case, we have implemented a node of the DEVS kernel into Node-RED. The ACM then consists of two linked nodes, one for the synchroniser model and one for the logical behaviour model.

- **Stage F: New Genesis and Application models extraction from WIMAC**

Regarding the use-case, only one instance of *IoTApplication* has been modified. This instance is referenced in the *DeploymentComponent* instance of a WIMAC *DeploymentModel*. The reverse transformation of stages A and B, for extracting the GeneSIS deployment model and the Node-RED application model, is de facto very simple. It is even possible to modify the IoT application by adding or removing nodes. An excerpt of the modified application in our implemented use-case is depicted in Figure 78.

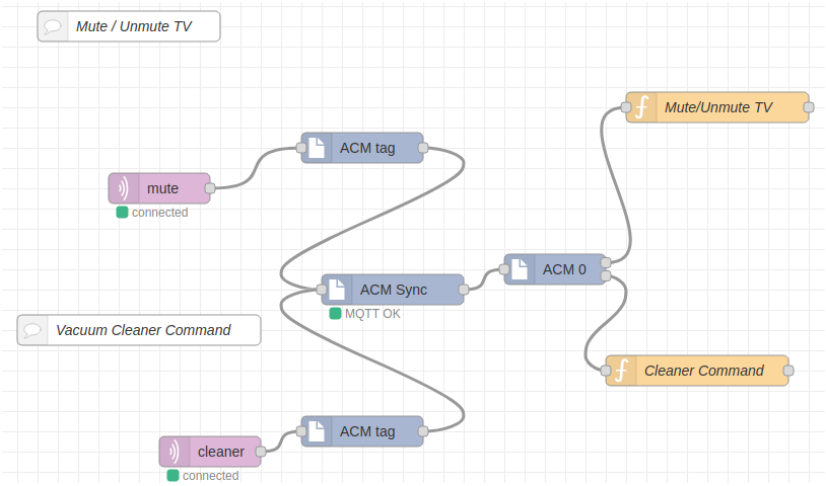


Figure 78: Excerpt of the modified application by the actuation conflict management enabler

5.3 Evaluation

5.3.1 *KPIs & Requirements*

The following section details how our approach addresses the requirements defined in Section 4.3 of deliverable D2.1 specified throughout two use-cases.

Table 8: Actuation conflict management requirement achievements

Req. ID	Req. Type	Comments	Coverage
R1	Accuracy	Modelling tool for a comprehensive actuation system integrating models of the physical environment and its evolutions according to the interactions with competing applications.	**Covered.** Actuation conflict management enabler V2 is a chain of tools with a step by step design methodology. Enabler has been enhanced for reusing custom ACM and for validating FSM execution engine before deployment.
R2	Usability	Shall integrate a GUI tool for actuation modelling support, based on simple and interpretable modelling frameworks.	**Covered.** Large-scale Actuation Conflict Management uses AGG tool. This tool provides a simple GUI for instantiating off-the-shelf ACMs into the complete model

			of SIS from transformation rules. Moreover, custom ACM modelling can be achieved using a user-friendly Extended ECA rules.
R3	Trustworthiness (reliability)	Shall help identifying complex environmental actuation conflicts.	**Covered**. Actuation conflict management enabler V2 provides an actuation conflict management considering indirect conflicts that may occur through a shared physical environment.
R4	Reusability	Shall permit the reusability of solutions already designed for similar cases.	**Covered**. The current actuation conflict management is based on off-the-shelf ACMs. It allows to modify, remove or add new ACMs on the go.
R5	Trustworthiness (reliability)	Shall aid in the design of the conceptual model of the conflict's controller (*e.g.*, formal test & verification).	**Covered**. The formal modelling framework underlying ACMs is a finite state automaton. Thereby, it can be validated thanks to classical model-checker like NuSMV.
R6	Trustworthiness (reliability)	Shall provide tools for testing conflicts co ntroller through an operational model (inte nded to validate conflict resolution solutio ns in an operational context).	**Covered.** This mainly depends on WP3 behavioural drift analysers used to detect unexpected behaviours from observations in the physical environment despite actuation conflict management.
R7	Trustworthiness (reliability)	Shall manage actuation conflicts despite bl ack box modelled components.	**Covered**. Actuation conflict management enabler V2 manages actuation conflicts despite black box components (see WIMAC model).
R8	Adequacy	Conflicts resolution at runtime shall be aut omated as much as possible.	The actuation conflict management enabler is a tool for the Devs phase. Alone, it cannot solve a conflict at runtime and therefore at Ops time. However, combined with D3.3 enablers this requirement is covered. Indeed, thanks to D3.3 enablers, any behavioural drift due to a conflict that is not or partly managed is detectable and diagnosable at runtime. A Devs cycle can be triggered then with a reuse of the actuation conflict management enabler at a new Devs time (adaptation) or thanks to online learning enabler at Ops time (self-adaptation).
R9	Trustworthiness (safety)	Shall provide actuation conflicts alerts dur ing the deployment on ENACT platform.	**Covered.** Actuation conflict management enabler V2 automatically detects actuation conflicts before deployment and provides developers tools to solve it and then deploy updated applications.
R10	Monitoring & trustworthiness (safety)	Shall continuously monitor behavioural dr ift to assess deployed solutions.	**Covered.** This mainly depends on WP3 behavioural drift used to detect unexpected behaviours from observations in the physical environment despite actuation conflict management.
R11	Monitoring & traceability	Shall trig a new development cycle from t he quantitative behavioural drift assessme nt value/threshold.	**Covered.** The behavioural Drift analyser can be used to trigger a new development cycle involving the Actuation Conflict Manager, thanks to the integrated ENACT platform.
R12	Scalability	Shall provide tools allowing to manage hu ndreds of sensors and actuators, thus tens o f actuation systems.	**Covered**. Large scale Actuation Conflict detection and resolution is one of the focus of the actuation conflict management

			enabler V2. Test has been done throughout evaluations (in lab and within use-cases).
R13	Scope	Actuation conflicts management tools shall support several targets ranging from IoT, Edge to cloud.	**Covered.** Leveraging on GeneSIS and WIMAC models, the actuation conflict management is compatible with different targets ranging from IoT, Edge to Cloud. In V2, ACMs are instantiated as nodes in Node-RED middleware embedded in a Docker container (covering edge and cloud) or as Thing with the help of ThingML for Arduino (covering IoT).
R14	Integration	Actuation conflicts management tools shall support different kind of frameworks (GeneSIS, ThingML, Node-RED) and middleware (SMOOL, SOFIA2, etc.).	**Covered.** In V2, ACMs are instantiated as nodes in Node-RED middleware and as Thing in ThingML middleware. ACMs are deployed thanks to GeneSIS. SMOOL/SOFIA middleware has been addressed by generating a custom ACM that can be integrated to SMOOL thanks to the GeneSIS/ThingML and SMOOL integration.

The following table addresses the requirements from the use-case providers described in deliverable D1.1.

Table 9:Actuation conflict management requirements from use-cases

Req. ID	Use Case	Comments	Coverage
DO-3.3.1 **DO-3.3.4**	INDRA, Rail Use Case	The orders must be prioritized	**Covered.** In V2, ACMs can be designed to prioritize orders for actuation system.
DO-3.3.2	INDRA, Rail Use Case	The Context Monitoring and actuation conflict management enabler and the Monitoring must have a GUI	**Covered.** GUI of actuation conflict management V2 and BDA enablers.
DO-3.3.3	INDRA, Rail Use Case	Low delays of alerting to the rail operator in order to avoid critical accidents	**Covered**. In V2, ACMs can be designed as Finite state machine (FSM), allowing response time to be kept within acceptable delays.
DO-3.3.6	TECNALIA, Smart Building use-case	The actuation conflict management enabler should be able to identify and avoid conflicts in colliding commands sent to same actuator by two IoT apps.	**Covered.** In V2, direct actuation conflicts are detected and solved.
DO-3.3.7	TECNALIA, Smart Building use-case	The actuation conflict management enabler should be able to identify and avoid conflicts in commands sent to actuators impacting the same physical variable, by two IoT apps at the same time.	**Covered.** In V2, indirect actuation conflicts are managed, considering counterproductive effects that may be produced by different actuators sharing a common environment.

5.3.2 *Other evaluations*

Another evaluation of the WIMAC Modelling Language has been organized during a hackathon. The results obtained are detailed hereafter. The first questionnaire about the WIMAC modelling language started with an introduction to the main concepts of the actuation conflict management enabler. The average time answering the question was 14:00 minutes.

5.3.2.1 Evaluation of the Actuation Conflict Management Enabler

In the first question, the participants were asked to identify the types of all the components involved in the WIMAC model shown in Figure 79, written using the graphical concrete syntax.

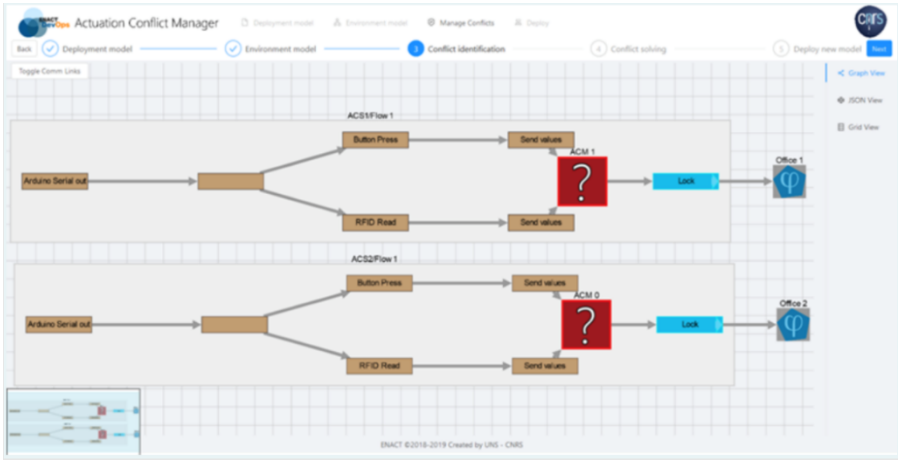


Figure 79: WIMAC model used in the questionnaire

As depicted in Figure 80, in average, *SoftwareComponent*s were successfully identified by 85.6% of the participants (average of the percentage of good answers for all software components with a standard deviation between the components of 1.5). In average, *ActuationConflictManager*s were successfully identified by 94.3% of the participants (average of the percentage of good answers for all actuation conflict managers with a standard deviation between the components of 3.8). In average, *PhysicalSystems*, were successfully identified by 97% of the participants (average of the percentage of good answers for all physical systems with a standard deviation between the components of 4). Finally, in average, *ActionComponent*s were properly identified by only 73% of the participants.

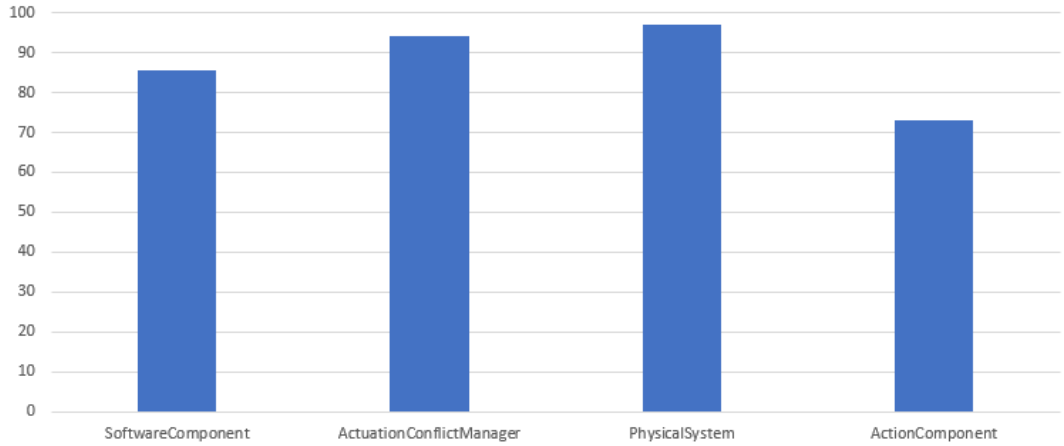


Figure 80: WIMAC model concepts understanding

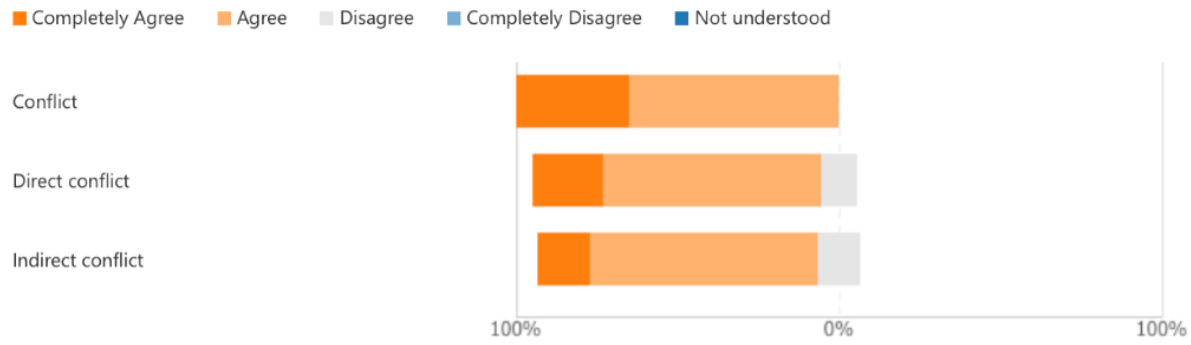


Figure 81: Intuitiveness of the concepts of direct and indirect conflicts - Likert scale

Overall, these results confirm the acceptability of the WIMAC concepts and their representation in the graphical syntax but improvements could be considered to make the semantic of the elements depicting *(i)* the type of conflicts and *(ii)* of the *ActionComponent* more intuitive.

Following the same approach than for GeneSIS, after the hands on with the actuation management tool, the participants were asked to fill the same questionnaire once more. In general, we only observe a significant improvement in the participants' ability to identify the WIMAC concepts reaching almost 99% of correct answers. Yet only 54% of the participants were able to properly identify the indirect actuation conflict.

5.3.2.2 Evaluation of the Integration Between GeneSIS and Actuation Conflict Management Enabler

The integration of GeneSIS and the Actuation conflict management tool is perceived by 51% of the participants as very good and by 43% as good and participants clearly see how our tools can be integrated with typical DevOps solutions such as for Debugging of IoT, Edge, and Cloud applications (31 answers), Risk Management (17), Security Monitoring (17), Test and Simulation of IoT systems (22). Overall, the participants clearly see the benefit of using our solutions compared to manually performing similar activities.

Regarding the different models used in our approach and their integration, we asked participants if they understood the need for the different models and why they are used. As depicted in **Error! Reference source not found.**, overall, the results are positive.

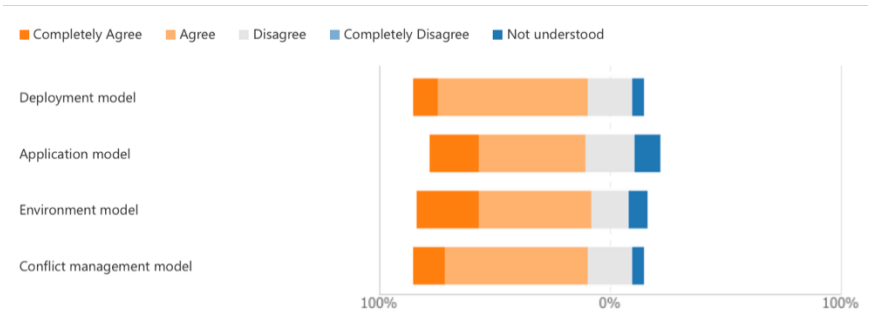


Figure 82. Understanding the role of the models used by GeneSIS and ACM

Finally, participants were asked to suggest improvements for both the tools and the languages. The main recommendations are:

- Improve the logging mechanisms with more extensive details about the deployment process, and in particular in case of failures. User interface should be extended to easily understand, for each component, what was successful or not.

- Provide mechanism to easily rollback a deployment or part of a deployment from the user interface (*i.e.*, not by providing a new deployment model).
- Provide mechanism to easily rollback modifications in ACM tool (modifications on the WIMAC model, the environment model and the edition of a Custom ACM).
- Provide a user-friendly classification of ACM to facilitate the search into the existing ACMs library. A corollary is the construction of a generic library of conflict managers.

5.4 Beyond ENACT

Multiple perspectives and future works potentially emerge from actuation conflict management research in ENACT. Here are described some of these:

- Firstly, actuation conflict management enabler is limited to the management of direct and indirect conflicts at the time of development within a predefined model of a presupposed operational context. However, many perturbations and unforeseen changes in the physical environment require more reactive adaptation. For this, the actuation conflict management enabler needs to be complemented by a tool at operating time to continuously learn the model of the changing physical environment. Thanks to an up to date physical environment model, the ACM tool will be able to potentially trigger again a new automatic or semi-automatic phase of conflict resolution. Behavioural drift analysis and root cause analysis in D3.3 should be usable as feedbacks for that purpose.
- Secondly, the physical system models in WIMAC are limited. Indirect interaction between devices are limited to shared physical properties. However, in our use cases, some added concepts and information could be used to identify such indirect interactions between devices. For example, additional information about Context could characterized where devices are located and interacting (e.g. kitchen, living room, etc.). Due to the large number of concepts we may add and to avoid some too specific extensions of the WIMAC model, we investigate some ontologies-based approach and some semantic reasoning with inferring rules.
- Thirdly, we could extend the new AGG rules application engine. New operators would be added for the application of several rules in parallel on the same starting graph followed by the merging of the results, These new operators will allow the transformations thus defined to no longer depend on the order of the rules and would thus reduce their complexity.

6 Test and Simulation for SIS

6.1 Overview & Main Achievements

6.1.1 *Overall approach*

In this section we describe the ENACT Test and Simulation enabler (TAS enabler) for SIS. We explain the motivations and provide a technical presentation based on the current scope of work.

6.1.1.1 Motivation

Software testing is a crucial step of any software development process, especially in the DevOps software development cycle. Having access to a production-like environment that reproduces the same condition where a piece of software would run is usually tricky or close to being an impossible task. This is even more so in IoT environments where developers need to test their applications to ensure trustworthiness requirements are met and well managed.

Access to devices, sensors and actuators within a specific environment might not be trivial or can be limited due to many factors. Networks of physically deployed devices are typically devoted to production software, and testing applications on top of those networks might involve additional testing software, which might affect overall performance and maintenance costs of the devices if, for instance, they need to be stopped to load the new versions of applications.

In such scenarios, software simulators proved to be valuable in easing the validation of the requirements by providing developers with a testing environment to, at least, manage the execution of the tests on their applications. When it comes to IoT application testing, simulation tools provide developers with an initial testing platform to develop their applications before putting them into a production IoT network. In this way, the impact of the application development on IoT systems is minimized, and the time needed to deploy a new feature, or a new update is reduced. IoT testbeds also play a relevant role when it comes to testing applications. Testbeds offer a deployed network of IoT devices where developers can upload their applications and test their software in a real environment. IoT-Lab [36] and SmartSantander [37] are good examples of IoT testbeds. Testbeds often have a predefined fix configuration and architecture. They are also usually shared with other users, which can be a problem when it comes to measuring application performance. Hence, the main drawbacks of the testbed approach can make simulators more attractive, since they can provide a more customised and controlled environment. Furthermore, simulators avoid the need of having a more expensive physical network of devices.

In recent years, both academia and the commercial market offered solutions in the IoT simulation field (for some examples, please refer to D2.1). Although the objectives are similar, their approach is often entirely different. Academic solutions implement cutting-edge technology in the form of proof-of-concept, which usually are not ready for production systems. By contrast, commercial solutions focus on producing a stable and flawless solution, even though the technology behind might not be at the cutting-edge state-of-the-art level.

All of the above point out that there is a need for a complete set of test and simulation solutions for IoT, such that the system can be tested based on the predefined scenarios with use of sensor and actuator data which make sense in the given scenario but also stresses the boundaries of the scenario in order to detect potential problems. In Table 10, the scope of common IoT testing is listed. The ENACT TAS enabler addresses all of the categories apart from the ones bound to the component and communication testing which are purely connected to the hardware testing and as such are considered out of the scope of this enabler.

Table 10. Example test condition of IoT testing scopes

Test Categories	Sample Test Conditions
Components Validation	• Device Hardware • Embedded Software • Cloud infrastructure • Network Connectivity • Third-party software • Sensor Testing • Command Testing • Data format testing • Robustness Testing • Safety testing
Function Validation	• Basic device Testing • Testing between IoT devices • Error Handling • Valid Calculation
Conditioning Validation	• Manual Conditioning

	• Automated Conditioning • Conditioning profiles
Performance Validation	• Data transmit Frequency • Multiple request handling • Synchronization • Interrupt testing • Device performance • Consistency validation
Security and Data Validation	• Validate data packets • Verify data loses or corrupt packets • Data encryption/decryption • Data values • Data exfiltration • Data integrity • Users Roles and Responsibility & their Usage Patterns
Gateway Validation	• Cloud interface testing • Device to cloud protocol testing • Latency testing
Analytics Validation	• Sensor data analytics checking • IoT system operational analytics • System filter analytics • Rules verification
Communication Validation	• Interoperability • M2M or Device to Device • Broadcast testing • Interrupt Testing • Protocol

With the motivation of focusing on the network of sensors and the applications on top of them, we do not take into account the physical behaviour of the sensors and take as granted that the sensors would always react to the physical changes correctly (which means if physical temperature increase by 2 degree, we will receive the message immediately with +2 degree of reading).

6.1.1.2 Overall approach to Test and Simulation

Based on the challenges identified for the testing and simulation of smart IoT systems described in D2.1 section 5.1.1, the aim of the enabler is to provide a simulation tool to test the trustworthiness of the IoT application. The approach we will explain in detail in section 6.2.1.2 addresses main challenges and provides a solid baseline for the extension of the enabler to advance the simulation and test capabilities.

Our approach focuses on simulating sensors and actuators, as well as building a set of testbeds so that the tool can cover a wide range of scenarios. The test and simulation also can be done on a snapshot of the real SIS to be able to evaluate the effect of SIS updates before deploying them on the real SIS.

Table 11 showcases the scope of the testing capabilities ENACT TAS enabler can cover.

Table 11. Scope of the ENACT Test and Simulation Capabilities

IoT elements Testing Types	Application	Network
Functional Testing	True	False

Usability Testing	True	False
Security Testing	True	True
Performance Testing	True	True
Compatibility Testing	True	False
Services Testing	True	True
Operational Testing	True	False

6.1.2 *Main achievements and innovations*

The TAS enabler provides a new set of tools for helping in the development of IoT applications. With the simulation of sensors and actuators, the time and cost for setting up the environment is expected to be substantially reduced. The enabler provides a powerful method to create datasets to cover a large number of testing scenarios which are not easy to produce in the real environment. The enabler has been designed so that it can be easily integrated into any DevOps-based development, which brings automation capabilities for testing IoT application (system) to a new level, saving a lot of resources (*e.g.*, costs and time). The validation in two use cases, Intelligent Train System and eHealth, show that the tool can have an important role in developing and operating an IoT system.
Regarding the status of all the features to be delivered at the end of the project as described in the extra-deliverable: "Plans for development and evaluation for 2020 and until the end of ENACT". Submitted to the Project Officer in January 2020 as an additional document to the planned deliverables. All the promised features are delivered as summarized in Table 12.

Table 12. Features status

Feature	Description	Status at M22	Status at M35
Simulate multiple sensors/actuators: different types and Test the operations and functionalities	See Sections 6.2.2.1,6.2.2.2, 6.2.3	Planned	Completed
Simulate the communication protocols (***e.g.***, MQTT, MQTTS etc.) and Test the communication in the system	See Section 6.2.2.4, 6.2.2.5, 6.2.3	Planned	Completed
Simulate a big number of sensors and Test the scalability of the system	See Section 6.2.2 and 6.2.3	Planned	Completed
Simulate cyber-attacks and Test the security of the system	See Section 6.2.3.5	Planned	Completed
Simulate failures and Test the operations of the system in case of failure	See Section 6.2.3.5	Planned	Completed
Generate simulation model from GeneSIS model	See Section 6.2.2.4	Planned	On going
Collect the system data (e.g, network traffics, logs, etc.) for replaying data	See Section 6.2.3.4	Planned	Completed
Automate the notification of Test and Simulation results	See Section 6.2.3.2, 6.2.3.6 and 6.2.4.2	Planned	Completed
Synchronise the real-time data with simulation model	See Section 6.2.3.4	Planned	Completed

6.1.3 *Improvements over D2.2*

Several improvements have been carried out during the development of the last version of the enabler reported in D2.2. In the following, a summary of these improvements is provided:

Improvements in the architecture and content

- The architecture of TAS enabler has been improved in order to conceive a modular tool
- Restructure the content so that we have a separate section on Simulation and on Testing Enabler

Due to the change of consortium, the progress was delayed at M22, and we have managed to catch up in M35, this is why most of the content for this enabler is new in D2.3.

New features

- New sensor model with multiple measurements
- Complete Regular and Malicious Data Generator flow
- New graphical Interface
- Complete Regular and Malicious Data Generator flow
- Data recorder module
- Validation module
- Test campaign Management
- Packaging the enabler as a docker image

6.2 Test & Simulation enabler (TAS enabler)

6.2.1 *Overview*

In this section, we present a high-level architecture as well as some technical details of the Test & Simulation enabler.

Before going into a detailed description of the TAS enabler architecture, we present the main components of the IoT system used by the enabler for simulation purposes.

6.2.1.1 Basic SIS component model

Figure 83 shows the basic SIS component model that includes all the following components:

- Sensor Node: captures, pre-process and sends sensor's data to the gateway, which can be a Raspberry PI, an Arduino, etc. It implements some basic modules:
  - Sensor: captures the environment's information.
  - On-board processing: reads the sensor's data and pre-processes it (*e.g.* performs calculations, formalises and validates data). This can be an IoT application, a Node-RED flow, etc.
  - Communication: communicates with the gateway to send or broadcast the processed data.
- Gateway device: receives data from the sensor nodes and processes or just forwards it to the other components/services such as cloud solutions, control centre, etc.
- Actuator node: reacts and controls the actuator based on the reactions of the IoT system. It contains some basic modules:
  - Actuator: triggers the change on the IoT device such as: open the door, activate the alarm system.
  - On-board processing: reads the actuator data signal and converts it into the action.
  - Communication: communicates with the gateway to receive the actuated data signal.
- Other components: higher level components that can provide a service or an application that receives the data, processes it and performs reactions based on the specifications.

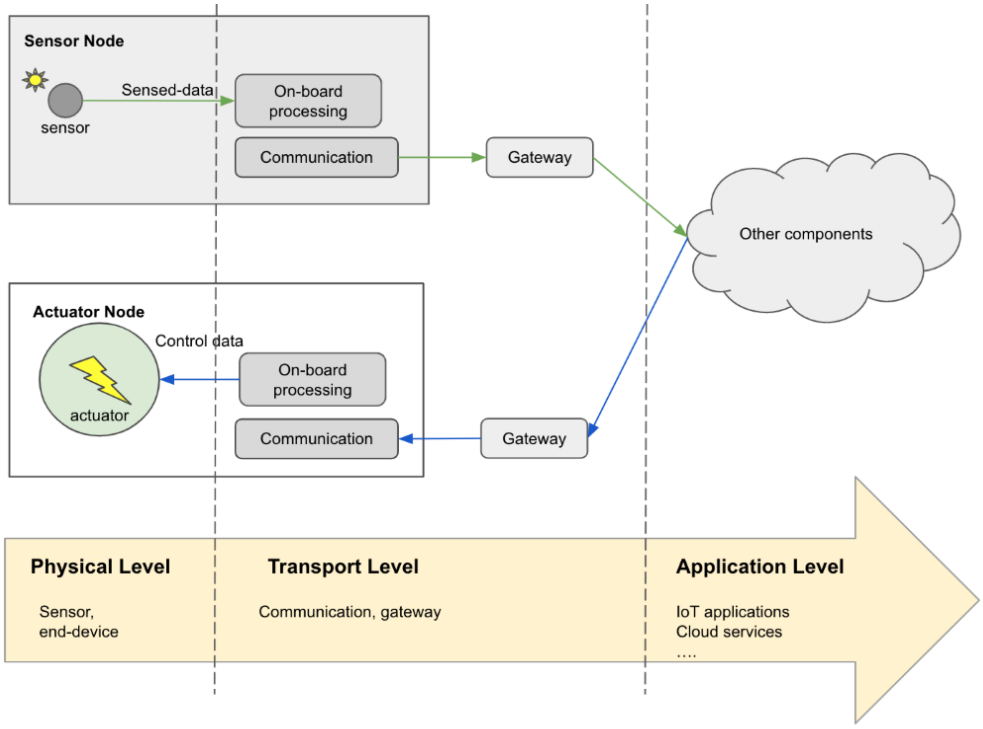


Figure 83. A simple IoT network architecture

6.2.1.2 Approach to simulate a SIS system

A SIS can be presented with 3 levels as depicted in Figure 83. The physical level contains all the physical components such as sensors, devices, which are produced by a manufacturer and cannot be changed by developers. The transport level is responsible for transmitting the data in the SIS network and can be configured by developers to use a specific port number or a specific protocol. Finally, the highest level is the application level, which contains the application written by developers. The application receives the data from sensors, processes it and produces an action achieved by the actuators (*e.g.* turn the light on or off).

A software application needs to be well tested every time there is a change in its source code or on the infrastructure it uses. The planned tests aim to cover many scopes (see table 7) involving different scenarios. In order to cope with multiple scenarios, the testing environment needs to be flexible, making it easy to manipulate the input and measure the output. In regard to the IoT application, having such a testing environment is not easy since the smart things in the physical level (see figure 54) depend on their actual external environment. This means that only the scenarios matching the current condition of the environment can take place and be tested.

In an IoT network, a sensor captures the information of its surrounding environment at a specific time. The information is transformed into a digital format. This information will then be read by a reader or broadcasted by the sensor. The format of the information can be a single value (15 degrees, 81% humidity, etc.), or a composite value in case of a GPS position for example. Since it is not reasonable to wait for the change in the environment to test the IoT application, simulating various data measurements in the sensor is very beneficial when testing IoT applications. It allows the developer to completely control the value provided by the sensors and, thus, simulate and test the IoT application in all scenarios without waiting for environment changes.

An actuator is responsible for controlling a system, for example switching on a light bulb. The status of the actuator is the result of reactions to changes in the environment. For example, when a movement sensor detects a movement, this event will be processed by the IoT application which as a consequence will send a command to the actuator, and the result is that the light

turns on. For testing the reaction of a system based on a specific input data, it is enough by measuring the output which are the commands sent to the actuator. For simulation, we can simulate an actuator by simply creating a hub to receive the actuated command instead of using a real actuator. Note that the impact of an actuator on a sensor is not yet considered.

Using this approach, to simulate a SIS we only need to simulate the components at the physical level. The other (software) components can be cloned from a real system and configured to work in a classical test and simulation environment, avoiding the need for communicating with a production environment, as we can see in Figure 84.

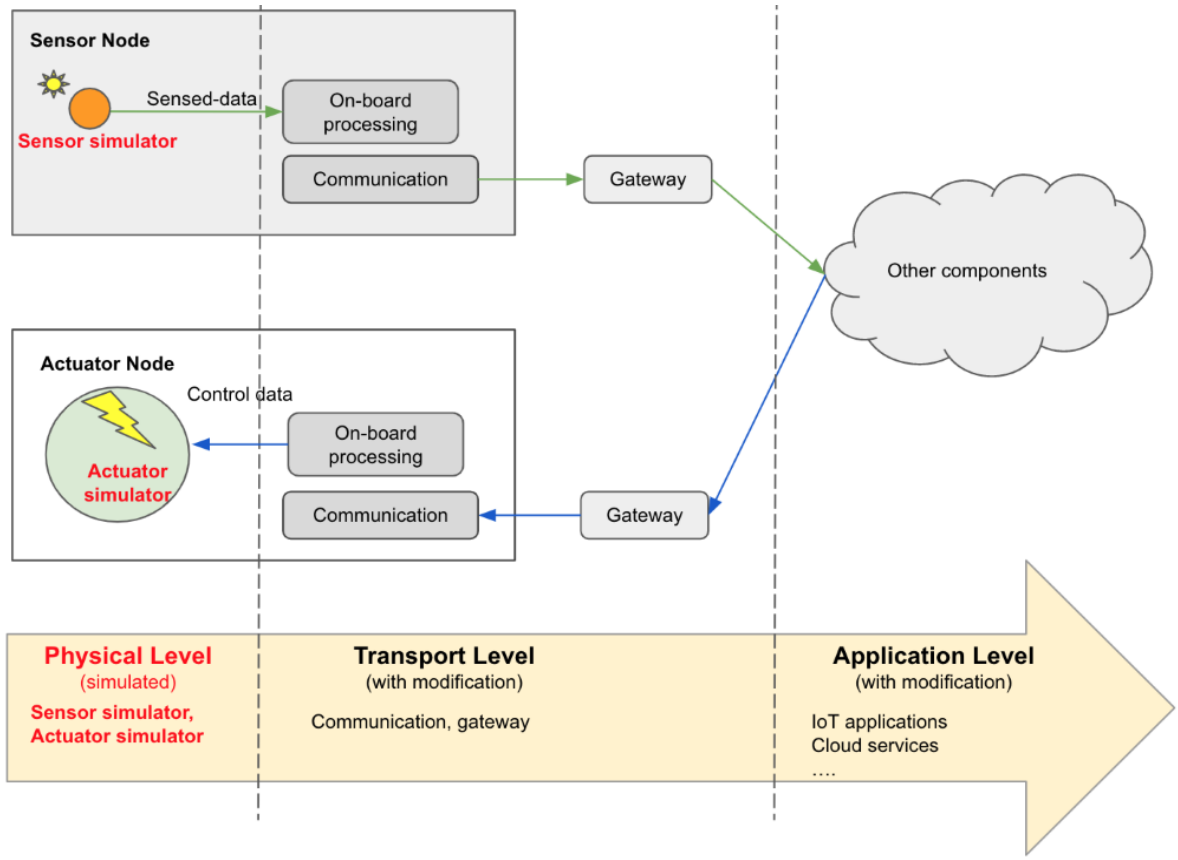


Figure 84. A simple IoT network architecture with simulated components.

Note that with this approach, IoT application developers still can test their application in a realistic IoT network as presented in Figure 85.

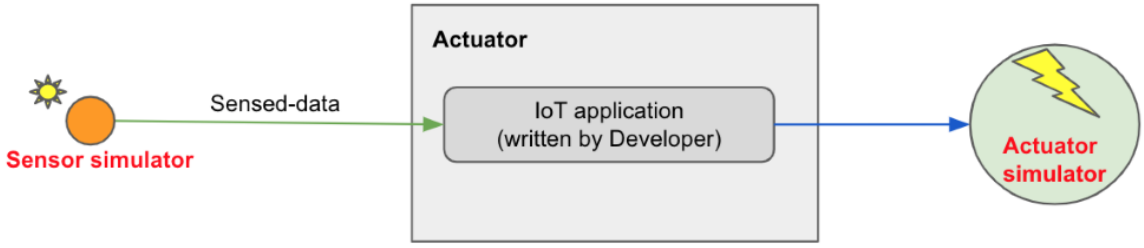


Figure 85. A mini-IoT network

In the next section, we give a more detailed description of the architecture of the TAS enabler for SIS.

6.2.1.3 The Architecture of TAS enabler

In this section, we present the architecture of TAS enabler, which is based on the concept of Digital Twins. Figure 86 illustrates our approach.

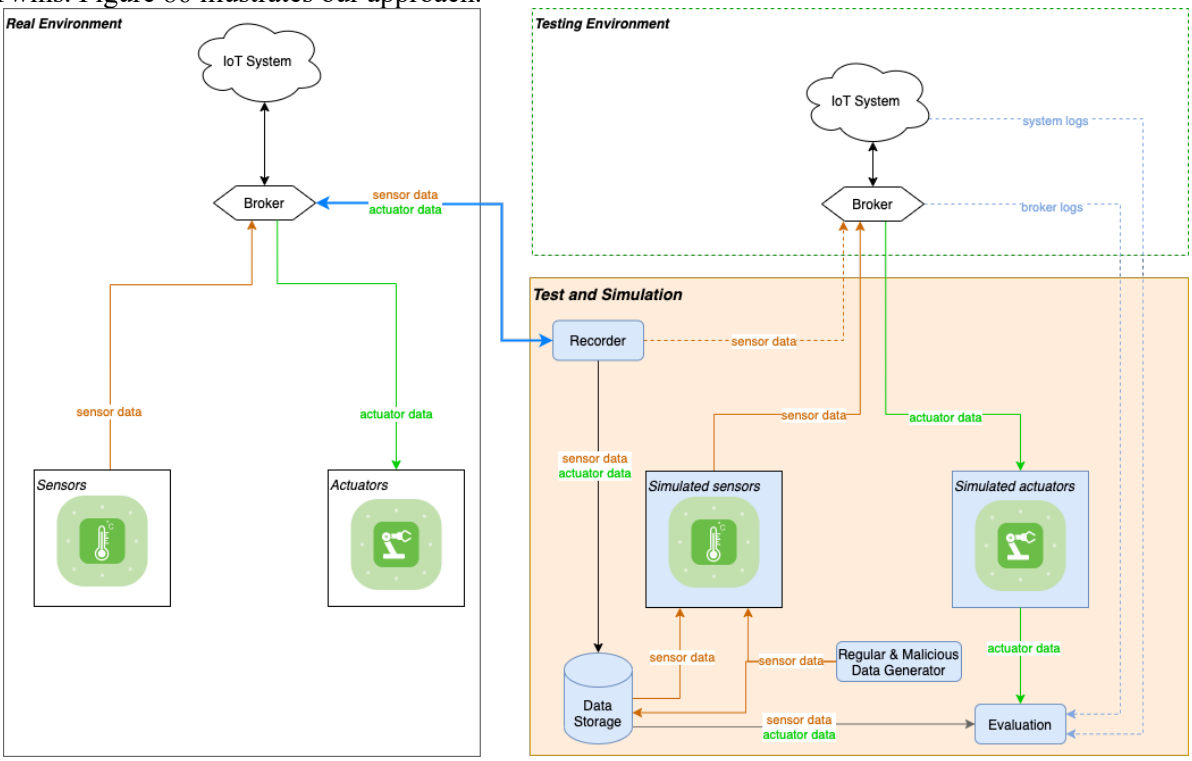


Figure 86. Test and Simulation Enabler Architecture

On the left side is the system in real environment. The communication between the sensors, actuators with the IoT components are done via a broker. Following the concept of Digital Twins [38], on the right side of the figure we have the system under testing environment that has exactly the same topology as the real environment with some differences. *The Recorder module* records all the messages that go throw the broker in the real environment. Each message can be seen as an event happening in the real environment. Then, the recorded messages are forwarded to the broker in the testing environment. In this way we have a "twin' version" of the real environment. What has happened in the real environment is reproduced in the testing environment. In addition, the recorded messages are stored in a *Data Storage* as a dataset for later testing. The simulated sensors have the same role as the real sensors in terms of providing the data signal to the IoT Components. However, they are much more valuable than a real sensor in terms of testing.

- Firstly, by using the dataset which is recorded from the real environment, the simulated sensors can repeatedly simulate the surrounding environment at a specific time. Which means that, in reality, an event may happen only once but, with the simulated sensor, the same event can be generated repeatedly for testing purposes.
- Secondly, the real sensors passively capture the state of the surrounding environment. This means that it can be very difficult to obtain different input for the real sensors. In contrast, the simulated sensors use the dataset in the *Data Storage* as a source of data, so by modifying the event in the *Data Storage*, many different testing scenarios can be generated.

- Moreover, the TAS enabler also provides a very powerful tool to manipulate the input of the sensors. The *Regular and Malicious Data Generator* provides the ability to generate regular data to test the functionalities, operations, performance, and scalability of the IoT system. It also can generate malicious data to test the resiliency of the system to attacks.

On the right side of Figure 86, the simulated actuators provide a destination for receiving the reaction from the IoT Testing Components. By combining the states of the actuators, the logs from the broker and the system, the *Evaluation module* can analyse the results from a simulation.

Figure 87 presents the full architecture of the TAS enabler that can be integrated into any DevOps cycle. The simulation and testing process can be triggered via an API call which occurs when the developer manually triggers it using the *Graphical Interface*, or due to a new code commit, a new configuration, or a new testing scenario. After the evaluation has been made, the *Notifier module* provides the result via a preset webhook, so that the other modules in the DevOps cycle can receive the resulting post-action.

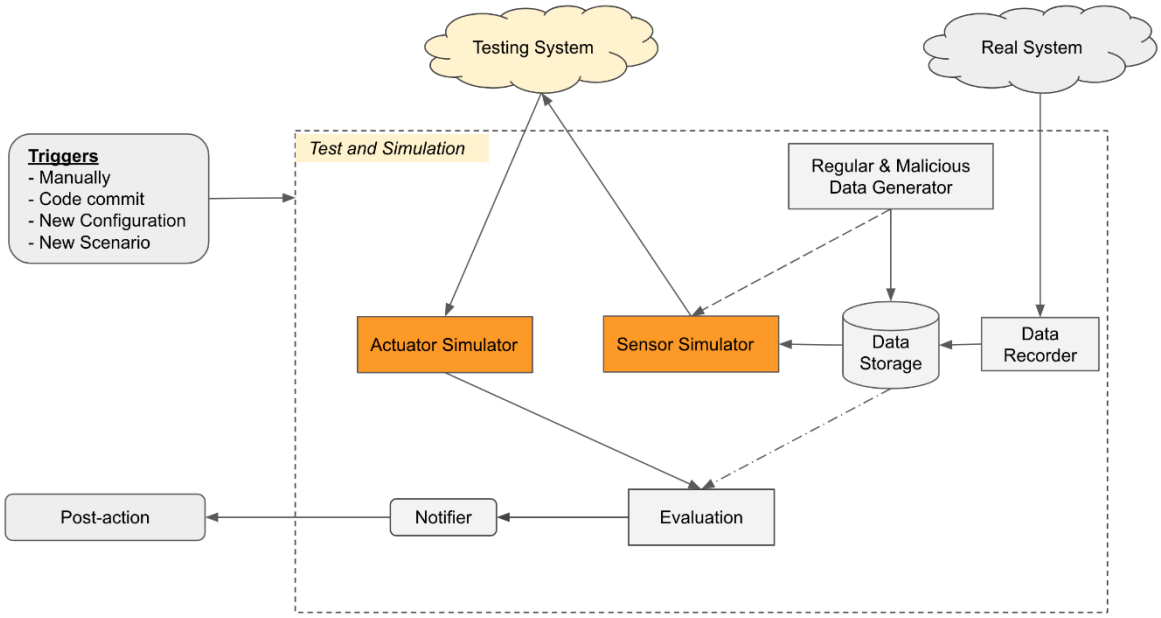


Figure 87. Test and Simulation Enabler Architecture in DevOps loop

6.2.2 *The Simulation*

6.2.2.1 The simulation of sensor

The sensor provides the input data of an IoT system. The simulation of the sensor corresponds to the simulation of the input data stream. The class diagram and the implementation of a basic *Sensor Simulator* model are depicted in Figure 88.

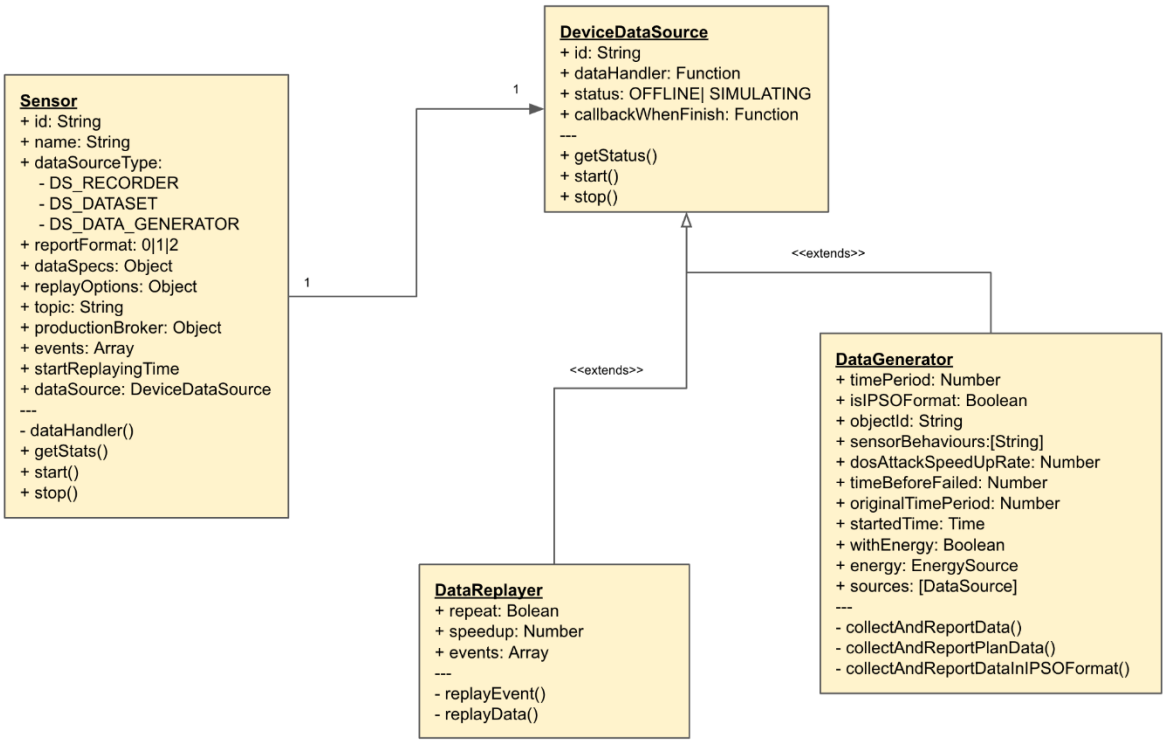


Figure 88. Sensor Simulator's class diagram

The *topic* defines the channel of a message bus on which the data will be published. In the list of all the sensors who publish data to the same broker, each *Sensor* has a unique *topic*, so the topic also can be used to identify the sensor. The *dataSourceType* property indicates where the simulating data source is from. With the *DS_RECORDER* type, the simulating data is the forwarded data from a data recorder which also means that the data in the testing environment are the same as the data in the real environment and they are the real-time data. When the *DS_RECORDER* option is selected, we have to configure the *productionBroker* property which provides the information to establish the connection to the real environment. The *DS_DATASET* indicates that the data sources of the simulation are from the Data Storage, which either can be the recorded data, or data generated by the *Regular and Malicious Data Generator*. When the *DS_DATASET* option is selected, we have to provide a configuration of *replayOptions* property which provides the information about how the simulation process uses the dataset, for instance the possibility of speeding up or repeating the events. And the last option is *DS_DATA_GENERATOR* that specifies the data source the simulation is generating at run-time. The *dataSpecs* property represents the specification of the simulated sensor. It provides the configuration used for generating the data and for the validation process at the end of the simulation; such as: the sensor's behaviours, the measurements as well as some possible cyber-security attack injection during the simulation. The *reportFormat* defines the data format that will be reported; where the value *"0"* means that the sensor will only report the measure value of the sensor without any additional information. For each simulated sensor, the *start* and *stop* methods are used to manipulate the simulation of the sensor.

Figure 89 presents the configuration of a TV status sensor. The data source is from a *dataset* and is replayed with *normal speed*, *repeatedly*. The data messages are published on the topic channel *enact/sensors/cec/status*. The sensor has only one measurement which is *status*. No abnormal behaviours are selected.

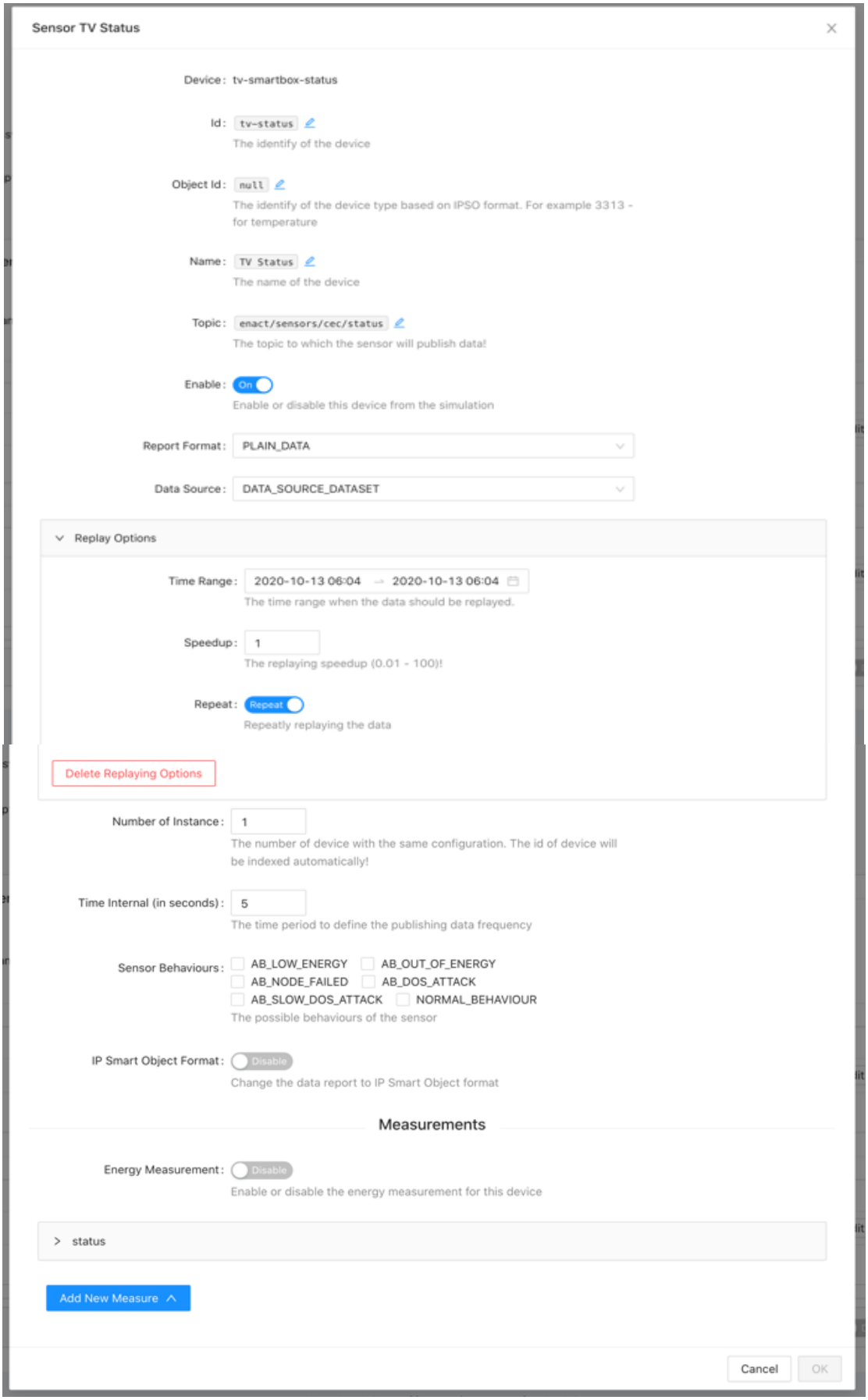


Figure 89. A TV Status sensor

6.2.2.2 The simulation of actuators

The actuator can be considered as the device that receives the reaction of the IoT system based on the input data. In the simulation, we simulate the actuator as a component that will receive the reaction signal (actuated data) from the IoT system. Figure 90 shows the class diagram of the *Actuator*.

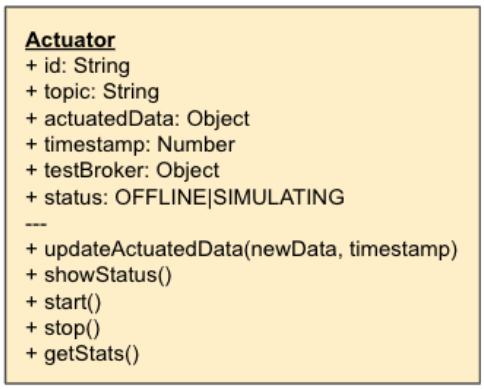


Figure 90. Actuator Simulator's class diagram

Each *Actuator* has a unique *topic*, so the topic also can be used to identify the actuator. The topic defines the channel on which the actuator will connect to obtain the actuated data. While doing the simulation, the simulated sensors send data to the *testBroker*. Then, the IoT application processes the data and provides, via *testBroker,* the reactions that need to be performed by the actuators. The actuators connect and listen for the actuated data coming from the *testBroker.* The methods *start* and *stop* are provided to manipulate the receiving actuated data process.
Figure 91 presents a TV smart-box mute control.

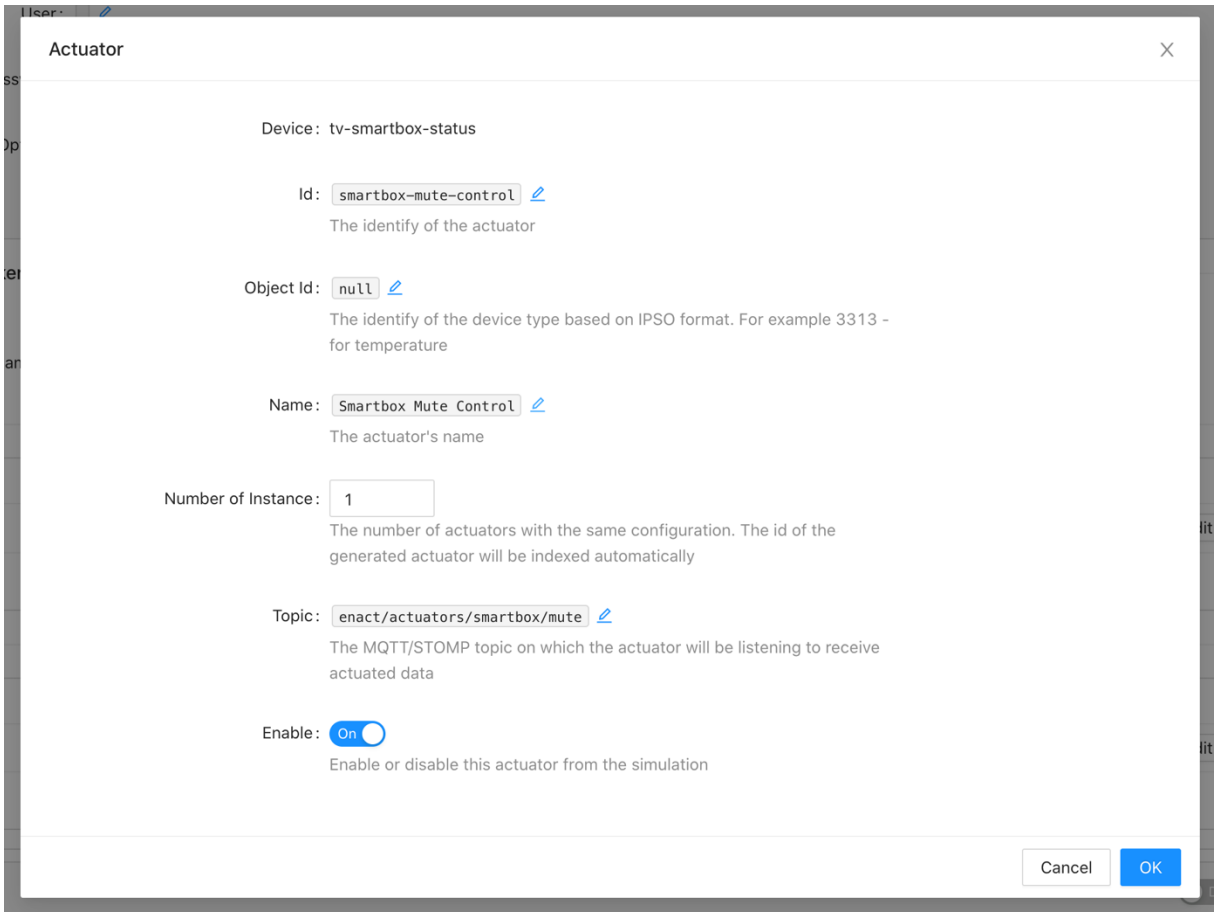


Figure 91. A TV smart-box mute control

6.2.2.3 The simulation of an IoT device

In an IoT system, the sensor and actuator are usually part of the same device. An IoT device can contain one to many sensors as well as one to many actuators. Figure 92 illustrates the class diagram of a *Device*.

Figure 92. Device Simulator's class diagram

Beside the list of *sensors* and *actuators*, a device also contains some information related to the communication between the device and the other components. The *dataStorage* provides the communication with a database which contains all the data required by the TAS enabler. The *testBroker* provides the communication between the tool and the testing environment. The *productionBroker* provides the ability for synchronizing the data between the real system and the testing environment.

The *timeToFailed* defines the time before the device is going to stop simulating if *GATEWAY_DOWN* behaviour is set in *behaviours*.

When the simulation uses the data sources from the database, there are a few options that need to be set. The *datasetId* indicates which dataset is to be used. The *globalReplayingOptions* defines the global configuration for the replaying, such as, the start time, end time (the duration for replaying the events), speedup (the speed for replaying the events), and if the replaying should be repeated. The *isReplayingStreams* defines if the simulating will be based on the sensor definition or streaming data. In

case of streaming, the *upStreams* represents all the data streams sent by sensors and the *downStreams* represents all the data streams received by actuators.

The *newDatasetConfig* defines the new dataset where the generated events will be stored during the simulation.

Finally, we have some methods for manipulating the devices, such as, *addSensor*, *removeSensor*, *addActuator*, *removeActuator*, *start*, *stop*.

In an IoT system, we usually have a lot of IoT devices. The next section will present the simulated network topology.

Figure 93 presents an IoT device which contains a TV sensor and a Smart-box Mute Control actuator.

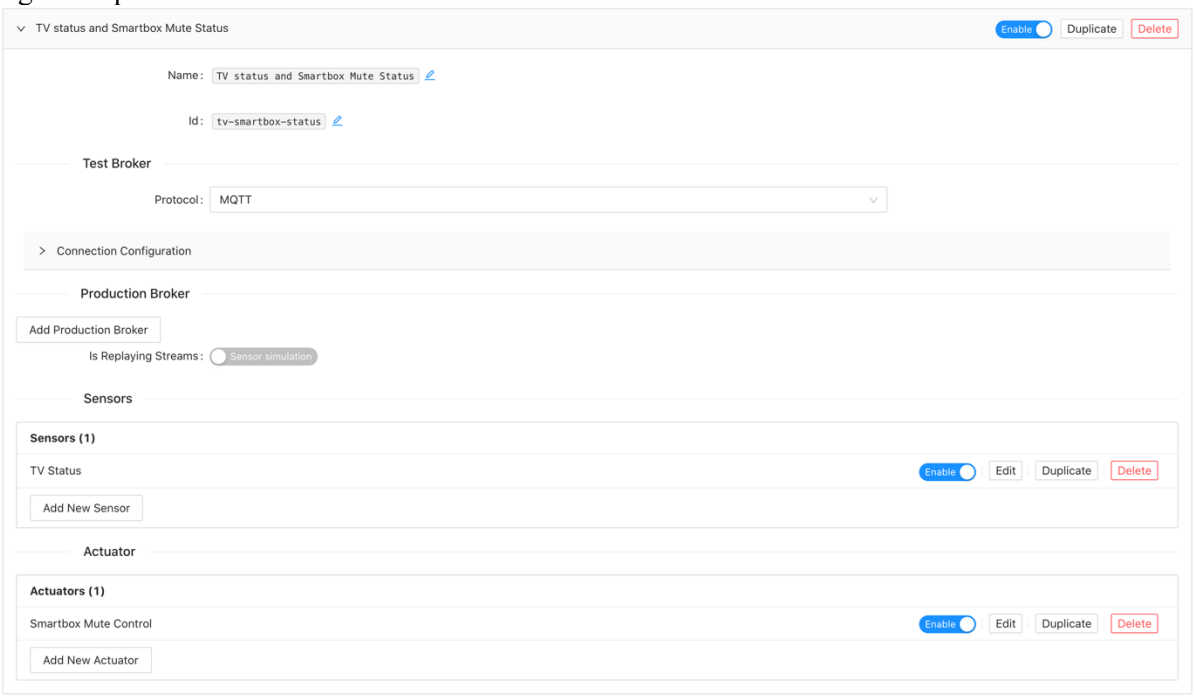


Figure 93. An IoT device

6.2.2.4 The simulated network topology

The network topology is the minimum input needed to perform a simulation. Figure 94 presents the class diagram of the Simulation.

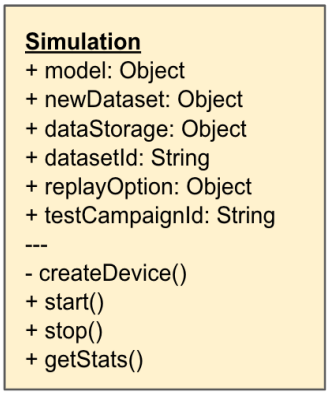


Figure 94. Simulation class diagram

The simulated network topology presents the list of simulated IoT devices. Beside the list of devices, a network topology can also provide the *datasetId* for replaying data, the global replaying options, the

configuration to connect with the database, and the definition of the new dataset where the generated data from the simulations will be stored.

Figure 95 presents a network topology which contains two devices. The first device contains one sensor TV status and one actuator Smartbox Mute status device. The second device contains only one sensor Call status. The data source is from the dataset with the Id *smarthome-dataset-01*. The replaying option defines the duration time that the data will be replayed (in the example between 1.am and 15h34 on 23rd September 2020), the speedup mode (3 times faster than the original speed), and that it will be replayed only once. The data generated during the simulation will be stored in a new dataset with the id *smarthome-dataset-replayed-02* and the *Data Storage* will use the default database connection.

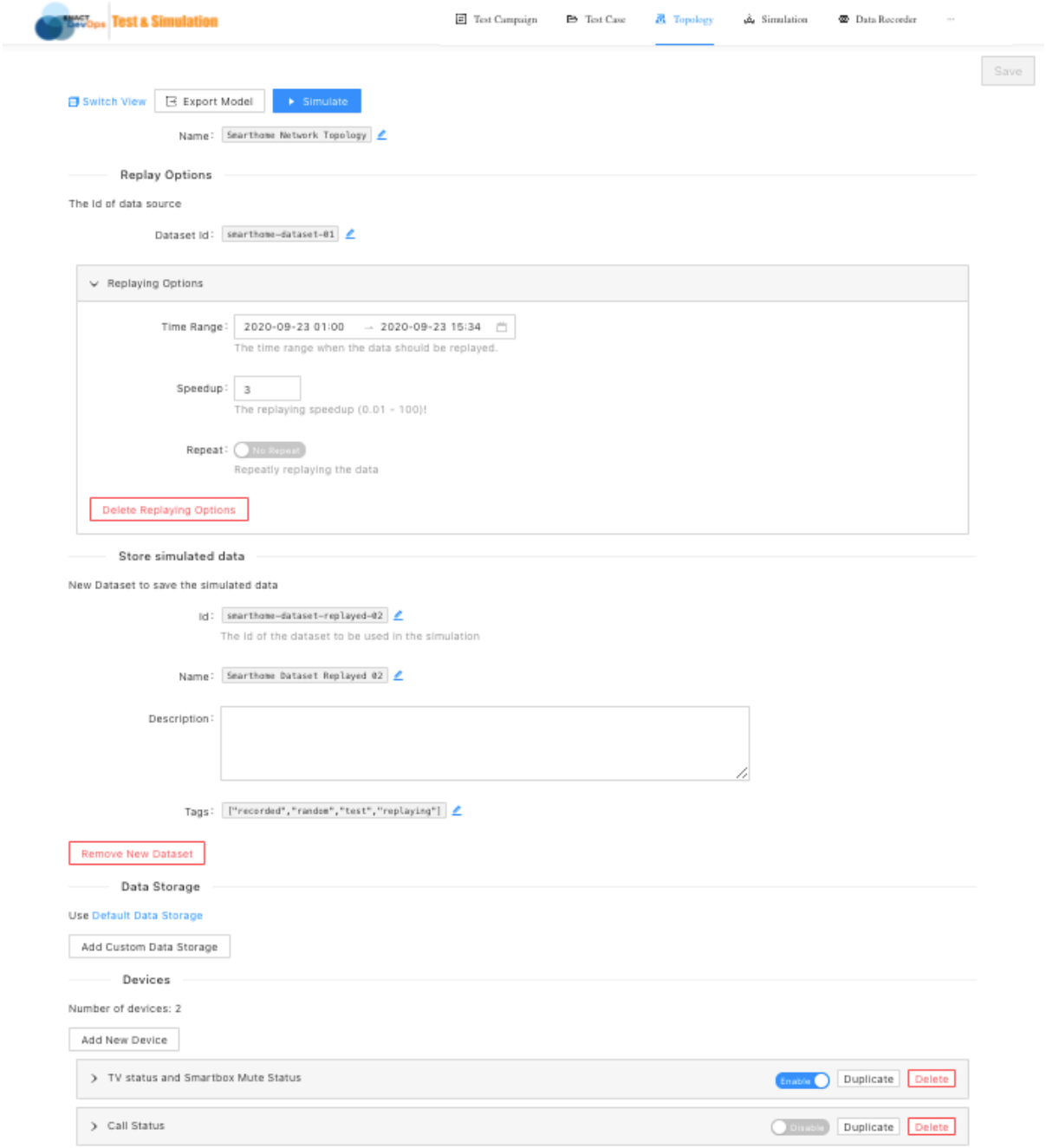


Figure 95. A network topology

Figure 96 presents the configuration of the same network topology in JSON format.

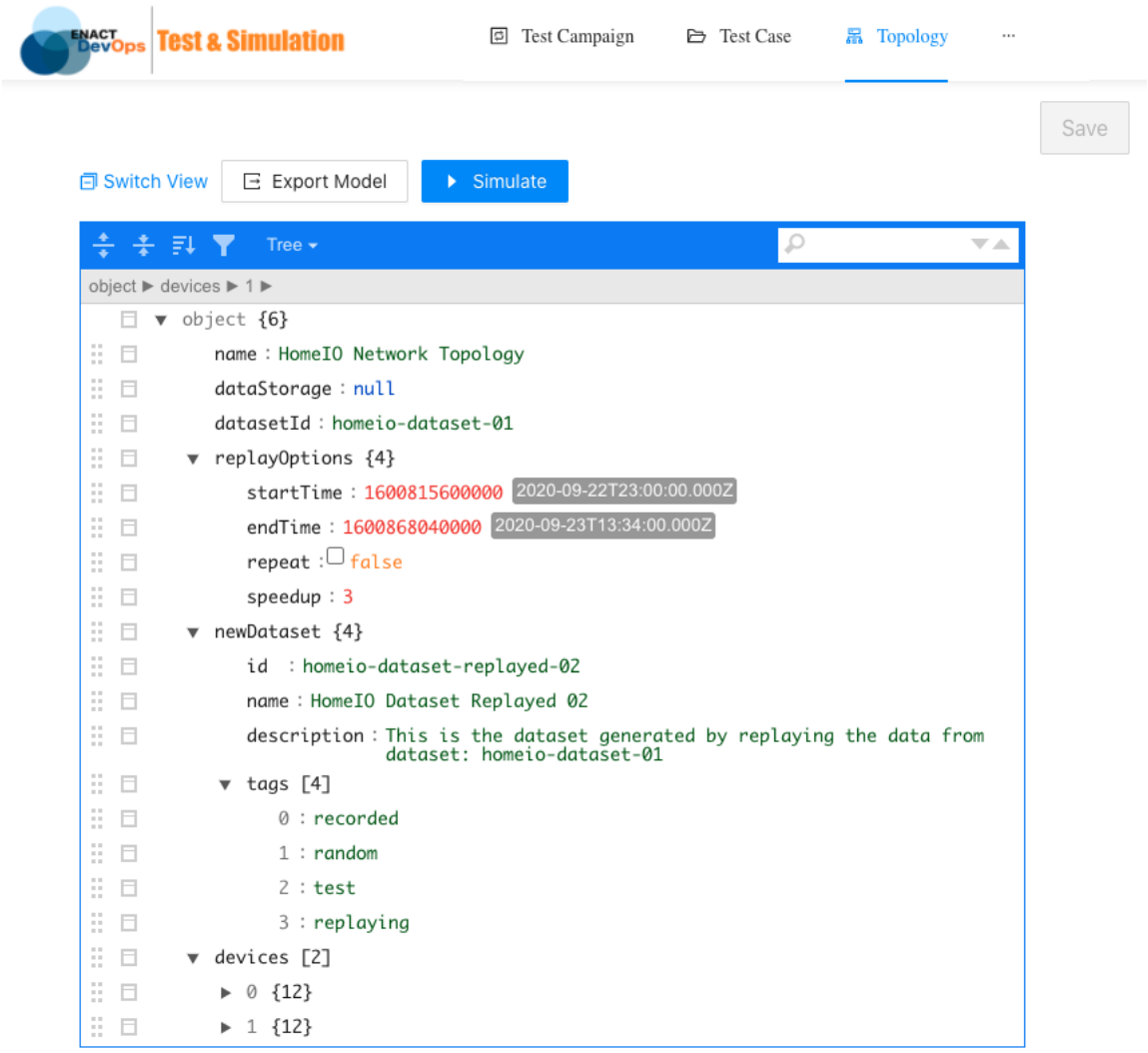


Figure 96. A network topology in JSON format

6.2.2.5 The communication

There are two main types of communication between simulated components and other components. The first type is the connection to a database where all the data used by the TAS enabler will be stored. It depends on the type of the database server, and the connection protocol can be, for instance, MongoDB or CouchDB. In the scope of the ENACT use cases, the MongoDB connection protocol has been implemented, but other protocols can be added to extend the communication part of the enabler. The second communication is the connection between the simulated devices with the testing IoT system, and the *Data Recorder* with the real IoT system. In the ENACT use cases, some message queue protocols have been implemented. Figure 97 presents the class diagram of the Message Bus which is similar to the interface for all Message Queue Protocol.

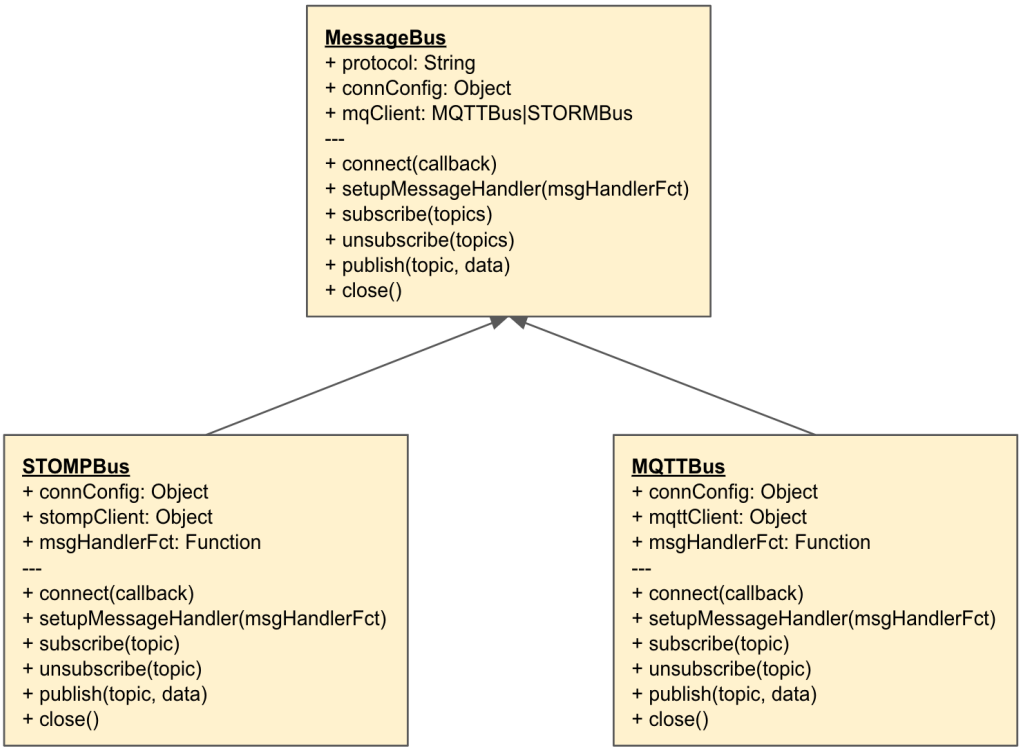


Figure 97. Message Queue Bus class diagram

Some basic message queue bus protocol methods have been implemented such as: *subscribe*, *unsubscribe*, *publish*, *connect*, and *close*. The *connConfig* contains the necessary information for establishing the connection, such as, *host, port* and *tls configuration*.

By design, each IoT device can have its own way of communicating with the database and the testing and real system, so the addition of any communication protocols is possible and can be mixed.

6.2.3 *The Test*

6.2.3.1 The Testing methodology

This section covers the testing methodologies that can be implemented in the TAS enabler (TAS enabler). In this version of the enabler, only data-driven and data-mutation testing methodologies are implemented. The other methodologies described above are possible future extensions.

Data Driven Testing

Figure 98 presents the architecture of *Data Driven Testing* method. The *Data Storage* contains the list of datasets which are recorded from the real system or entered manually. Each dataset contains a set of sensor input and the expected actuator output. The set of expected actuator output can be the value recorded from the real system in a normal scenario. They can also be entered manually via the *Graphical Interface*. The *Evaluation* module will use the expected outputs to compare it with the output of the simulation to determine if they match. A test is passed if the output of the simulation matches the expected output. The *Data Driven Testing* method can be used for functional and regression testing.

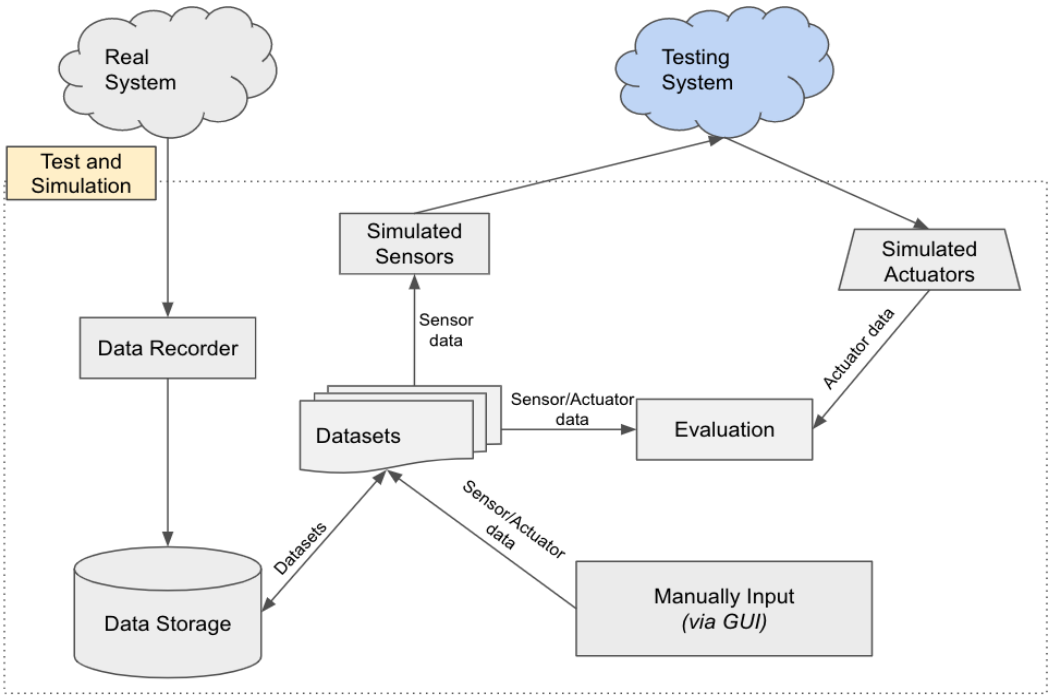


Figure 98. Data Driven Testing

In Test and Simulation enabler (TAS enabler), the *Data Driven Testing* has been implemented as the main testing methodology.

Data Mutation Testing

Figure 99 illustrates the *Data Mutation Testing* architecture. The *Mutant Generator* generates new input from existing input stored in the *Data Storage*. The new input is generated by applying one or many mutated functions, such as, change the event order, change an input value, delete an event. The mutated input will be used for the simulation. The *Evaluation* module will generate a report indicating the changes found in the output when testing with the mutated and the original input data. The *Data Mutation Testing* method can be used for penetration, robustness, security, or scalability testing (*e.g.* mutate the device identifier to obtain new devices). In the TAS tool, the identity of the devices can be mutated to generate many devices so that the scalability of the system can be tested. There is also an interface to manually apply some mutation functions on a dataset, such as, change the event order, change an event value, and delete an event.

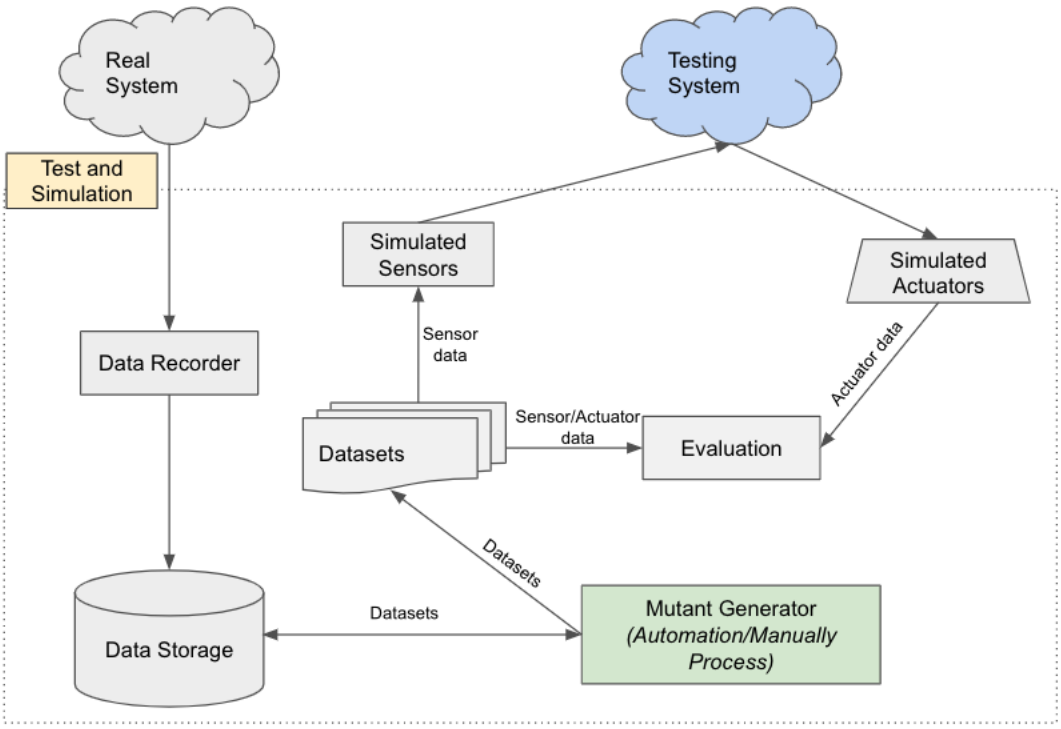


Figure 99. Data Mutation Testing

Model based Testing

Figure 100 presents the *Model Based Testing* architecture. This method uses models to calculate the expected output from a specific set of input. The models are also used to generate test cases. The models can be a state machine, a state chart, or a simple index table. To create the models, there are two methods. The first method consists in manually creating the model and the testing input via the *Graphical Interface*. This method requires that the persons using the tool have a certain level of expertise. The second method is generating the model from traces using reverse engineering and using the *Model Generator* module. All the recorded events from the real system are used as an input for the *Model Generator*. Then, by using some specific methods such as Mealy Machine [30], the *Model Generator* can generate models as well as the testing input to cover all the generated models. For evaluating the test, the *Evaluation* module will compare the output of the simulation with the expected output which is calculated from the models based on the testing input. *The Model Based Testing* can be used for functional and regression testing. In the scope of the ENACT project, this method has not yet been implemented.

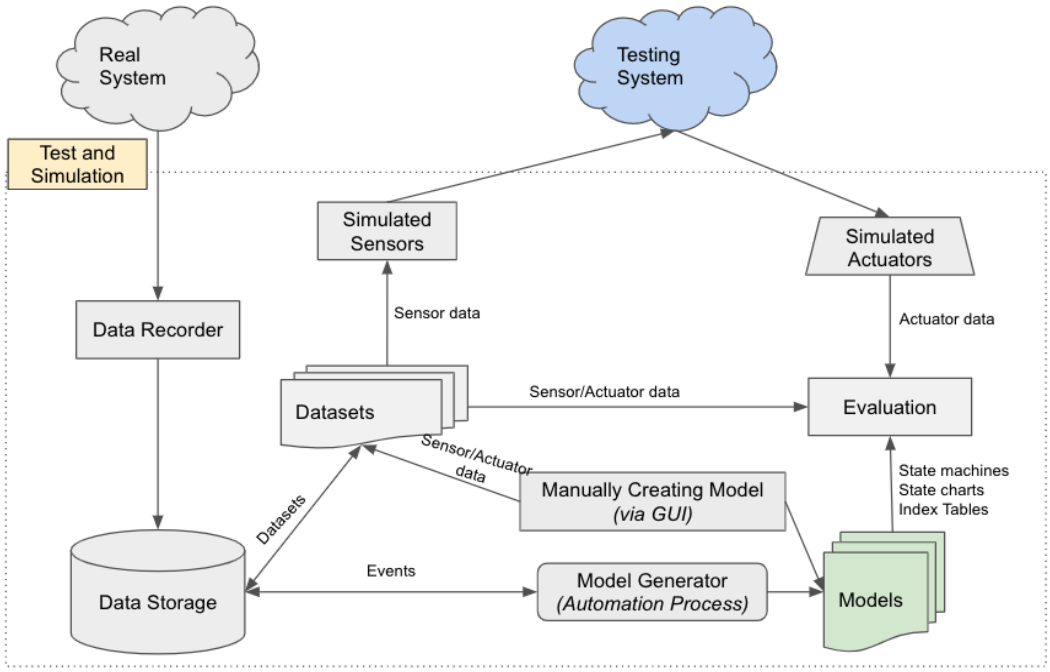


Figure 100. Model Based Testing

Risk Based Testing

Figure 101 presents the *Risk Based Testing* architecture. The risk assessment identifies the potential risks, analyses them, and generates the risk input when possible. The *Attack* tool contains a set of attacks that can be triggered from the *Risk Management*. The simulation uses the input data generated (or specified) by *Risk Management*. Finally, the *Evaluation* module will generate a report that analyses the simulation output and provides system metrics. In the TAS enabler, some security attacks have been implemented, such as, DOS attack and Invalid Input. By preparing the dataset which contains the potential risk, the test can be done automatically whenever the *Risk Management* triggers the process.

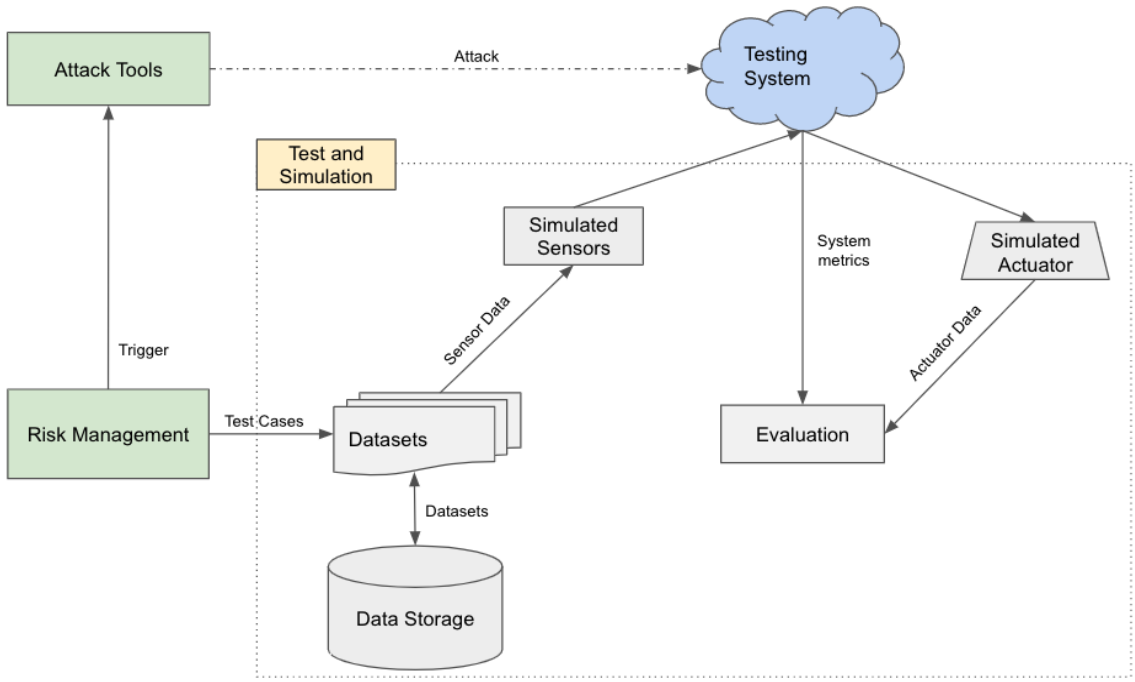


Figure 101. Risk Based Testing

6.2.3.2 The Testbed

To simulate and test an IoT system in some specific scenarios, one of the easiest methods is to use a testbed. With testbeds, the developer can define exactly what is the input and what should be the corresponding output. In this way, the tests can be done automatically and easily integrated in the Continuous Integration and Continuous Deployment processes.

DataStorage

The *Data Storage* contains all the datasets for testing and simulating.

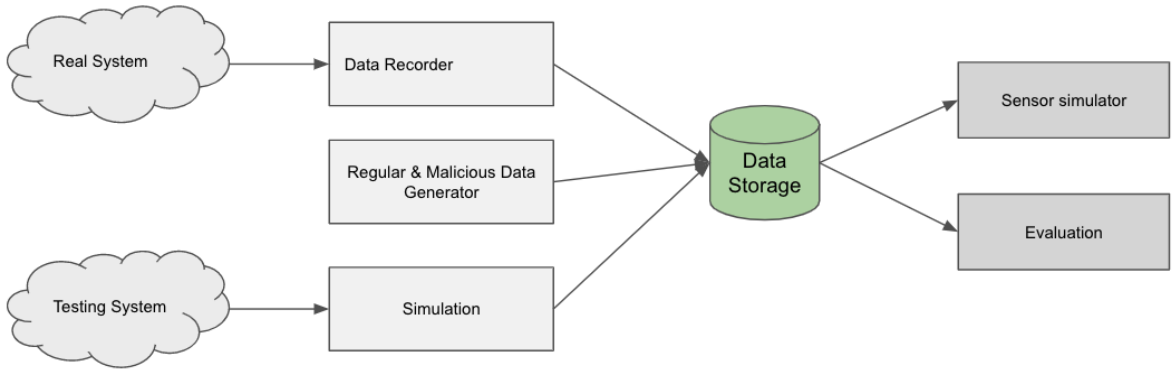


Figure 102. DataStorage

As depicted in Figure 102, the datasets are fed into the *DataStorage* via three sources: the data from the real system recorded by the *Data Recorder*, the data generated by *Regular and Malicious Data Generator,* and the data generated by the simulation. The datasets in the *Data Storage* are used to simulate the sensors and to validate the output of the simulations. Figure 103 presents the class diagram of the *DataStorage*.

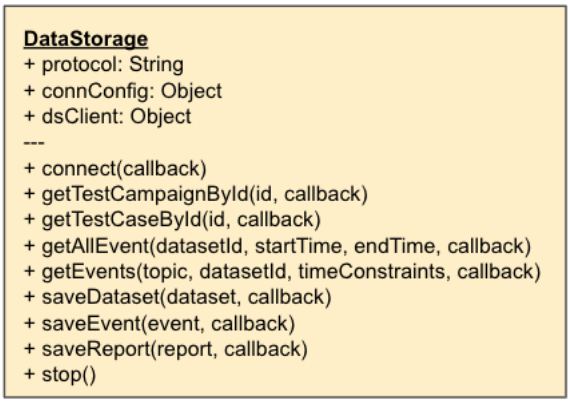


Figure 103. DataStorage class diagram

The *protocol* attribute defines the type of the database connection. In the scope of the ENACT project, we have implemented the MongoDB protocol. The *connConfig* attribute contains the information for establishing the connections. The *dsClient* attribute specifies how to connect to the database. A number of methods have been implemented to work with the database.

For the TAS enabler, the connection to the database is flexible. Two simulations can use different databases. If there is no configuration specified, a default database will be used. Figure 104 presents the interface for setting up the default *Data Storage* configuration. The database to connect to can be any database that can be reached by the TAS enabler.

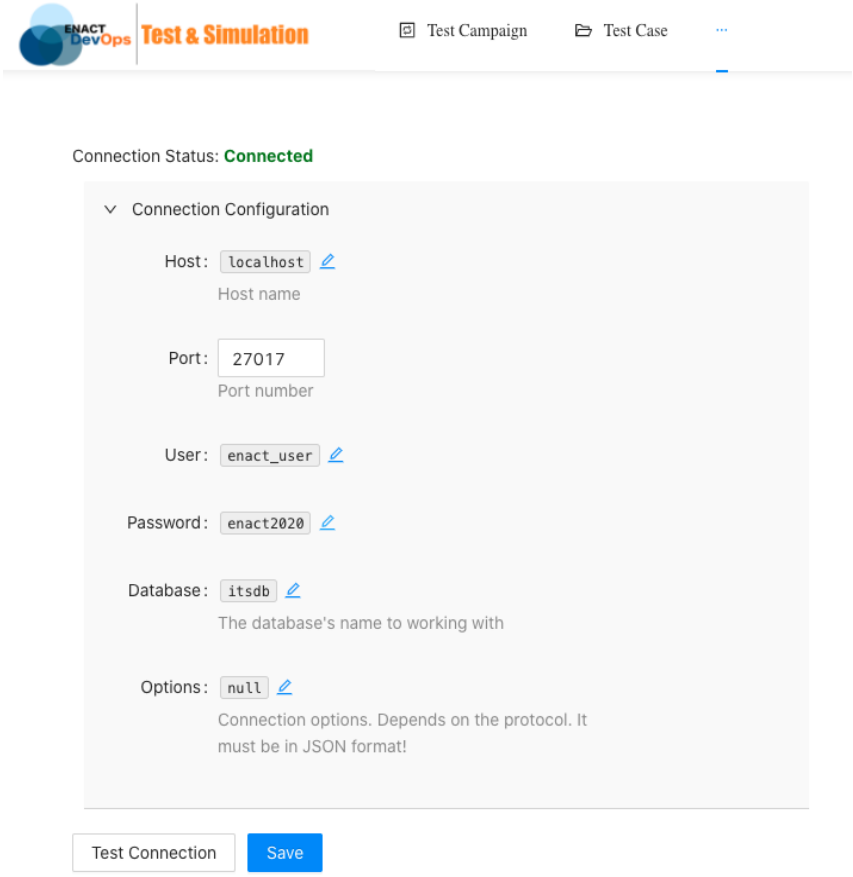


Figure 104. Set the default database configuration

Event

An event represents a message sent through the communication channels. It can be a data message sent by a sensor, or an actuated data received by an actuator. Figure 105 presents the format of the event Schema.

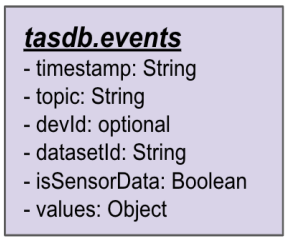


Figure 105. Event Schema

The *timestamp* attribute indicates the time when the event has been captured. The *topic* or the *devId* defines the source of the event in case the data message corresponds to sensor data or the destination of the event in case the data message is an actuated data received by actuator. This value is very important for identifying which event will be replayed. The *datasetId* attribute defines the dataset that the event belongs to. To separate the event from a sensor or for an actuator, the *isSensorData* value is set to *True* for the message data from a sensor and *False* for the message data to an actuator. The *values* attribute contains the value of the message data. This value can be a number, a string, or an object. This design helps make the event generic and can consider any type of message data.

Dataset

A dataset contains a series of events for a specific scenario. Figure 106 presents the schema of a dataset. The *tags* attribute defines the category, type, grouping the datasets into the same group, etc. Each dataset has a unique *id*, *name,* and *description* to describe the objective of the dataset. The *source* attribute indicates the source of the dataset. A dataset can be created from a recording session by the *Data Recorder* (*source*: RECORDED), or can be generated by the *Regular and Malicious Data Generator* (*source*: GENERATED). A dataset can also be created by cloning and modifying data from another dataset (*source*: MUTATED).

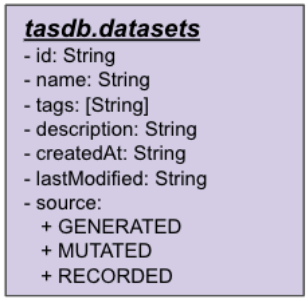


Figure 106. Dataset Schema

By grouping the events by the dataset *Id*, we have all the events belonging to a dataset. Figure 107 presents the details of a dataset. This is a dataset generated from the *Regular and Malicious Data Generator*. In this example there are 30 events in the dataset, and the events have been generated from 07h53:35 to 07h54:45 on November 19th 2020. In the list of events, there is the possibility to mutate the dataset using, for instance, the methods adding a new event, deleting an event, changing the event order, or modifying the value of an event.

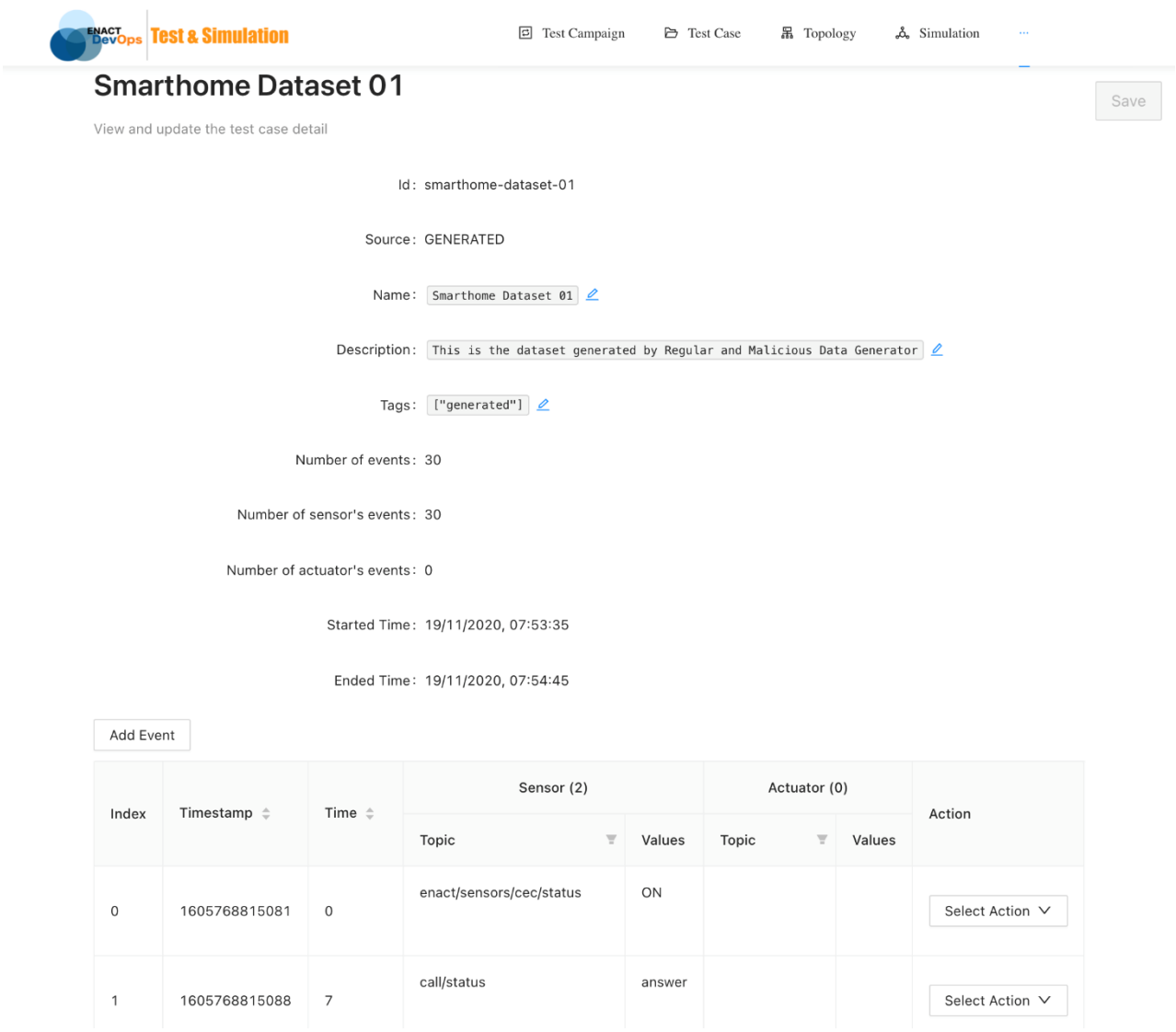


Figure 107. Dataset dashboard

6.2.3.3 Automatic testing

The TAS enabler has been designed to be one of the widgets in the Continuous Integration and Continuous Delivery processes. Figure 108 illustrates the TAS enabler concept. One of three events will trigger the test and simulation process: code commit, new component added, new scenario added. A number of tests will be executed by simulating the different testing scenarios on the system under test. The tests can cover functional, operational, security, performance, and scalability testing. If all the tests have been passed, the new changes can then be deployed on the real environment.

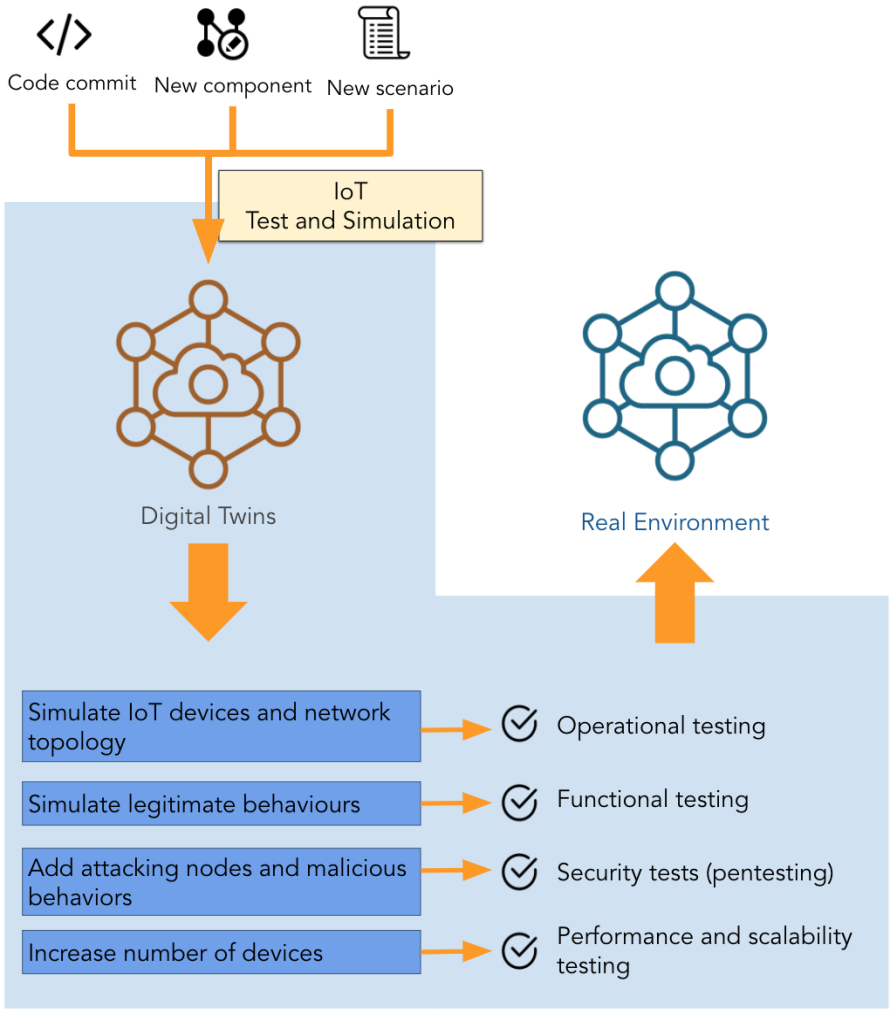


Figure 108. Test and Simulation Enabler concept

Following that process, every change in the system can be tested automatically, and every testing scenario can be covered.

Test Case

While testing an IoT system, we may want to test different network topologies, such as, adding a new device, removing a device, or just changing the way to connect devices. A test case is a collection of tests executed on one network topology. For one network topology, there can be many different types of tests (*e.g.* functional, security or scalability testing). The list of tests should be covered and tested with the list of datasets in a test case. When a test case is executed, the TAS enabler will execute the simulation and test using each dataset, and follows the ordering in the list. The order of the datasets can be changed via the web interface. By grouping the tests by network topology, it is easier to find the optimized configuration of the IoT system.

Figure 109 presents the details of a test case. The *name* and the *description* describe the characteristics of the tested network topology and the objective of the test case. The *modelFileName* indicates the network topology to be tested. The *tags* help to quickly search and organize the test case by its characteristics. And finally, the *datasets* list all the datasets that will be used for testing in this test case.

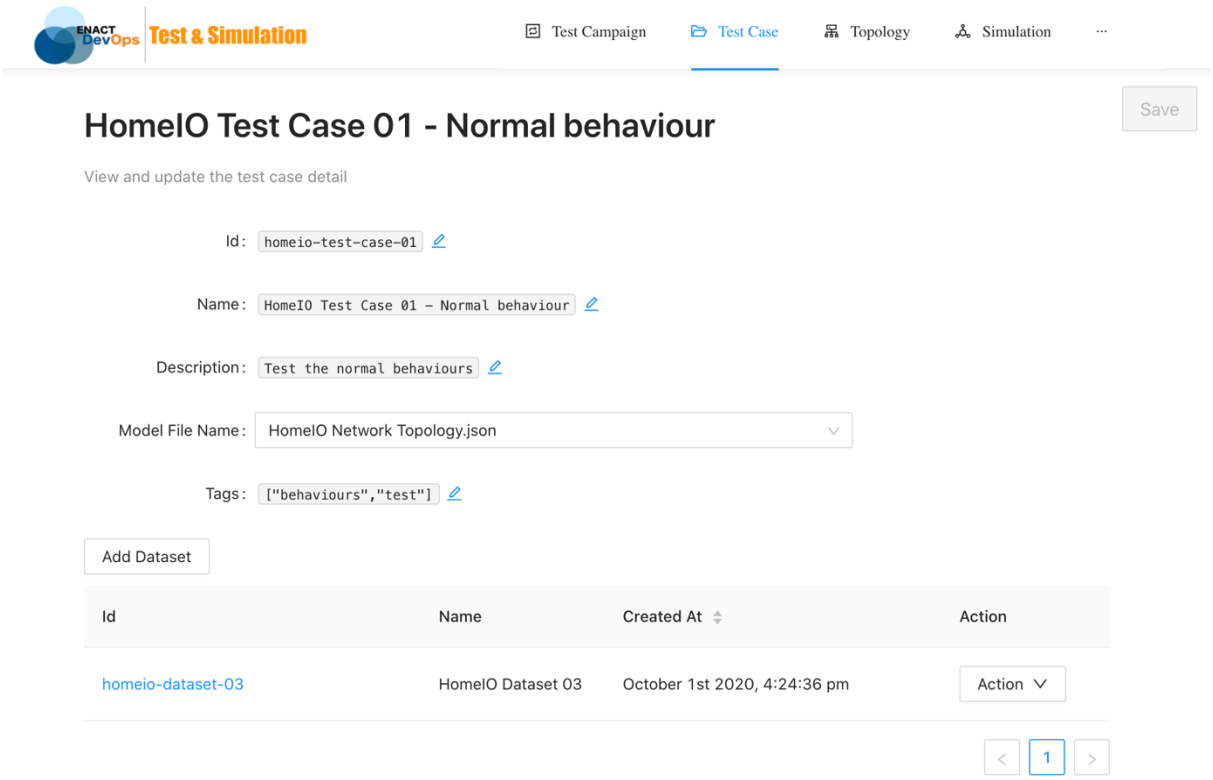


Figure 109. Test case

Test Campaign and the integration into DevOps cycle

While the test case groups the test by the defined network topologies, the test campaign contains all the test cases that should be tested for each modification of the IoT system. This is the global test that covers every testing scenario and testing aspect. The test campaigns are designed to be executed automatically every time there is a change in the system. When the test campaign is executed, the TAS enabler executes every test case in the list, respecting the ordering of the list. The order can be changed via a web interface.

Figure 110 presents the details of a test campaign. The *name* and the *description* describe the objective of the testing campaign.

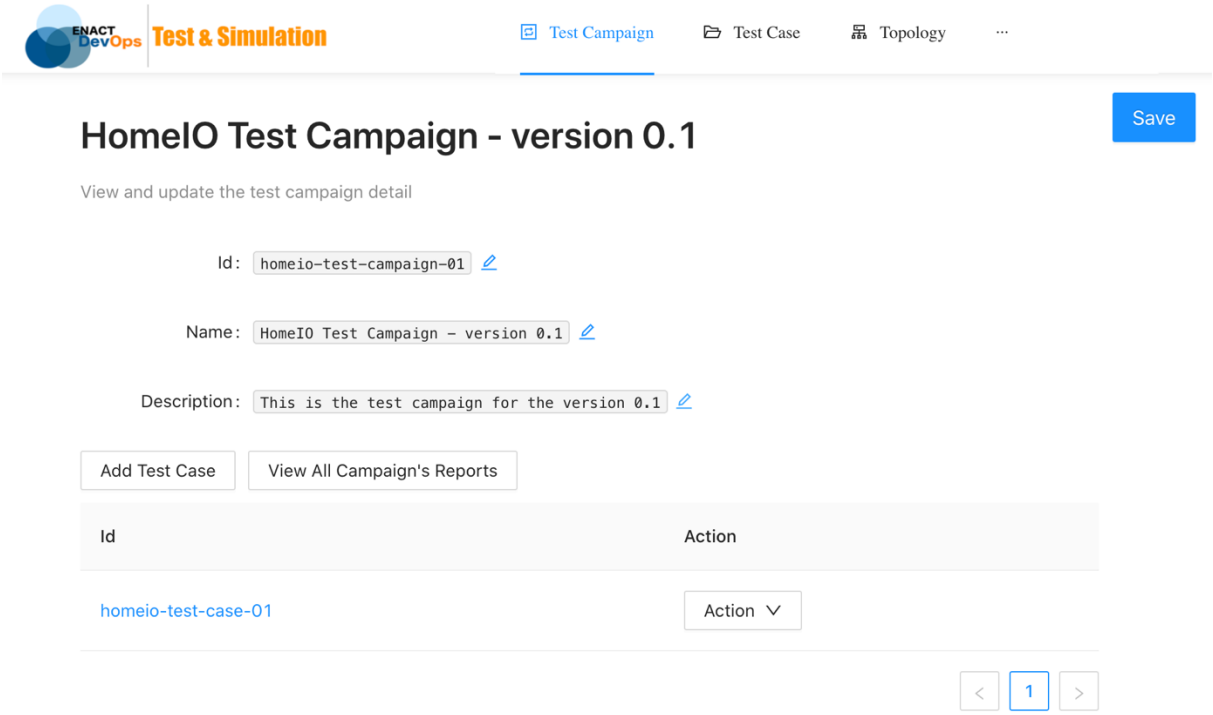


Figure 110. Test campaign

Figure 111 presents the configuration for the automated testing process. When the testing process is triggered, the test campaign *homeio-test-campaign-01* is executed. When all the tests have been executed and evaluated, the result will be sent back to the webhook at the address: *https://dev.enact-server.com/tas-webhook*. Based on the testing result, the new change can be deployed to the production environment, or else needs to be fixed to pass all the tests. This is what we need to be able to integrate the test and simulation process into a DevOps cycle.

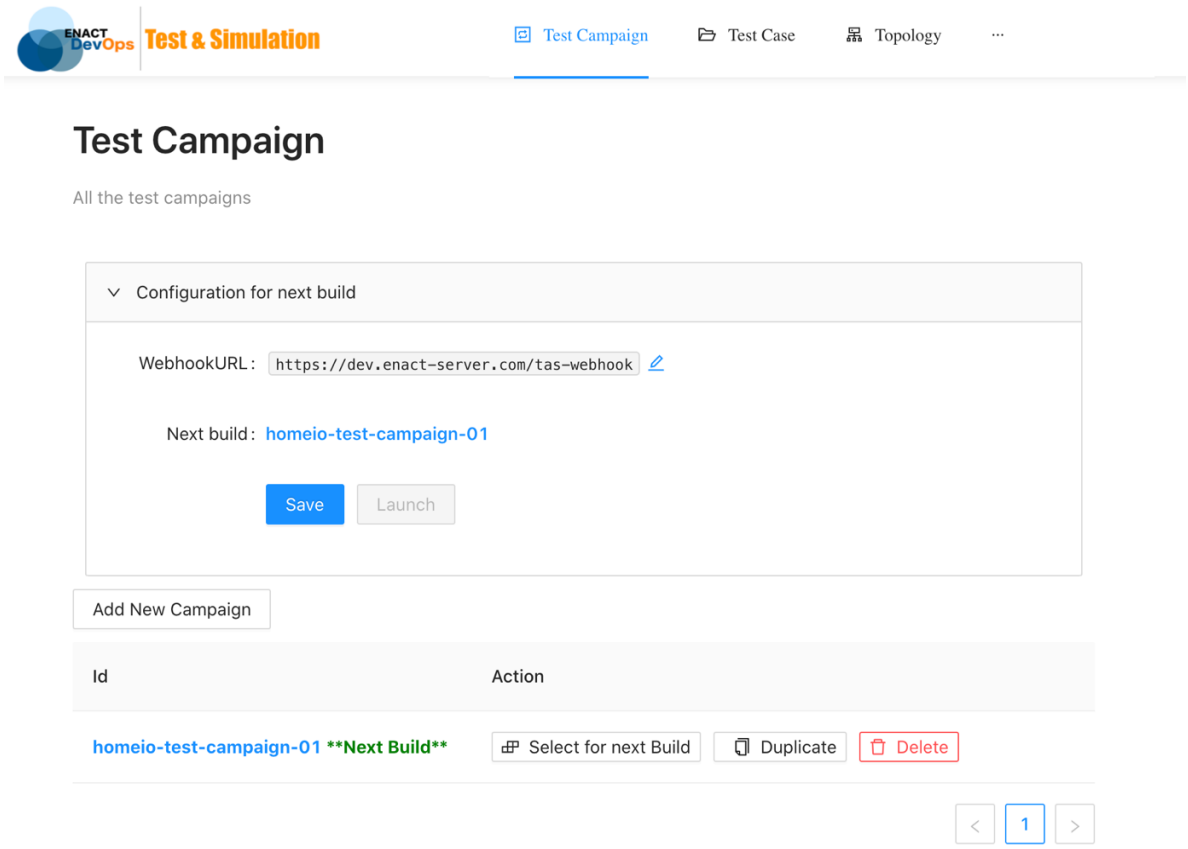


Figure 111. Automation Testing configuration

6.2.3.4 The Data Recorder and Digital Twins concept

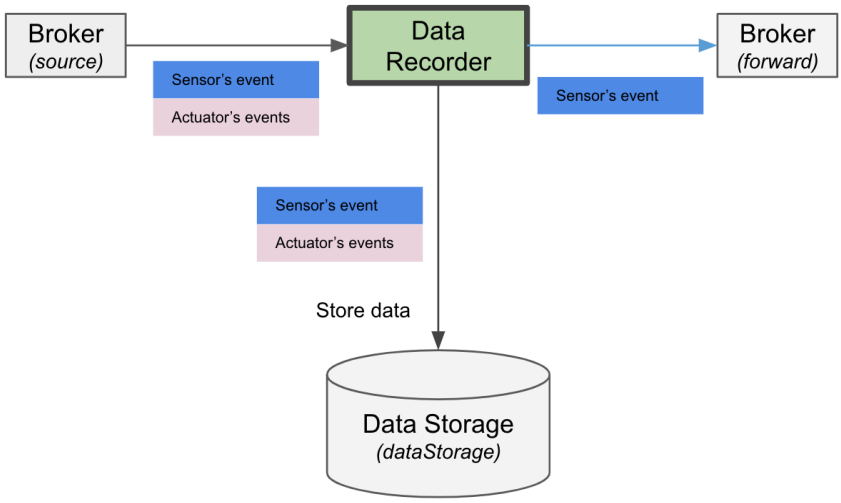


Figure 112. Data Recorder data flow

The TAS enabler provides the possibility to simulate a SIS using historical data. In order to do this, a *Data Recorder* module is needed.

Figure 112 presents the data flow of the *Data Recorder*. All the events in the real system (coming from the broker) will be recorded. This data (including both sensor and actuator data) will be stored into the *Data Storage* as a dataset. The sensor data can be forwarded directly to

the testing system (using the forwarding broker). With the recorded data from the real system, the SIS can be tested with real input. The more data is recorded from sensors, the more scenarios can be tested. By synchronizing the timestamps of the Sensor simulator with the *Data Recorder*, it will be able to simulate a particular SIS (following the *Digital Twin* concept). By monitoring both input and output of the SIS, we can build an automatic testing process for a complex IoT system.

Figure 113 presents the *Data Recorder*'s dashboard. It is possible to execute multiple *Data Recorders* at the same time.

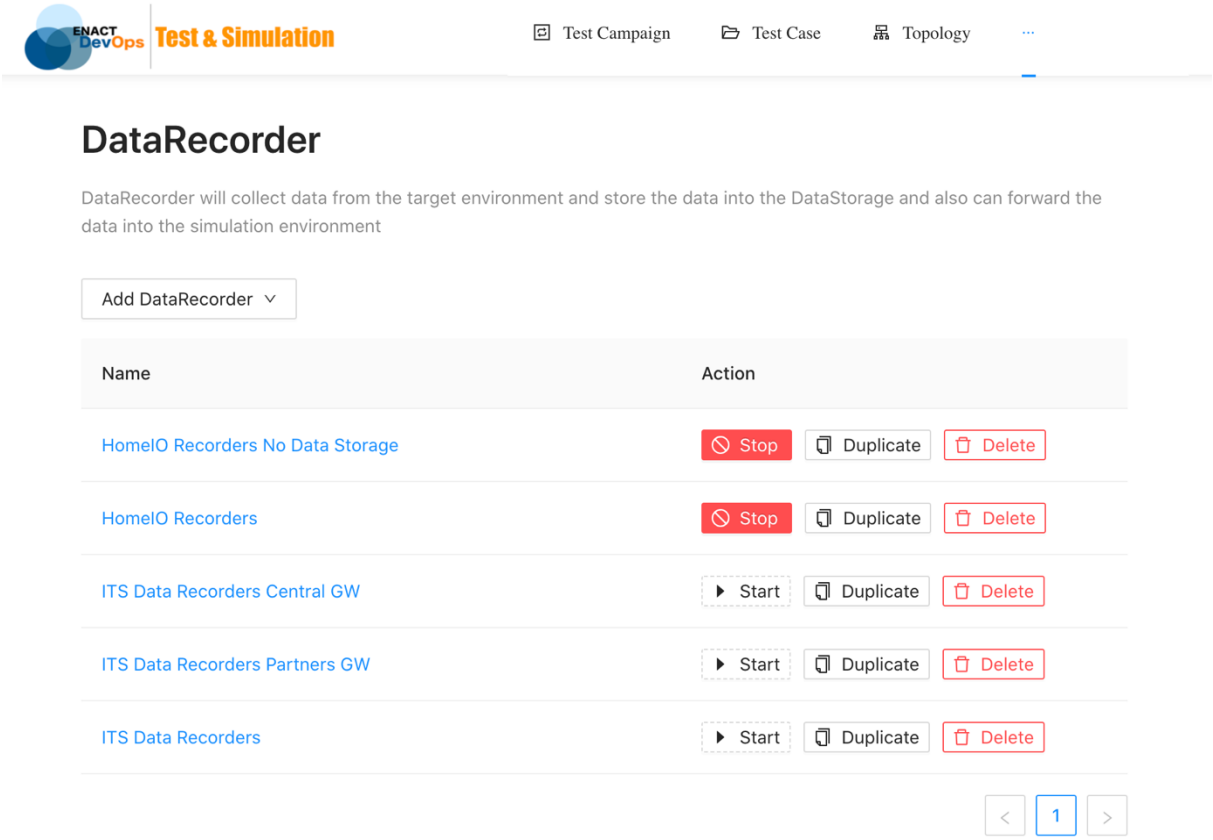


Figure 113. Data Recorder dashboard

Figure 114 illustrates the class diagram of the *Data Recorder*. The *id* and *name* are specified for each *Data Recorder*. The *source* contains the configuration to establish the connection with the source data and also the list of topics to be listened to. The *forwarder* is the configuration to establish the connection with the testing broker where the sensors' data will be forwarded to. This is an optional configuration. The *dataStorage* and the dataset indicate where and how the recorded data will be stored. The *init* method starts the recording process, and the *stop* method ends it. For each recording section, to respect the timeline between all the events, only one single dataset will be created.

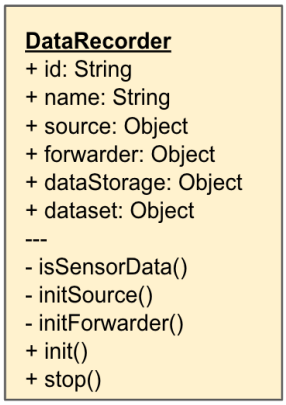


Figure 114. Data Recorder class diagram

The following JSON file content shows an example of a *Data Recorder* configuration file which includes two recorders. These two data recorders record data from different sources. In each source, the *upStreams* defines the list of topics to which the sensors publish data, the *downStreams* defines the list of topics from which the actuators receive data. All the recorded data will be stored into one data storage as a single dataset. By storing all the events of all the recorders in a single dataset, the order of all the events will be respected when replaying them. As it can be seen in the file, only the first recorder forwards sensor data to a testing system.

```
{
    "name": "HomeIO Recorders",
    "dataStorage": {
        "protocol": "MONGODB",
        "connConfig": {
            "host": "192.168.1.21",
            "port": 27017,
            "username": null,
            "password": null,
            "dbname": "homeiodb",
            "options": null
        }
    },
    "dataset": {
        "id": "homeio-dataset-02",
        "name": "HomeIO Dataset 02",
        "description": "This is a new dataset",
        "tags": [
            "recorded",
            "conflict",
            "homeio"
        ]
    },
    "dataRecorders": [
        {
            "name": "TV status recorder and Actuator",
            "id": "homeio-recorder-01",
            "enable": false,
            "source": {
                "protocol": "MQTT",
                "connConfig": {
                    "host": "192.168.1.22",
                    "port": 1883,
                    "options": null
                },
                "upStreams": [
                    "enact/sensors/cec/status"
                ],
                "downStreams": [
                    "enact/actuators/smartbox/mute"
                ]
            },
            "forward": {
                "protocol": "MQTT",
                "connConfig": {
                    "host": "192.168.1.21",
                    "port": 1882,
```

```
                    "options": null
                }
            }
        },
        {
            "name": "Call Status recorder",
            "id": "call-status-recorder",
            "enable": true,
            "source": {
                "protocol": "MQTT",
                "connConfig": {
                    "host": "192.168.1.21",
                    "port": 1883,
                    "options": null
                },
                "upStreams": [
                    "call/status"
                ],
                "downStreams": []
            },
            "forward": null
        }
    ]
}
```


The recorded data can be used as a source for simulation. It can also be mutated so that it can contain different values for a modified testing scenario. In the next section, we will explain how to generate a new dataset using a given behaviour profile.

6.2.3.5 The Regular and Malicious Data Generator

When testing the IoT system, there are many testing scenarios and cases that do not frequently occur in reality. With the real IoT system, it is almost impossible to collect the datasets for many testing scenarios. The TAS enabler provides a powerful tool to solve this problem.

The *Regular and Malicious Data Generator* module helps the developer in creating a testbed. It allows generating the data of the sensors in different scenarios. It can generate the sensor data in both normal and malicious or abnormal conditions, such as, making the temperature too high or too low. By combining multiple data, one can create a testbed that includes many incident or attack scenarios, such as, DDoS and data poisoning. All the generated data is stored in the *Data Storage* for further use.

Based on the data type and constraints on the time, values, or energy use, there are many types of abnormal behaviours that can exist as depicted on Table 13.

Table 13. Abnormal behaviours based on the data type and the constraints

Behaviour / Data Type	Boolean	Integer/Float	Integer / Float + Value Constraint	Enum	Composed
Fix value (*Always send the same value*)	Yes	Yes	Yes	Yes	Yes
Value out of range (*Send the value out of possible range*)	NA17	Yes	Yes	NA	*
Value out of regular range (*Send the value out of the regular range*)	NA	NA	Yes	NA	*
Value change out of regular step (*The data change step is out of the regular step*)	NA	NA	Yes	NA	*
Invalid value (*Send invalid value - attack to crash the system*)	Yes	Yes	Yes	Yes	Yes
Low battery (*Reduce the sending data frequency - 1/2*)	Yes	Yes	Yes	Yes	Yes
Run out of battery (*Stop sending data*)	Yes	Yes	Yes	Yes	Yes
Possible node failed (*Stop sending data after some period of time*)	Yes	Yes	Yes	Yes	Yes
Possible DOS attack (*Send data with the period less than the minimum time period*)	Yes	Yes	Yes	Yes	Yes
Possible Slow DOS attack (*Send data with the period more than the maximum time period*)	Yes	Yes	Yes	Yes	Yes

The abnormal behaviours based on the energy constraints and the reporting time constraint concern the behaviour of the sensors themselves. While the ones based on the values concern the behaviour of the measurement data captured by the sensors.

Figure 115 illustrates how a data value has been generated based on the selected behaviours of the sensor shown in Figure 116. To start, the energy constraint will be checked. There are two behaviours related to energy. When the sensor is in low battery mode, the reporting frequency can be reduced (notice that the value is in general set by the user). If the sensor is out of battery, it will stop sending data. In the next step, the time constraint will be considered. In this step, three behaviours can be selected. The *possible DOS attack* increases the reporting frequency to simulate a DOS attack. This means that the sensor will send a lot more data than what is considered normal. In some cases, if there is a constraint on the maximum delay time of the reporting data, a behaviour like a *possible slow DOS attack* can be simulated. For example, if the system expects to receive the temperature information every 5 seconds maximum, then the possible slow DOS attack will change the behaviour of the sensor, so that it will send a message every 6 seconds. Based on the time constraint, a sensor can be simulated to stop sending data for a certain period of time (*node failed*). Finally, for each measurement provided by a sensor, the

[17] NA = Not Applicable

value constraint will be checked. There are many behaviour types based on the data type of the measurement, such as, invalid value or fix value as depicted in Figure 117. Based on the selected behaviour, the *Data Generator* function will return a specific value for the measurement.

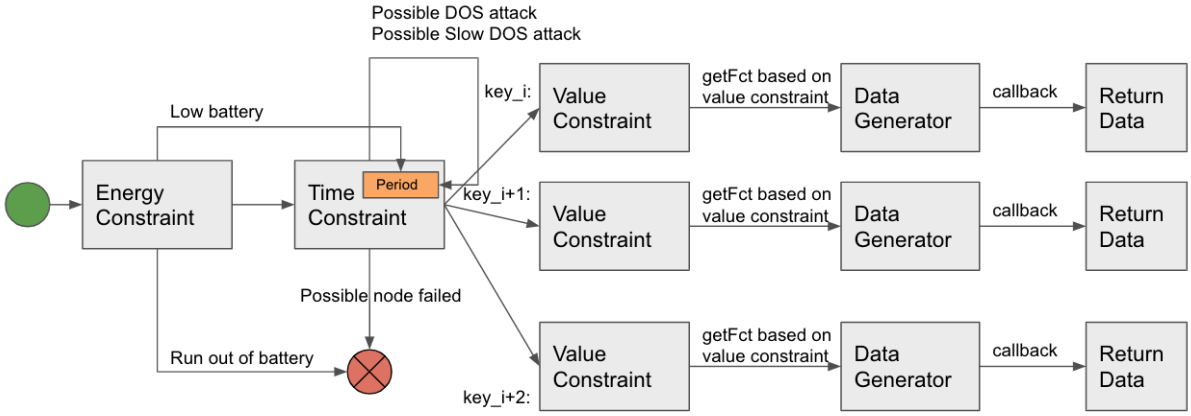


Figure 115. Data Generating Flow

Beside the sensor's behaviour, it is also possible to change the behaviour of the IoT devices. For example, the *GATEWAY_DOWN* behaviour makes the simulated IoT device stop working after a certain time. When an IoT device stops working, all the sensors and actuators belonging to that device will also stop working.

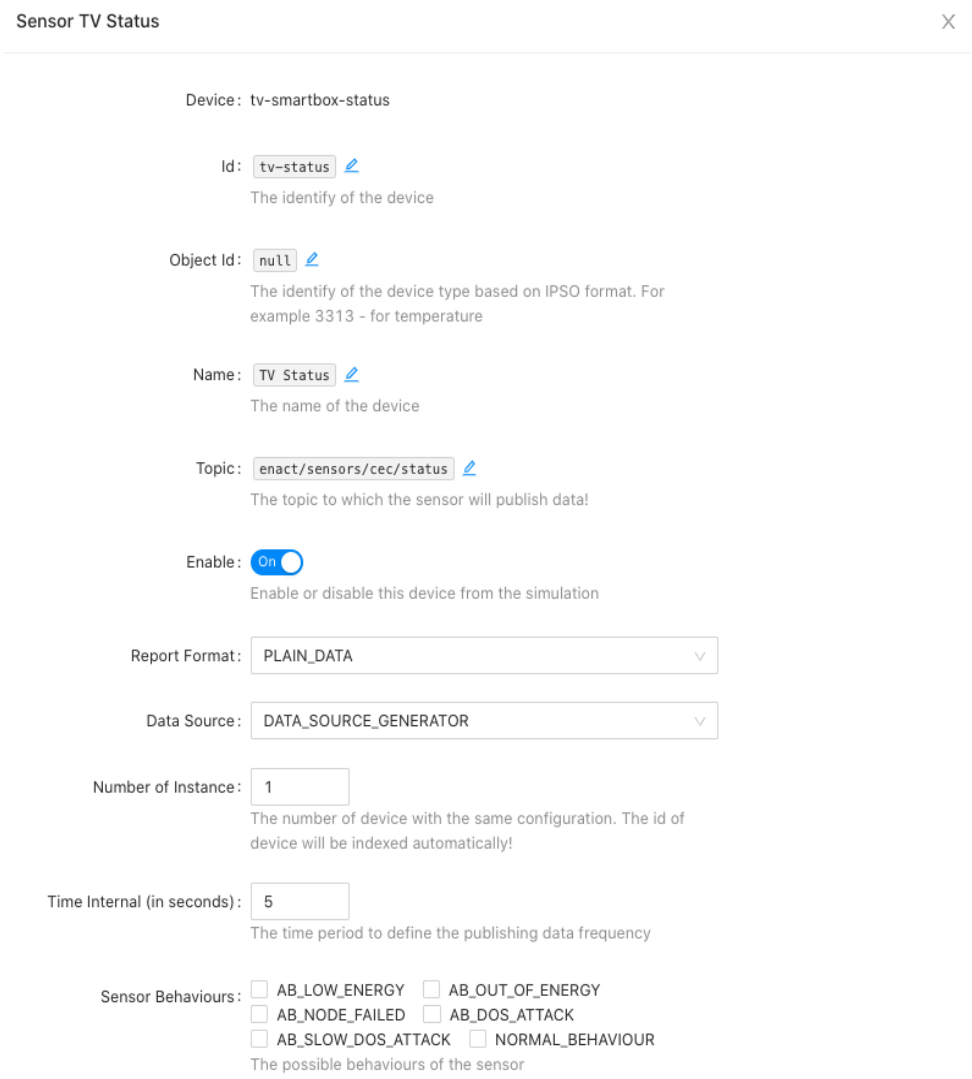


Figure 116. Abnormal behaviours of the sensor

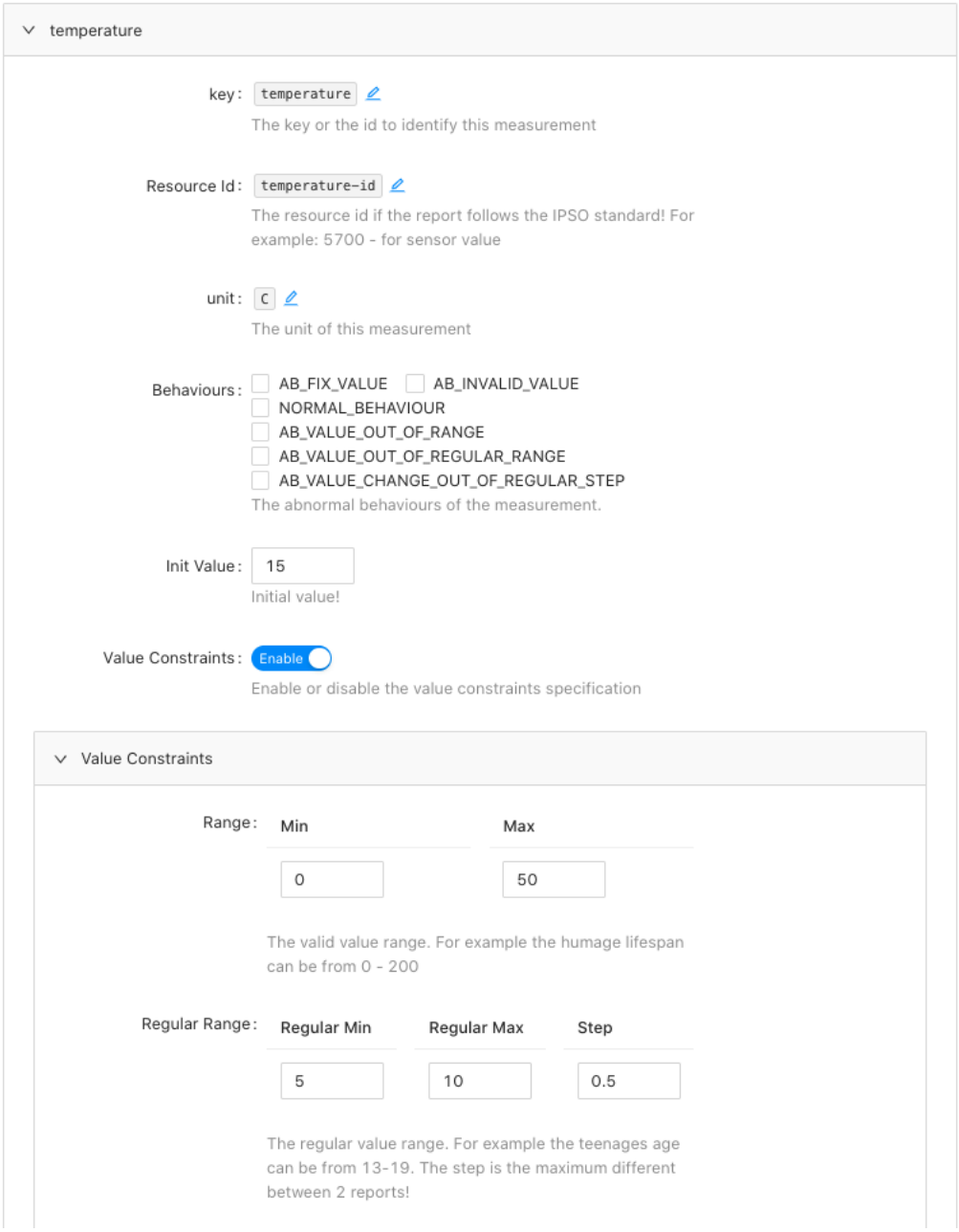


Figure 117. Abnormal behaviours of the measurement based on value constraints

6.2.3.6 The evaluation module

The *Evaluation* module collects the simulated actuator data as well as the other metrics of the system. Then, it performs the evaluation based on the testing methodology (see section 6.2.3.1 for more details).

6.2.4 *Implementation*

In this section, some of the main technical details of the implementation are presented.

6.2.4.1 The Test and Simulation Docker Image

The TAS enabler has been designed to be portable, it can be installed as a nodejs application, and is also packaged as a docker container. Figure 118 presents the communication between the modules inside a docker and between the docker container and other modules.

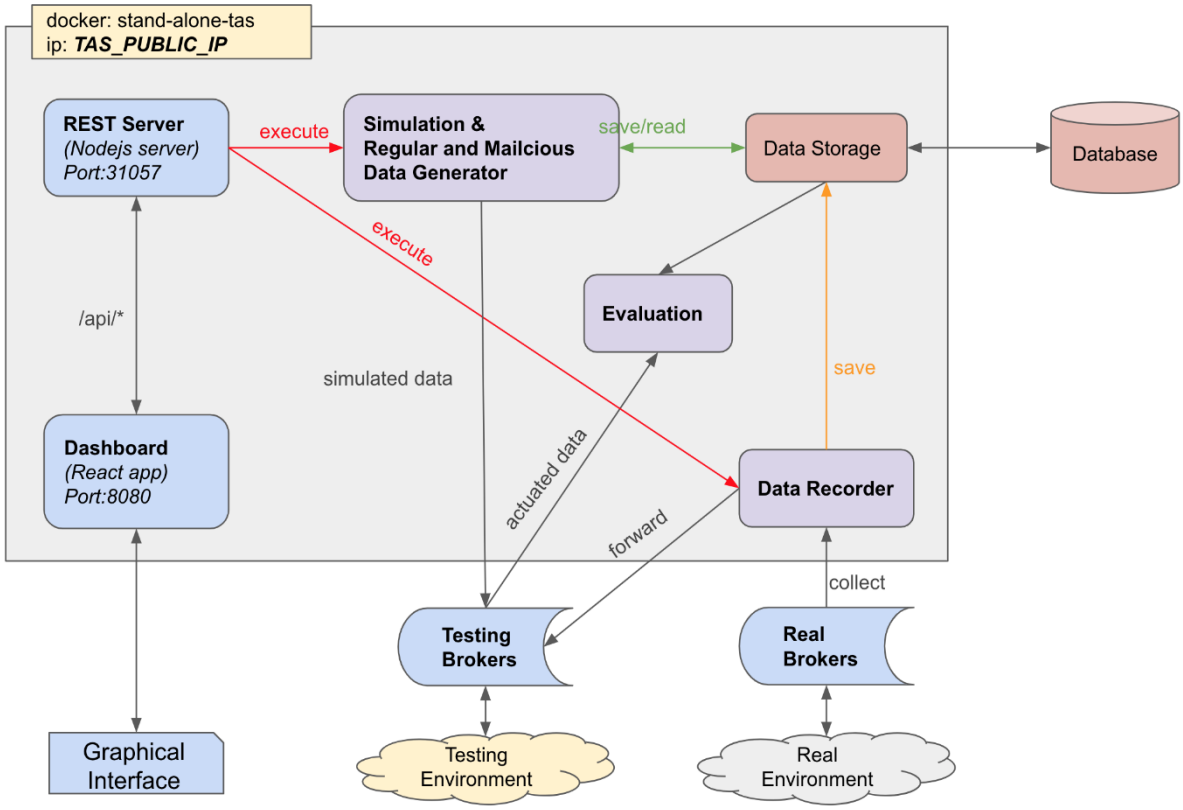


Figure 118. Test and Simulation Enabler docker image

The REST Server provides an API to interact with the tool. Via this API, the module *Data Recorder*, *Simulation*, and *Regular and Malicious Data Generator* can be executed. The *Database* is external to the docker and can be connected via the *Data Storage* module. The dashboard is the graphical interface, implemented using ReactJS [27].

6.2.4.2 Basic APIs

Table 14 presents the list of basic APIs which are provided by the tool to integrate into a DevOps cycle.

Table 14. Basic APIs to integrate into DevOps cycle

Path	Method	Data	Response
`/devopts/`	GET		Get automation testing configuration (see Figure 111)
`/devopts/`	POST	{webhookURL, testCampaignId}	Update the campaign and (or) webhookURL
`/devopts/start`	GET		Trigger the simulation and test process
`/devopts/stop`	GET		Stop the simulation and test process
`/devopts/status`	GET		Get the status of current execution

The full APIs list can be found in the appendix.

6.3 Evaluation – KPIs & Requirements

In the following section is explained how our approach addresses the requirements defined in Section 5.2.1 of deliverable D2.1 where two use cases expressed their requirements that need to be covered by

the enabler. In the Table 15 below, the *Intelligent Transportation System* use case is referred to as UC1, and the *Digital Health use case* is referred to as UC2.

Table 15. KPI and requirements

ReqID	Requirement	Description	Coverage
UC1 R1, *UC2* R1	Scalability	The simulator should easily scale the number of components that are involved in the simulation.	**Covered** by the approach where the sensor simulator and actuator simulator can be simply multiplied by creating many of their instances.
UC1 R2	Component modelling	The simulator should model virtual devices that reproduce the behaviour of real devices.	**Covered** by simulating IoT device and the sensors behaviours as well as the IoT device behaviours.
UC1 R3, *UC2* R3	Simulation of multiple sensor events	The simulator should generate signals from multiple types of sensors: – Accelerometer: ADXL362Z, SPI – GNSS: A2035H, UART – XBEE radio for RSSI: Xbee868LP, SPI – RFID: SparkFun Simultaneous RFID Reader - M6E Nano, UART – Battery monitoring (load current, battery voltage)- analog voltage on ADC inputs of energy harvesting module internal in EDI node (controller). – Analog measurement circuits, including current-voltage converter and instrumentation amplifier circuits is being developed by BOSC. Particular IC is not chosen yet.	**Covered** by the generic sensor model. Each sensor can have multiple measurements, and each measurement can have different data type.
UC1 R4, *UC2* R4	Simulation of multiple communication protocols	The simulator should simulate several communication protocols: – IEEE 802.15.4 ZigBee – IEEE 802.3	Only **covered** for the required protocols in use cases (MQTT and MQTTS).
UC1 R5	Simulation of dynamic geographical position	The simulator should simulate changes of geographical position of the system, considering possible mobile network disconnections, and other possible situations derived of the position changes of the system physical platform.	**Covered** by generating GPS sensor with the value data is the location information. However, it can be improved with the generating the trace of a movement from a location to another location.

UC1 R6, *UC2* R6	Failures simulation	The simulator should simulate the possible failures of the system. Failures can be related with networking issues, device disconnection, or fake readings.	**Covered** by simulating some abnormal behaviours such as: node failed, gateway_down.
UC1 R7, *UC2* R7	Attack simulation	The simulator should generate possible attacks to the system. Attacks include data poisoning, device disconnection, or device hijacking.	**Covered** with simulating the attacks: DOS attack, slow DOS attack. Thank to Data Recorder module, any type of attack can be performed on a testing system, then the Data Recorder will record all the events to have a dataset for testing.
UC2 R8	Real environment interoperability	The simulation should be able to be plugged into a real system so that it can interact as a real part of it.	**Covered**. The Test and Simulation enabler has been designed to be able to provide the input to any system via a broker, so technically it can be plugged into a real system as a real part of it
UC2 R9	External actors' simulation	The simulation should simulate the interaction of external actors. An external actor can be a human being or another system.	**Covered**, by using the Data Recorder module, any events in the system can be recorded (which includes any interaction of external actors), then be re-used as the dataset to test the system.

6.4 Beyond ENACT

The TAS enabler can be used for testing any IoT project where there is the need of simulating the sensor data. From a simple project like checking the smoke detection alarm to a more complex project such as *Intelligent Train System* or *eHealth*. The enabler's openness allows it to be used to provide simulated input data to any non-IoT system. For example, if we need to test a function that handles the age of a human, we can create a "sensor" with a single measurement. Then we can inject some abnormal behaviours based on the value constraints, such as, invalid value, value out of range to test how the function handles different types of input.

7 Conclusion

The goal of WP2 of ENACT is to provide support to define and ensure the trustworthy continuous delivery of Smart IoT Systems (SIS). The work package deals deployment, definition, and maintenance of the IoT oriented systems. During planning, deployment, and maintenance of trustworthy IoT systems, the tools provided by the work package ensure that the targeted system:

- Can have the exact level or risks mitigation with appropriate functions and setup decided though an innovative risk assessment strategy in a continuous manner
- Can be abstracted to a concrete modelling language which embeds trustworthiness metrics from ground up and can be translated to number of IoT solutions regardless of the technical requirements
- Can handle dynamicity associated to the IoT oriented architectures and act accordingly to situations which are foreseen in design time
- Can leverage simulation techniques to ensure that the scenarios directly affecting dimensions of trustworthiness are properly designed and that the system behaviour predictable and ready for all types environmental changes which can occur in the production at any scale.

The main objective of this report is the description of the design and implementation of all the enablers which are the core of the work in WP2. The document describes the details of the delivered software prototypes of the enablers above and presents the operation of the different methods and tools and how they can be used in the ENACT framework. The report explains the “Lab Experiment” which was used by the work package in order to showcase the use scenarios in a complete form. Next, the deliverable offers the analysis of overall achievements of each of the enabler as well as the improvements over the previously reported state. Furthermore, it provides a report of the KPI’s and evaluation status of the requirements expressed by the use cases. Finally, each enabler covers how the results of the project will be carried out beyond the project duration.

The tools are seamlessly integrated among them and with other ENACT enablers and details of final integration please refer to deliverable *D5.4 ENACT DevOps Framework- Final version*.

The usefulness and efficiency of the enablers has been evaluated in a series of tests within the ENACT use cases and the results of such evaluation are provided in *D1.5 Final evaluation and validation report* of ENACT.

Appendix A Deployment and Orchestration Approaches for IoT: An Update of the SotA

A long with the progress of the project, we have kept updating on the state of the art on IoT deployment approaches and reported the results in these publications [1-3]. We have been refreshing our work and completing a manuscript entitled "*Deployment and Orchestration Approaches for IoT: The Good, The Bad, and A Way Forward*" with the latest results until end of 2020. In a short summary, the latest results have included many recently published primary studies on IoT deployment approaches, and our in-depth analysis and discussion on the updated results. Moreover, we have extended the results by presenting the most significant commercial tools for IoT deployment such as Microsoft IoT Hub[18], Balena.io[19]. We then discus the academic approaches vs. commercial tools used by industry and highlight the research challenges to be addressed.

Appendix B A Decade of Research on Patterns and Architectures for IoT Security

We have submitted a full paper with the title "*A Decade of Research on Patterns and Architectures for IoT Security*" (with a significant extension on [39, 40]) to a journal.

In this paper, we have examined a research landscape of patterns and architectures for IoT security by conducting a systematic review. After systematically recognizing and reviewing 33 primary studies out of thousands of relevant papers in this domain, we have discovered that there is a slight rise in the number of publications addressing security patterns and architectures in the two recent years. However, our analysis has shown that security patterns are relatively "young" for the IoT domain and more efforts needed in terms of proper documentation and adoption. Indeed, we have found more papers with main contributions categorized as architectures rather than patterns. We have not seen any approach of applying systematically architectures and patterns together that can address security (and privacy) concerns not only at architectural level, but also at "weak links" at the network or IoT devices level. Most of the primary studies do not work in all the seven layers of the IoT World Forum Reference Model for IoT architecture. They mainly operate in the Physical Devices and Controller (L1), Connectivity (L2), and Application (L6) layers. There are four layers that have little coverage in terms of patterns and architectures for addressing IoT security challenges: Edge Computing (L3), Data Accumulation (L4), Data Abstraction (L5), Collaboration and Processes (L7). New IoT systems development should concentrate more on tending to security, which can be improved with progressively relevant security patterns to apply and reuse. In other words, we need to promote the utilization of patterns for IoT security (and privacy) by design.
To make security patterns for IoT approaches more viable. We consider the research collaboration between academia and industry is key in this domain. Security patterns in literature can be researched and applied in developing secure IoT systems with industrial context. Vice versa, experiences gained from securing industrial IoT systems can help to improve existing security patterns for IoT, or even new ones can emerge.

[18] https://azure.microsoft.com/services/iot-hub/
[19] https://www.balena.io

Appendix C Debugging Resource-constrained Devices using ThingML

We detail below our approach for debugging SIS, and especially resource constrained devices such as software component running on headless microcontrollers. We approached this issue in D 2.2 where we described how we used model-driven engineering to inject logging instructions into ThingML programs. ThingML is an executable modelling language, inspired by the statecharts of UML. However, by contrast with UML, ThingML includes an executable action language that enables simulation of the state machines. A ThingML model typically defines:

- Components communicating through asynchronous messages. Components expose well-defined ports, that specify the data the messages carry through their parameters.
- State machines that capture the behaviour of components, that is a set of states and transitions that describes the relationships between incoming and outgoing messages. As in UML, these state machines form a hierarchy: A state can be further decomposed into in sub states and transitions. State machines can also have explicit states defined through variables.
- Functions to encapsulate repetitive behavior happening in multiple states or transitions in the statecharts, or to abstract from platform specific details through abstract functions.
- An action and expression language to specify the body of functions and the behavior of states and transitions. This language is fundamentally a common-subset of constructs encountered in most programming languages, including Boolean and numerical algebra, control structures (conditional, iteration), variable declaration and assignment, function invocation, etc.

Being fully platform independent, the ThingML framework can convert these models into various languages such Java, Javascript, Go, POSIX C and Arduino C. This permits covering a large part of the Cloud / IoT continuum.

7.1.1 *Recalling the Approach*

As explained in D2.2, Section 3.2.3, our main objective is an automated, platform-independent and easy to use logging mechanism for ThingML developers. This logging approach aims at providing log information about the execution of their ThingML programs, in terms of ThingML concepts being executed. This approach does not intend to provide detailed information about the underlying execution of the target programs, *i.e.*, lower-level code generated by ThingML. Existing platform-specific debuggers and profilers can be used for this.

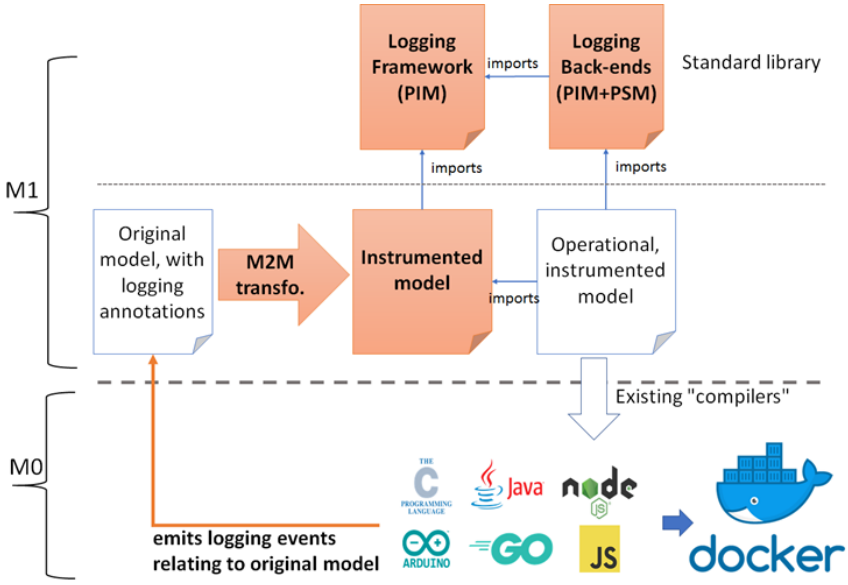


Figure 119: Overview of our model-driven, platform-independent logging approach (borrowed from D2.2)

Error! Reference source not found. above gives an overview of our approach. The main input to our approach, depicted on the left hand-side of the figure, is a "plain-old ThingML model" *i.e.*, a ThingML model that developers can specify with the current and un-modified ThingML tools. This model can be annotated with *@monitor* annotations, using ThingML's default annotation mechanism, which are used to specify what the developer wants to log. Based on these annotations, the input ThingML model will be transformed into an instrumented ThingML model, which extends a generic and reusable logging framework, depicted at the top of the figure. This instrumented model contains additional code that is woven into the original model to interact with the logging framework and log information as described in the annotations. This instrumented model (basically containing a number of component types) can then be further refined into an operational instrumented model, where a concrete back-end for the logging will be instantiated, which will be responsible for the actual storage and handling of the log events, for example locally or on a remote server. This operational model can then be "compiled" to one of the languages supported by ThingML, with no need to modify those existing compilers[20]. When executed, the generated code will emit log events as specified in the original model with logging annotations. Those events are self-descriptive, containing all the information needed to understand the execution of the program, in terms of ThingML concepts. In other words, the target programs executing at the M0 level will generate traces directly referring to concepts available at the M1 level.

Our logging approach allows logging five key information of any ThingML program, including:

- **Function calls**, where the function name, its optional return type, its optional return value and its optional list of actual parameters are recorded.
- **Property updates**, where the property name, its type, its previous and new values are recorded.
- **Events** where all incoming and outgoing events are recorded. More precisely:

[20] strictly speaking the ThingML ``compilers" are model-to-text transformations producing source code for the supported target languages.

- events emitted by a component, where the name of the message, the port on which it is sent, and its optional list of actual parameters are recorded,
- events received but discarded by a component, where the same information as for emitted events is recorded,
- events received and handled by a component, where the same information is recorded, as well as the state where the event has been handled, and the optional target state after the event has been handled.

7.1.2 *Human-readable logging using String based encoding*

Our first attempt, which we already described in D2.2, was to model event as human readable text, and to rely on string concatenation to pass all the needed information. **Error! Reference source not found.** illustrates the API of the logging framework. Each type of information is reified as a separate ThingML message, whose parameters are all of type String. In addition, two messages are defined to turn the logging on or off. Those two messages will be used by logging back-ends to decide if logs should be handled or discarded. **Error! Reference source not found.** shows a ThingML function instrumented with logging capabilities. The lines denoted by "[+]" indicate the lines the instrumentations added, whereas the lines denoted by "[-]" denote those removed.

```
component fragment LogMsgs {

  message log_on()
  message log_off()

  message function_called(
    inst : String, fn_name : String,
    ty : String, returns : String,
    params : String
  )

  message property_changed(
   inst : String, prop_name : String,
   ty : String, old_value : String,
   new_value : String
  )

  message message_sent(
    inst : String, port_name : String,
    msg_name : String, params : String
  )

  message message_lost(
    inst : String, port_name : String,
    msg_name : String, params : String
  )

  message message_handled(
    inst : String, source : String,
    target : String, port_name : String,
    msg_name : String, params : String
  )

}
```

Figure 120 String-based logging framework (borrowed from D2.2)

```
//Functions are transformed like this:
function f(a : Int16, b : Int16) : Int32 do
[+] readonly var params : String = a as String
                                 + "," + b as String
  if (a > b) do
[-]  return a - b + c()
[+]  readonly var return_exp : Int16 = a - b + c()
[+]  log!function_called(DEBUG_ID,
[+]    "f", "Int32", return_exp as String, params)
[+]  return return_exp
  end else do
[-]  return b - a + c()
[+]  readonly var return_exp : Int16 = b - a + c()
[+]  log!function_called(DEBUG_ID,
[+]    "f", "Int32", return_ext as String, params)
[+]  return return_exp
  end
end

//in any case, function calls are not affected e.g.:
a = f(1, 2)
```

Figure 121 A example of function instrumented with logging capabilities

The main challenge with this approach is the memory and CPU overhead introduce by the manipulation String objects. We refer the reader to Section 3.2.3.3 for a comprehensive overhead evaluation. Besides, this approach does not apply to resource constrained environment such as Arduino C, that does not provide a string manipulation library. To alleviate this challenge, we developed an alternative based on binary encoding.

7.1.3 *Efficient Logging using binary encoding*

The logging framework that supports binary-encoding of events implements a different approach: As shown on Figure 122**Error! Reference source not found.**, only one generic message is specified to carry log information, and knowledge about the instrumented ThingML model is assumed to be able to exploit binary logs. This new API will, by default, generate optimized binary logs, about 5 times smaller than string logs. Those binary logs not being self-contained, they require additional processing, and knowledge about the instrumented ThingML

```
enumeration LogType as Byte {
  function_called = 0
  property_changed = 1
  message_lost = 2
  message_sent = 3
  message_handled = 4
}

thing fragment LogMsgs {
 message log_on()
 message log_off()

 message log(log : Byte[], size : UInt8)
 //log[0]: one of the LogType literal
 //log[1]: ID of the component (thing) instance
 //log[2]: ID of the function/property/message
 //log[3-N]: depends on the type of payload defined by log[0]
}
```

model, to provide the same information as the string logs.

Figure 122 Alternative logging framework based on binary encoding

Binary payloads are an efficient mean to represent, store and transport data. String logs can however be better suited in the early phases of software development, to provide rapid feedback to developers. Our legacy logging approach produced string logs, but came with several drawbacks [7] which made it impractical on resource-constrained devices. Nevertheless, being able to produce string logs is useful for developers. Rather than using our legacy approach for string logging, which basically dynamically allocates strings and uses concatenation, we propose to keep the instrumentation for binary logs unchanged, and generate an additional

component, which parses binary payloads and produces strings in an efficient manner, as shown in **Error! Reference source not found.**. The goal of this generated parser is to yield string logs on the standard output, which can easily be re-directed to a file or as input to another process, from binary logs. The parser reads the values of the bytes in sequence, to classify the log and yield the corresponding string output. In other words, this generated parser provides an automated and efficient interoperability layer between optimized binary logs and human-readable string logs

```
function parse(payload : Byte[], size : UInt8) do
  readonly var log_type : Byte = payload[0]
  readonly var inst : Byte = payload[1]

  if (log_type == LogType:property_changed) do
    readonly var prop : Byte = payload[2]
    if (inst == 0) //assuming DEBUG_ID in previous listing is 0x00
      if (prop == 3) //3 (0x03) is the ID of this property
        if (size == 7) println "property_changed(MyInstance, a, Int16",
        ", ", ((payload[3] << 8 | payload[4] << 0) as Int16),
        ", ", (payload[5] & << 8 | payload[6] << 0) as Int16), ")"
        else errorln "expecting payload of size 7 for property a"
      else ... //other properties for that instance 0
    else ... //properties of other instances
  end else ... //other aspects e.g. function calls, events handled
end
```

Figure 123 Converting binary log to string log for better human readibility

Error! Reference source not found. shows the result of this transformation. As for the previous transformation, the function is annotated with a unique @id annotation. Then, if the return type of the function is not void, every return statement is instrumented similarly to how property assignments are instrument: identifiers, the actual parameters of the function and the optional return are serialized in that order. Ultimately, a logging event is emitted just before the function returns, describing the function, its parameters and optional return value.

```
//Functions are transformed like this:
function f(a : Int8, b : Int8) : Int16
[+] @id "0x23"
  if (a > b) do
[-]  return a - b + c()
[+]  readonly var return_exp : Int16 = a - b + c()
[+]  readonly var log_f : Byte[7] = {
[+]    LogType:function_called, DEBUG_ID, 0x23,
[+]    a as Byte, b as Byte,
[+]    (return_exp >> 8) & 0xFF, (return_exp >> 0) & 0xFF }
[+]  log!log(log_f, 7)
[+]  return return_exp
  end else do
[-]  return b - a + c()
[+]  //same logic as for previous "return"
  end
//in any case, function calls are not affected e.g.:
a = f(1, 2)
```

Figure 124. A ThingML function instrumented for the binary-encoding logging framework

7.1.4 *Overhead evaluation*

We use a single, representative, medium-sized ThingML program to assess our approach. This program is about 650 lines of ThingML code and is composed of 13 messages exchanged between 4 components, as well as 9 states and 23 transitions to define the sequencing of those messages. In our experiments, seven versions of that program will be used:

1. **no**: The original program without logging instrumentation
2. **string-off**: The original program is instrumented with our legacy string concatenation-based approach for logging (cf. D2.2). Though the instrumentation is indeed woven into the program, it will not be activated *i.e.* logs will not actually be produced.
3. **string-on**: Same as string-off, with instrumentation activated *i.e.*, logs are produced to the standard output, re-directed to a file.

4. **Bin-off**: The original program is instrumented with our new binary-based approach for logging. Though the instrumentation is indeed woven into the program, it will not be activated *i.e.* logs will not actually be produced.
5. **bin-on**: Same as bin-o_, with instrumentation activated *i.e.*, logs are produced to the standard output, re-directed to a file.
6. **hyb-off**: The original program is instrumented with our new hybrid approach for logging, using the binary instrumentation as well as a generated parser transforming binary logs into string logs. Though the instrumentation is indeed woven into the program, it will not be activated *i.e.* logs will not actually be produced.
7. **hyb-on**: Same as hyb-off, with instrumentation activated *i.e.*, logs are produced to the standard output, re-directed to a file.

Whenever having the logging feature woven into the original program but turning it on or off is irrelevant, or produces the same results, we will only refer to 4 programs: no, string, bin, hyb.

To get the most precise idea of the impact of our logging approach, those seven approaches will be compiled to all the languages and platforms supported by ThingML:

- **JavaScript** running on the v8 interpreter, inside a Docker container
- **Java** running on the JVM, inside a Docker container
- **Golang** compiled to a Linux binary and running inside a Docker container
- **POSIX/C** compiled to a Linux binary and running inside a Docker container
- **POSIX/C** compiled to an Arduino binary, running on an Arduino DUE, a 32-bit ARM core microcontroller with 512 KB of ash memory, 96 KB of RAM and running at 84 MHz.

Impact on size of binaries

Error! Reference source not found. shows the binary size for the five different cases, in the seven different modes.

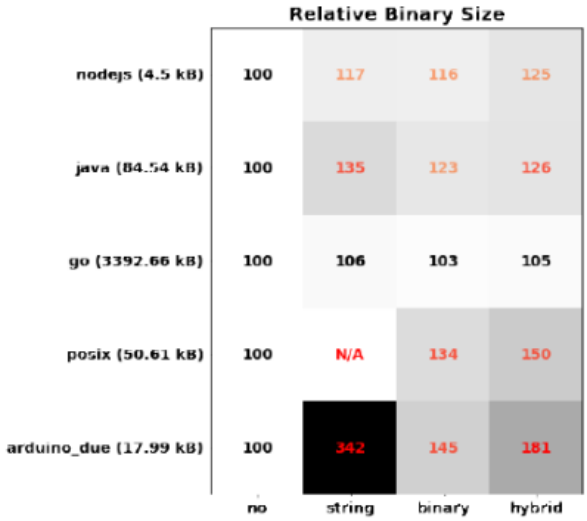


Figure 125. Size of generated binaries using for multiple logging approaches and target languages

We observe that the size of the binaries varies both according to the type of logging, and according to the targeted language. In Go the impact of our logging is very well contained, from +3% to +6% depending on the type of logging. In Java and Javascript, this overhead is significant, from +16% to +35%. This overhead is very relative, knowing that the much larger overhead induced by the JVM or the v8 interpreter is not accounted for in our measurement. More problematic are the two C-based platforms, POSIX/C for Linux and Arduino. First, our legacy string-based approach, using dynamic string allocation and string concatenation using the '+' operator

does not work for C for Linux, hence N/A in the figure. Nevertheless, it was possible to get our legacy logging approach to work on Arduino, almost without any change, simply by using the String library available in Arduino, which provides a Java-like experience, including concatenation of strings through the '+' operator. This allowed us to rapidly confirm our worries: our legacy string-based logging approach is not adapted to C in general, and in particular for resource-constrained devices. The binary instrumented with this approach is over 3 times larger (+242%) than the binary without instrumentation. Our new binary-based approach only incurs a +45% overhead, while the hybrid approach, functionally equivalent to our legacy approach, incurs an overhead of +81%. This overhead is significant but much lower than the overhead of our legacy approach, as we shall detail in Appendix 1437.1.4.

Impact on memory

Error! Reference source not found. shows the RAM consumption for the five different cases, in the seven different modes.

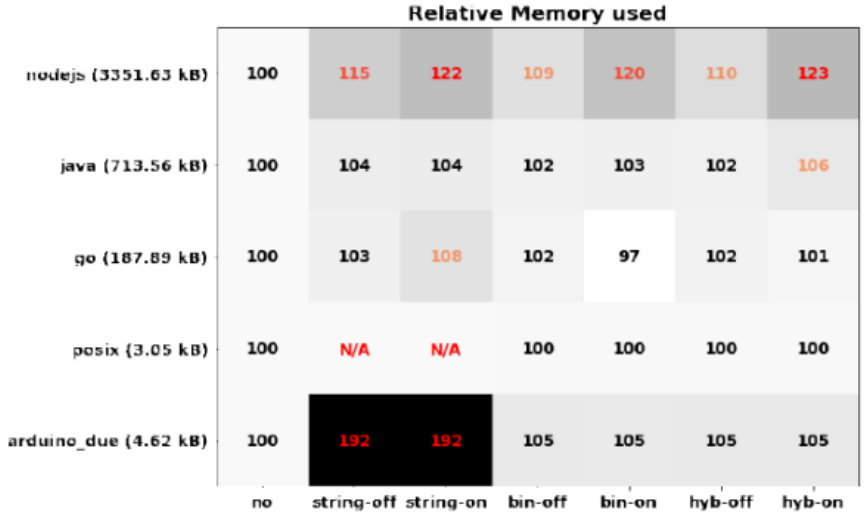


Figure 126 Memory usage for multiple logging approaches and multiple target languages

The RAM consumption significantly increases in JavaScript in all modes where logging is present, whether it is turned on or off, with an overhead of +9% to +23%. Our binary logging approach being marginally less RAM-consuming. Note that in JavaScript, and in the other garbage-collected languages (Java and Go), we do notify the garbage collector before we measure the RAM used. There is, however, no guarantee that all garbage memory has been freed when the measure is taken. We do however perform those measures in a consistent way across all programs, and we do run each program 25 times to get relevant data. The RAM consumption in Java and Go is well contained in all modes. Note that, in Go, the bin-on program is consistently using less RAM than all the other programs, including the original program (no): -3%. A detailed explanation about this fact would certainly require a good insight about the Go compiler and garbage collector, which is beyond our competencies and beyond the scope of this paper. Our best guess is that the bin-on in Go is particularly prone to efficient garbage collection *i.e.*, less garbage but not freed yet memory is measured in that mode compared to other modes. More generally, our instrumentation introduces a refactoring of the original ThingML program. For example, the actual parameters of a message to be sent are systematically extracted in local variables, to avoid side effects. This makes the code generated from the instrumented program more systematic.

Impact on execution time

Error! Reference source not found. shows the execution times for the five different platforms, in the seven different modes. Note that this experiment measures the real execution time of the

programs, and not the actual CPU time. To measure the real execution time, we take a timestamp (with ms resolution) when the ThingML program enters its initial state, and a timestamp when it reaches its initial state. The real execution time will hence be the difference between those two timestamps. This real execution time is averaged over 25 executions.

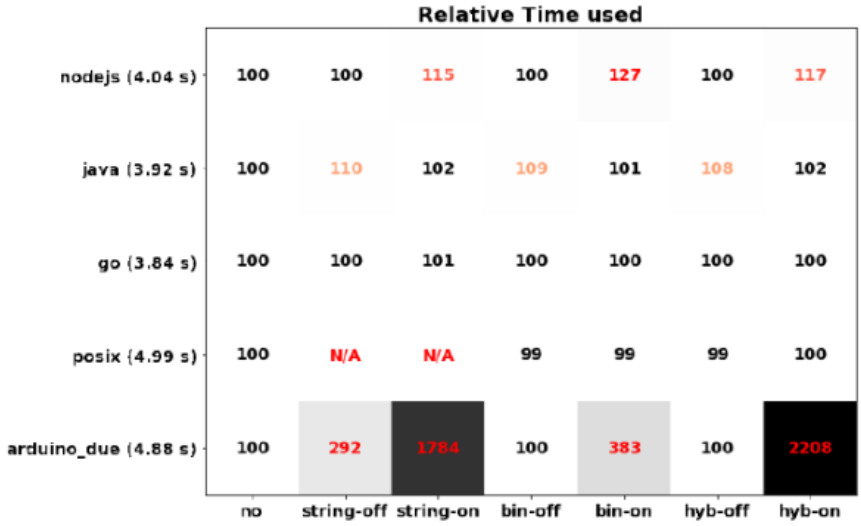


Figure 127 Execution time for multiple logging approaches and multiple target languages

In Go and POSIX/C for Linux, no significant overhead or issue is to be observed, other than the usual caveat regarding POSIX/C and the legacy string-based approach. In Java, the overhead is reasonably well-contained, from +1% to +9%. Note that turning the logging on significantly decrease this overhead. For example, in binary mode, this overhead is only +1% when logging is present and on (bin-on) but rises up to +9% when logging is turned o (bin-off). Running a profiler on both programs was not conclusive and rather suggest that the actual CPU time is lower with the (bin-off) program. In JavaScript, the overhead is significant (+15% to +27%) whenever the logging feature is activated, no matter which implementation of the logs is chosen. No overhead is observed when logging is present in the program but not activated. Again, the results on the Arduino platform are more prone to discussion. Let us start with our legacy logging approach, string-off and string-on. The simple fact of having the instrumentation embedded into the program, even though it is not activated and produces no log, slows down the execution of the program by a factor three. This is a rather hefty price for not doing anything more, from a purely functional point of view, than the original non instrumented program. The only feature the instrumented program offers is the ability to turn on the logging feature. Once turned on (string-on), the program produces logs, but become nearly 18 times slower.

Before we move to the other logging modes, it is important to understand how logs were collected in our experiments involving the Arduino Due. The Arduino Due offers no storage to store logs, and the logs were rather sent to a PC through the serial port (USB). We use the Arduino IDE running on a PC to interact with the Arduino Due. The Arduino IDE only allows a maximum baudrate of 250 kbaud per second, or 31.25 KB per second, even though the Arduino Due is capable of much higher rates. Knowing that about 2.6 MB of string logs are produced by our program, this means that nearly 90 seconds, actually most of the execution time in string-on, are spent just sending data on the serial port. We will discuss this in more details in Section 7. By contrast, our new hybrid approach, produces the very same logs as our legacy approach. In hyb-off mode, no overhead is to be observed, the instrumented program runs as fast as the original program, so long the logging is not activated. This is an improvement over our legacy approach. However, turning the logs will incur a factor 22 slow down. Again,

a large part of the time will be used to transfer data on the serial port. Nevertheless, this is slower than our legacy logging approach, the amount of logs being identical. On the Arduino Due, our new binary approach for logging is by far the most efficient, incurring no overhead in bin-off mode and a much lesser overhead than the two string-based approach when logging is activated. Again, the overhead is still large compared to other platforms, making the program nearly four times as slow when logging is activated. In binary mode however, the amount of log in our experiment is 488 KB, which means in theory that 16 seconds, or most of the time used in bin-on, are spent writing those data on the serial port, compared to 90 seconds. On other platforms, we did not comment on the time spent writing the logs to a file, as it was performed on a PC with fast SSD, and only a negligible portion of the execution time was used to write to disk.

References

1. Nguyen., P.H., et al., *Advances in deployment and orchestration approaches for IoT - A systematic review*, in *The 3rd IEEE International Congress on Internet of Things*. 2019, IEEE.
2. Nguyen., P.H., et al., *A Systematic Mapping Study of Deployment and Orchestration Approaches for IoT*, in *Proceedings of the 4th International Conference on Internet of Things, Big Data and Security - Volume 1: IoTBDS*. 2019. p. 69-82.
3. Nguyen, P.H., et al., *The preliminary results of a mapping study of deployment and orchestration for IoT*, in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. 2019, ACM: Limassol, Cyprus. p. 2040-2043.
4. Ferry, N., et al., *Continuous Deployment of Trustworthy Smart IoT Systems.* Journal of Object Technology, 2020. **19**(2): p. 16:1-23.
5. Atkinson, C. and T. Kühne, *Rearchitecting the UML infrastructure.* ACM Transactions on Modeling and Computer Simulation (TOMACS), 2002. **12**(4): p. 290-321.
6. Bergmayr, A., et al., *A Systematic Review of Cloud Modeling Languages.* ACM Computing Surveys (CSUR), 2018. **51**(1): p. 22.
7. Dearie, A. *Software deployment, past, present and future*. in *Future of Software Engineering, 2007. FOSE'07*. 2007. IEEE.
8. Ferry, N., et al., *GeneSIS: Continuous Orchestration and Deployment of Smart IoT Systems*, in *IEEE Computer Society Signature Conference on Computers, Software and Applications (COMPSAC)*. 2019, IEEE.
9. Ferry, N. and P.H. Nguyen. *Towards Model-Based Continuous Deployment of Secure IoT Systems*. in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 2019.
10. Bass, L., P. Clements, and R. Kazman, *Software Architecture in Practice*. 2012: Addison-Wesley Professional.
11. Baudry, B. and M. Monperrus, *The Multiple Facets of Software Diversity: Recent Developments in Year 2000 and Beyond.* ACM Comput. Surv., 2015. **48**(1): p. Article 16.
12. Noguero, A., A. Rego, and S. Schuster, *Towards a Smart Applications Development Framework.* Social Media and Publicity. **27**.
13. Cirillo, F., et al., *A Standard-Based Open Source IoT Platform: FIWARE.* IEEE Internet of Things Magazine, 2019. **2**(3): p. 12-18.
14. Luo, Y. *Casbin*. An authorization library that supports access control models like ACL, RBAC, ABAC. URL: https://casbin.org/ 2018 [cited 2020 November 2020]; Available from: https://casbin.org/.
15. Nguyen, P.H., P.H. Phung, and H.-L. Truong, *A security policy enforcement framework for controlling IoT tenant applications in the edge*, in *Proceedings of the 8th International Conference on the Internet of Things*. 2018, Association for Computing Machinery: Santa Barbara, California, USA. p. Article 4.
16. Frustaci, M., et al., *Evaluating Critical Security Issues of the IoT World: Present and Future Challenges.* IEEE Internet of Things Journal, 2018. **5**(4): p. 2483-2495.
17. Nguyen, P.H., S. Ali, and T. Yue, *Model-based security engineering for cyber-physical systems: A systematic mapping study.* Information and Software Technology, 2017. **83**: p. 116-135.
18. Nguyen, P.H., et al. *A Systematic Review of Model-Driven Security*. in *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*. 2013.
19. Nguyen, P.H., et al. *SoSPa: A system of Security design Patterns for systematically engineering secure systems*. in *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2015.
20. Lúcio, L., et al., *Chapter 3 - Advances in Model-Driven Security*, in *Advances in Computers*, A. Memon, Editor. 2014, Elsevier. p. 103-152.
21. Nguyen, P.H., et al., *An extensive systematic review on the Model-Driven Development of secure systems.* Information and Software Technology, 2015. **68**: p. 62-81.

22. Myrbakken, H. and R. Colomo-Palacios. *DevSecOps: A Multivocal Literature Review*. 2017. Cham: Springer International Publishing.
23. Metzger, A., *Cyber physical systems: Opportunities and challenges for software, services, cloud and data.* NESSI White Paper, 2015.
24. Gateway, E. *Express Gateway*. 2020; Available from: https://www.express-gateway.io/.
25. Gérald, R., et al. *An Actuation Conflicts Management Flow For Smart IoT-based Systems*. in *IEEE International Conference on Internet of Things: Systems, Management and Security (IoTSMS 2020)*. 2020.
26. Lavirotte, S., et al., *IoT-based Systems Actuation Conflicts Management Towards DevOps: A Systematic Mapping Study*, in *Proceedings of the 5th International Conference on Internet of Things, Big Data and Security - Volume 1: IoTBDS*. 2020, SciTePress. p. 227-234.
27. Tigli, J.-Y., *DevOps for IoT, Challenges and Recent Advances - Actuation Conflict*, in *IFIP Internet of Things (IoT) Virtual Conference*. 2020: Virtual Conference.
28. *Theory of Modeling and Simulation (Third Edition)*. 2019: Academic Press.
29. Kiczales, G., et al. *Aspect-oriented programming*. 1997. Berlin, Heidelberg: Springer Berlin Heidelberg.
30. Cimatti, A., et al. *NuSMV 2: An OpenSource Tool for Symbolic Model Checking*. 2002. Berlin, Heidelberg: Springer Berlin Heidelberg.
31. Zeigler, B., H. Prähofer, and T. Kim. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. 2000.
32. Sehili, S., et al. *Discrete Event Modeling and Simulation for IoT Efficient Design Combining WComp and DEVSimPy Framework*. in *SIMULTECH*. 2015.
33. Vangheluwe, H.L.M. *DEVS as a common denominator for multi-formalism hybrid systems modelling*. in *CACSD. Conference Proceedings. IEEE International Symposium on Computer-Aided Control System Design (Cat. No.00TH8537)*. 2000.
34. von Rueden, L., et al. *Combining Machine Learning and Simulation to a Hybrid Modelling Approach: Current and Future Directions*. 2020. Cham: Springer International Publishing.
35. Gonnin, T., et al., *Actuation Conflict Management Enabler for DevOps in IoT*, in *10th International Conference on the Internet of Things Companion*. 2020, Association for Computing Machinery: Malmö, Sweden. p. Article 18.
36. Adjih, C., et al. *FIT IoT-LAB: A large scale open experimental IoT testbed*. in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. 2015.
37. Sanchez, L., et al., *SmartSantander: IoT experimentation over a smart city testbed.* Computer Networks, 2014. **61**: p. 217-238.
38. Fuller, A., Z. Fan, and C. Day, *Digital Twin: Enabling Technologies, Challenges and Open Research*. 2020.
39. Rajmohan, T., P.H. Nguyen, and N. Ferry. *Research Landscape of Patterns and Architectures for IoT Security: A Systematic Review*. in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2020.
40. Rajmohan, T., P.H. Nguyen, and N. Ferry. *A Systematic Mapping of Patterns and Architectures for IoT Security*. in *IoTBDS*. 2020.