Title:	*Risk-driven Continuous Delivery of Trustworthy Smart IoT Systems — First Version*
Authors:	*Jacek Dominiak (BAW), Nicolas Ferry (SINTEF), Elena González (BAW), Stéphane Lavirotte (CNRS), Brice Morin (SINTEF), Victor Muntés (BAW), Phu H. Nguyen (SINTEF), Alexander Palm (UDE), Angel Rego (TECNALIA), Jean Yves Tigli (CNRS)*
Editors:	*Nicolas Ferry (SINTEF), Phu H. Nguyen (SINTEF)*
Reviewers:	*Erkuden Rios (TECNALIA), Uģis Grīnbergs (BOSC)*
Identifier:	*Deliverable # D2.2 v1.0*
Nature:	*Report*
Date:	*30 June 2019*
Status:	*Delivered*
Diss. level:	*Public*

Executive Summary

D2.2 provides the first 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.

The research leading to these results has received funding from the European Community's H2020 Programme under grant agreement n° 780351 (ENACT).

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
01 December	Initial	Nicolas Ferry (SINTEF)	Outline and summaries
13 February	Initial	Nicolas Ferry (SINTEF)	Introduction
13 May	Draft	Nicolas Ferry, Brice Morin, Phu Nguyen (SINTEF)	Almost complete version of section 3
24 May	Draft	Jean-Yves tigli, Stéphane Lavirotte (CNRS)	Almost complete version of Section 4
4 June	Draft	Nicolas Ferry, Phu Nguyen (SINTEF)	Final version section 3
4 June	Draft	Jacek Dominiak	Draft version section 2
5 June	Draft	Jean-Yves tigli, Stéphane Lavirotte (CNRS)	Final version section 4
5 June	Draft	Elena González (Beawre)	Draft version section 2
5 June	Draft	Victor Muntés (Beawre)	Review draft version section 2 + further contributions to section 2
11 June	Draft	Nicolas Ferry (SINTEF)	Final version of Section 1 and 6
11 June	Draft	Jean Yves Tigli (CNRS)	Update Section 4 after WP2 internal review
19 June	Draft	Erkuden Rios	Internal Review
25 June	Draft	Ugis Grisberg	Internal Review
27 June	Final	Nicolas Ferry (SINTEF), Phu Nguyen (SINTEF), Victor Muntes (Beawre), Jean Yves Tigli (CNRS)	Update based Internal Reviews

Contents

1 Introduction

This document presents current achievements in building the first 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 Section 1.1, we give an overview of the context of objectives of this work. Section 1.2 highlights the main achievements of our work presented in this document. Finally, we present in Section 1.3 the structure of the main content in this document.

1.1 Context and objectives

In order 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 will deliver 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.**

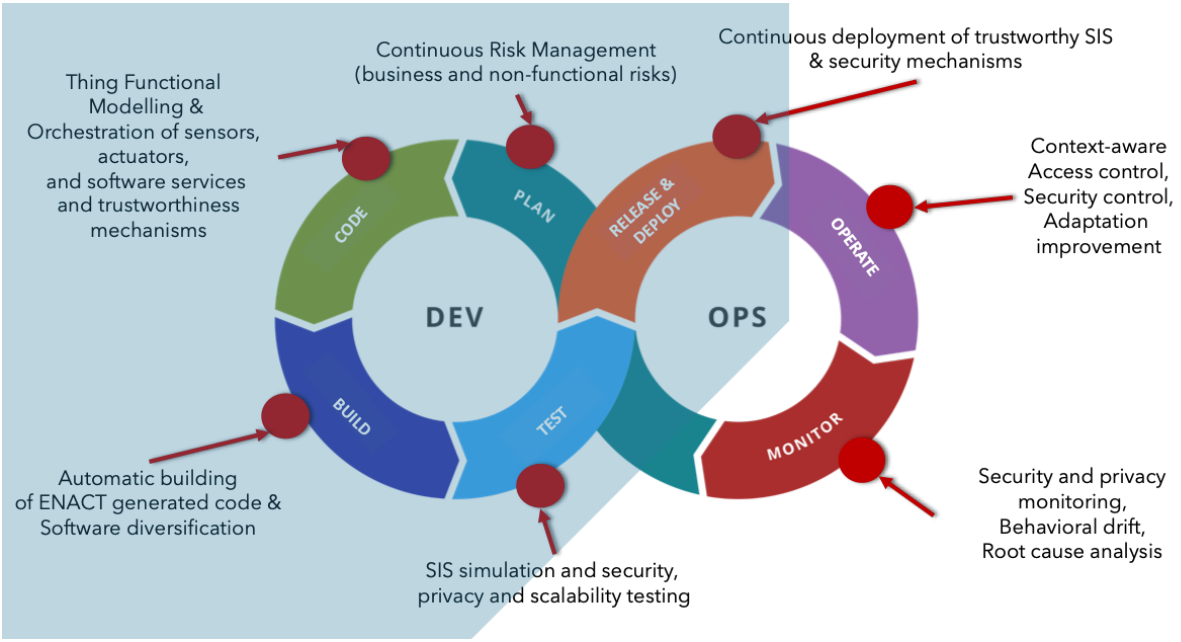


Figure 1. WP2 focuses on the Dev part of the DevOps process.

In particular, WP2 will develop four enablers:

1. The ENACT **Risk Management** Enabler: This enabler will support 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 will facilitate the development and continuous deployment of trustworthy SIS, allowing decentralized processing across heterogeneous IoT, edge, and cloud infrastructures. GeneSIS includes: *(i)* a domain-specific modelling language to model the orchestration and deployment of SIS; and *(ii)* an execution engine that supports the orchestration of IoT, edge, and cloud services as well as their automatic deployment across IoT, edge, and cloud infrastructure resources.
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 will support the identification, analysis and resolution of actuation conflicts.
4. The ENACT **Test and Simulation** Enabler: IoT systems need to cope with the uncertainty related to the physical world (*e.g.*, communication links may fail, nodes may run out of battery, etc.). The delivery model advocated to manage this uncertainty should provide proper support to assess the system's behaviour and trustworthiness early in the life cycle. ENACT will deliver an enabler to test smart IoT systems.

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.

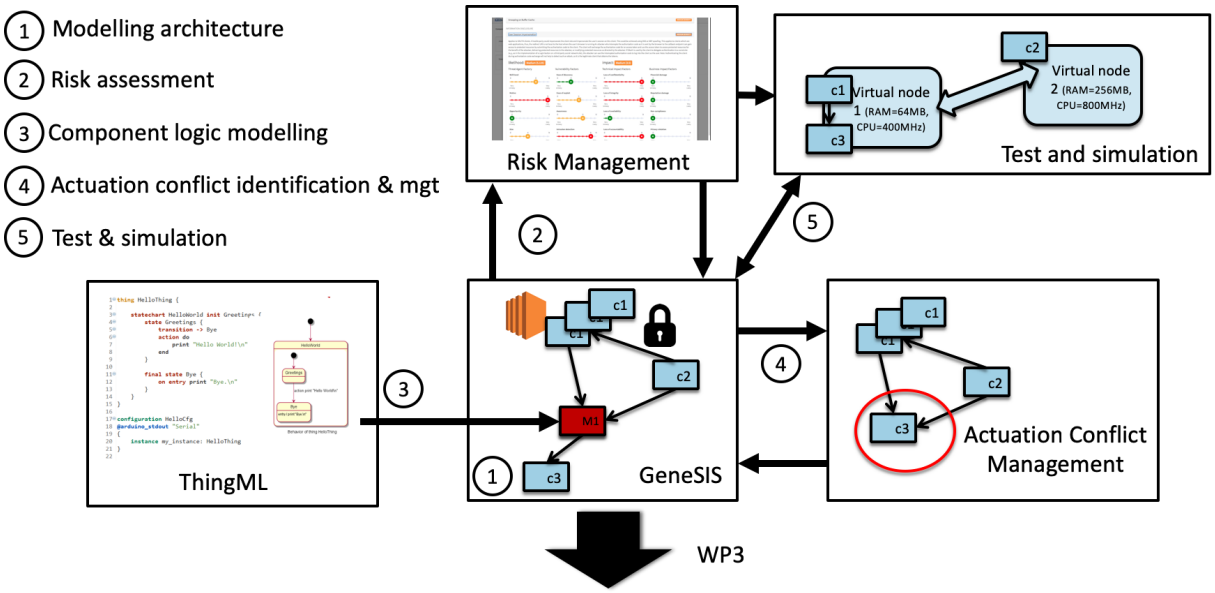


Figure 2. An example of workflow between WP2 tools

Deliverable D2.2 consists of *(i)* first 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; together with *(ii)* this document, which provides a technical description and the necessary documentation to use each of the enablers.

1.2 Achievements

The following table summarizes the main achievement of WP2 for the release of D2.2.

Table 1. Table of achievements

Objectives	Achievements
Provide first version of each enabler of WP2	Based on the theoretical foundations gained in D2.1 we developed initial versions of the different WP2 enablers. This includes not only the actual source code of the tool but also the specification of the models manipulated by the tool. All the implementations are available online in the ENACT repository. More precisely: • The initial design and implementation of the Risk management enabler, which provide users with a way to express any type of risks and enablers to relate the risks

	among them. It adopts a Kanban-like presentation to show the status of the risk analysis process and a mock-up of a dashboard to detect exceptions related to the risk analysis process. • The initial implementation of the Orchestration and deployment enabler (modelling language and execution environment), supporting deployment over IoT, Edge and Cloud infrastructure. Initial integration with SMOOL, FIWARE, and Microsoft IoT Hub. • ThingML has been integrated together with the Orchestration and deployment enabler. Moreover, ThingML has been with monitoring mechanism to facilitate the debugging of ThingML programs, including on small devices, *i.e.*, logs at the ThingML level in a platform independent way. In the future, these logs will be accessible through the Orchestration and deployment enabler. • The initial implementation of the Actuation Conflict Management enabler. This includes a first implementation of a tool to support the design of custom actuation conflict managers with guarantees on logical properties. A first implementation of the tool to identify actuation conflicts in large-scale SIS. First off-the shelf actuation conflict managers. • The initial conceptual design of the Test and simulation enabler.
Provide description of conceptual solution of each tool	We developed a conceptual solution for each tool as described in this deliverable, which laid the foundation for the corresponding tool implementation. In addition, we evaluated the status of the enablers with respect to the requirements defined in D2.1.
Provide documentation	To guide the use of the enablers, they are released together with a first version of the online documentation available on the ENACT git repository (https://gitlab.com/enact). In particular, this includes: a README detailing how to install, set up and start the enabler, at least one tutorial, and a set of examples.

1.3 Structure of the document

The remainder of the document follows the structure of WP2 and is composed of the following four Sections. Section 2 focuses on the enable for the agile and continuous risk management of SIS. Section 2 presents the enabler for the continuous orchestration and deployment of SIS. Section 4 describes the status of the enabler for the management of actuation conflicts. Section 5 covers the presentation of the test and simulation enabler. In each of these sections, an overall presentation of the enabler is provided before the main technical results and highlight are presented. Thus, an evaluation of the status of the enabler with respect to the requirements defined in D2.1 and D1.1 is provided. Finally, Section 6 draws some conclusion. At the end of the document appendices provide details about the implementations of the enablers as well as

some initial guidelines to use the enablers. It is worth noting that for each of the enablers, more detailed instructions and tutorials can be found in their code repository in the ENACT git repository (https://gitlab.com/enact)

2 Agile and Continuous Risk Management

2.1 Overall presentation of the enabler

Risk Management has been a center 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 which need to be included in the DevOps process.

The novelty of the tool comes from the 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. In particular, we try to solve several different challenges presented in [Mun18], 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. 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 will also consider this aspect and provide mechanisms for multi-asset vulnerability and threat definitions.

2.1.1 *Motivation*

We devote a brief subsection to explain the usual scenario we may find in a software/IT company when it comes to managing risks. This scenario is particularly relevant in highly regulated markets where companies need to comply with existing standards such as ISO27001 for security for instance or they need to prove that they follow a risk-based approach as required by GDPR.

Companies have risk management owners in the organization. The exact role they play may depend a lot on the type of company and the type of requirements this company may have. For instance, the company may have a Chief Security Officer (CSO), a Data Privacy Officer (DPO) as requested in some situations by GDPR, etc. In large organization, it is common to have risk analysts to support the risk management process, while in SMEs this is typically a role that staff playing other roles play. Companies that are truly following DevOps principles, tend to make

[1] H2020 MUSA project: https://www.musa-project.eu/

decisions around risk in a collegial manner, involving product owners, risk analysts, developers, operations, etc.
In this context, companies need to face many challenges, including coping with rapid software evolution, the need to obtain and keep relevant certificates to gain the trust of their customers, the need to prepare for any unwanted incident that may negatively impact their businesses, etc. For this, they need to prepare internal policies to describe their procedures to handle risk management. Among these procedures, frequent meetings are usual where they need rapidly understanding the status of risks in their projects, as well as, to set up priorities to lead to minimize risks as well as to strengthening their businesses. When these needs are crossed with IoT they become even more significant. Continuous evolution of technology in the IoT makes it even more complex to have a long-term risk management plan. Requirements change, technology changes, and companies need to adopt agile software development processes. With this, the need for continuous risk management points to which it is probably the only effective strategy.

The ENACT Risk Management Enabler is facing the challenge to support such organizations. One of the big gaps that we are planning to fill in this project is the lack of tools that avoid continuous control of risks. Commercial tools are in general focused on analysing risks at design time, defining some mitigation actions and approving the risk management plan. However, companies have little control on the level of implementation of such mitigation actions or controls and their actual effectiveness. Besides, many companies fill this gap by using manual procedures based on storing all the information in spreadsheets like Microsoft Excel. While this approach may be effective in small projects where risks can be still controlled manually, they rapidly turn inefficient as projects or teams grow. Even if projects are kept relatively small, duration of projects force for revising risk management plans several times and strategies based on simple tools or spreadsheets become ineffective.

Our goal is to provide a tool that fills some of these gaps. Figure 3 shows the high-level description of the envisioned processes. Our enabler supports a company developing software in the form of a product or SaaS (in the context of ENACT, connected to a trustworthy IoT system). Each product will have somebody playing the role of the product owner, who will supervise the whole development of that product. Apart from this role, and in particular for companies building systems in highly regulated markets, the company will typically have a risk owner. This role may happen in different forms: it may be a Chief Security Officer, a Risk analyst, a Data Privacy Officer, etc. In any case, this role may be the owner of the risk management process and strategy. This role may also take the form of a committee, composed of different people playing different roles, and bringing different perspectives, including Chief Product Officers, Chief Technology Officers or Chief Operation Officers. Risk owner will typically start and contribute and monitor the risk management process. Engineers may also be involved in this process including developers and operators, to empower a DevOps approach. Architects and Product Owners will also be involved in the risk management process. An efficient risk management process will help the organization to understand risk level associated to detected risks and prioritize the implementation of mitigation actions (or treatments or controls) in the form of new product features. In the continuous monitoring process, it will be important to monitor the implementation of the mitigation actions, and if data is available, the effectiveness of the treatments. Finally, risk management owner(s) needs to report the status of risk management. Reasons may be multiple, ranging from the willingness of the company to be compliant with existing standards and regulations to an executive board being interested in

enforcing a strict control on risks and using this information to improve decision-making processes.

We assume we are offering a tool for a software company that offers Trustworthy IoT solutions. This company needs to gain trust most probably because it is dealing with data that may be crucial about users. Following the TellU use case, this may be health data about the patients at home who are remotely monitored. Trust is not a matter of security (for instance by proving compliance with ISO 27001), but it includes many other aspects such as privacy in this case (especially since May 2018, when GDPR entered into force), or even safety in this and other scenarios.

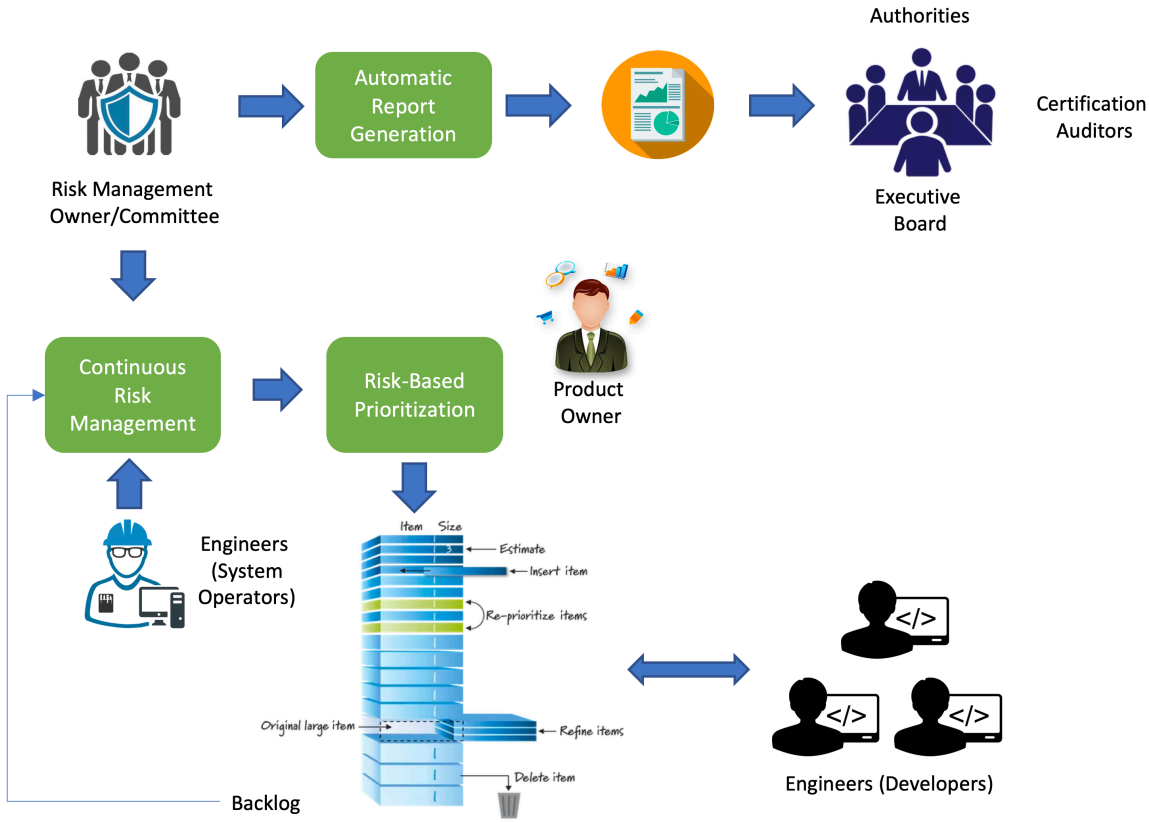


Figure 3. High level overview of the tasks around risk management that are relevant in the context of ENACT

There are several standards and certificates that include risk management as an essential aspect for both security and privacy. While ENACT has a strong focus on security, privacy is also important and probably less understood. Even when thinking about privacy, there are still two viewpoints: the security viewpoint and the privacy viewpoint. The security viewpoint focuses on the protection of the assets that are involved in the protection of personal data, also known as Personally Identifiable Information (PII) in non-European legal frameworks, while the privacy viewpoint focuses on the impact to the privacy of data subjects. Figure 4 shows two of the most significant standards considering privacy from a risk analysis perspective, both from the security and the privacy viewpoint. Our tool should be able to support security standards risk management procedures and help our users to show compliance with these standards.

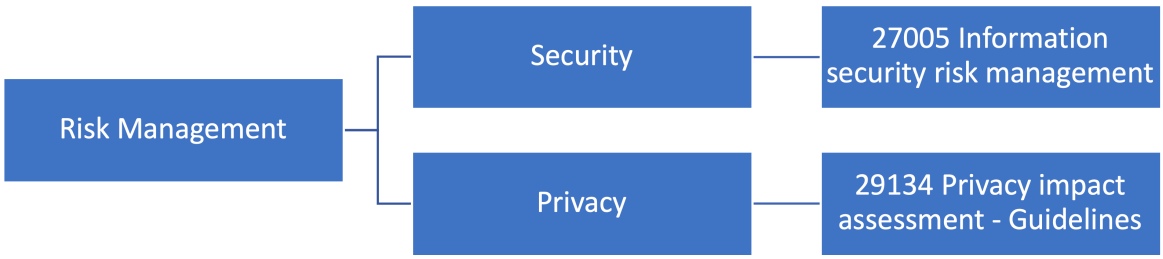


Figure 4. Landscape of current standards facing privacy from different viewpoints and focused on risk management.

2.1.2 *Background tool architecture: MUSA*

ENACT's Risk Management enabler is based on the MUSA Risk Assessment module, developed in the MUSA project to support the creation of multi-cloud applications by helping decision makers to make the right decisions. In the context of MUSA, the risk assessment module fed both the MUSA Cloud Service Selection tool, where selection of services was done following a risk-based approach, and the MUSA SLA generator[2]. Figure 5describes the process flow of the Risk Assessment module.

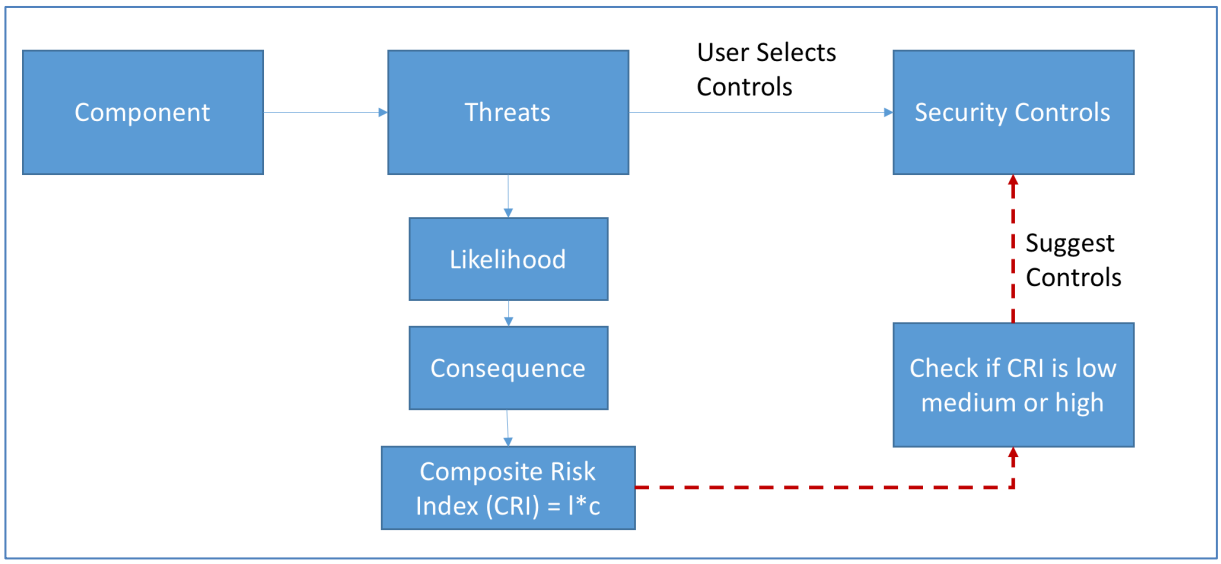


Figure 5. Outline of Risk Assessment module flow with MUSA framework (Design Time) (Extracted from MUSA D3.3)

The risk management methodology used to build this tool was inspired by CORAS [1] and the risk-based support system developed in the MODAClouds[3] project. In order to assess risks related to the different components of a multi-cloud application, the MUSA Risk Assessment module uses a risk model based on STRIDE[4], the OWASP[5] threat risk modelling, as well as OCTAVE[6]. Users choose among the threats in a threat catalogue that were potentially affecting a particular component of the multi-cloud application, as shown in Figure 5. Besides, the

[2] https://www.musa-project.eu/content/service-level-agreement-support
[3] http://www.multiclouddevops.com/#MODAClouds
[4] https://web.archive.org/web/20070303103639/http://msdn.microsoft.com/msdnmag/issues/06/11/ThreatModeling/default.aspx
[5] https://www.owasp.org/index.php/Application_Threat_Modeling
[6] https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=8419

MUSA Risk Assessment tool allows users to create their own definitions for threats, risks, etc at every stage of the risk assessment process. Once the threats are selected, they are automatically classified in the STRIDE categories (Spoofing identity, Tampering, Repudiation, Information disclosure, Denial of service and Elevation of privilege). Users are required to provide the likelihood and impact of each threat and the Composite Risk Index (CRI) of each threat is evaluated:

$$CRI = Likelihood * Impact$$

The sub-values influencing CRI are grouped by the type of factors and represented by the value in a scale of 0-9 where 0 represents a very unlikely scenario and 9 represents a very high likelihood of the factor to occur. After risk assessment, risks requiring treatment (high and medium risk level) are identified. In MUSA, a Threat Catalogue was provided establishing a mapping between that links security controls with risks. Based on this mapping, the required controls are obtained for the risks selected by users. These controls are then presented to the user as suggested but the user is free to extend the choice to all the available security controls if desired. Selected controls are further mapped to the CCM (Cloud Control Matrix) controls from Cloud Security Alliance (CSA). These controls are later used in the MUSA Framework for the cloud service selection. Users are finally requested to approve acceptance of the level of risk mitigation status. Within the MUSA Risk Assessment tool, we leverage ROAM model risk mitigation classification. ROAM is a common agile management risk mitigation classification and stand for:

- Resolved - the risk has been answered and avoided or eliminated.
- Owned – the risk has been allocated to someone who has responsibility for doing something about it.
- Accepted - the risk has been accepted and it has been agreed that nothing will be done about it.
- Mitigated - action has been taken so the risk has been mitigated, either reducing the likelihood or reducing the impact.

Note that only threats with status Accepted and Mitigated are considered as fully addressed. When the status is Owned the risk mitigation analysis must continue. When the status is Resolved, the corresponding threat is considered no longer relevant.

The MUSA Risk Assessment tool also proposes a new agile risk analysis framework to facilitate the creation of tools for agile risk management. This framework addresses the four challenges described above. With this framework, we facilitate translating traditional risk analysis practices for software development to agile software development contexts, allowing for continuous risk analysis and enabling stakeholders in agile teams to collaborate. MUSA tool uses a pull system in the style of Kanban, where the status of each asset with respect to a predefined risk analysis methodology is expressed through the different columns in the Kanban board. This makes the tool agnostic to any specific risk analysis methodology.

2.2 Technical presentation and highlights

In this section, we present technical aspects of the ENACT Risk Management enabler. In particular, we present the Risk Management methodology that we implement in this enabler and the architecture of the enabler with an initial description.

2.2.1 *Risk Management Methodology*

The backbone of any risk management tool is the methodology used for risk management. While all risk management methodologies presented in the literature and accepted by the international community through standards, scientific work or other best practices are similar, they also differ in different aspects. In this subsection, we aim to discuss some of the most well-known risk management methodologies and adapt them into a proposal that fits ENACT's project requirements.

In order to make this exercise, we explored best practices in industry and considered previous related FP7 and H2020 projects (in particular MODAClouds and MUSA) to come up with a proposal for ENACT. In particular, we considered the following approaches:

- **Risk management methodologies used in MODAClouds and MUSA (and CORAS methodology implicitly)**: MODAClouds risk management methodology has a strong influence from the CORAS methodology [1]. The methodology implemented in these projects, proposed a simplified version of the CORAS methodology to favour tools usability. We take this as one of the references and the starting point for ENACT.
- **ISO 31000:2018**[7]: ISO 31000:2018 provides guidelines on managing risk faced by organizations. The application of these guidelines can be customized to any organization and its context. This standard provides a common approach to managing any type of risk and is not industry or sector specific. Therefore, it can be used throughout the life of the organization and can be applied to any activity, including decision-making at all levels. Because of the fact that it is the most generic standard to describe risk management activities and it is agnostic to a particular context, we take it as a general reference for ENACT's Risk Management enabler.
- **ISO/IEC 27001:2013**[8]: ISO/IEC 27001:2013 specifies the requirements for establishing, implementing, maintaining and continually improving an information security management system within the context of the organization. It also includes requirements for the assessment and treatment of information security risks tailored to the needs of the organization. The requirements set out in ISO/IEC 27001:2013 are generic and are intended to be applicable to all organizations, regardless of type, size or nature. As an example, Tellu was recently certified under this standard, as an essential requirement to sell e-health applications using IoT infrastructures.
- **ISO/IEC 29134:2017**[9]: ISO/IEC 29134:2017 gives guidelines for: (i) a process on privacy impact assessments, and (ii) a structure and content of a PIA report. It is applicable to all types and sizes of organizations, including public companies, private companies, government entities and not-for-profit organizations. ISO/IEC 29134:2017 is relevant to those involved in designing or implementing projects, including the parties operating data processing systems and services that process PII.

As an example of the comparisons performed among existing methodologies for risk management, in Figure 6, we show a visual summary of the main steps followed by the risk management methodology in MUSA and the steps suggested in ISO 31000:2018 and in

[7] iso.org/standard/65694.html
[8] iso.org/standard/62289.html
[9] https://www.iso.org/standard/62289.html

ISO/IEC 29134:2017. While the vocabulary is not identical, the processes are very similar, and we were able to establish reasonable mappings among all the processes. For instance, in MUSA assets had to be defined and threats were identified with respect to those assets. We have also added a step to detect vulnerabilities, following CORAS recommendations, although we consider this step optional. In ISO 29134, the definition of assets and vulnerabilities is quite ambiguous, but they put the emphasis in the description of risk sources. Both methodologies or descriptions define threats (also called unwanted incidents in CORAS) and then risks. In general, a risk is to be considered an unwanted incident that have been assessed as a risk, *i.e.* the likelihood and the impact or consequence have been evaluated. In ISO 20134, the analysis of impacts is treated separately, but in the rest of standards, this is usually part of the risk analysis step. Some methodologies talk about treatments, while some other talk about controls. In general, these are all different terms to refer to mitigation actions.

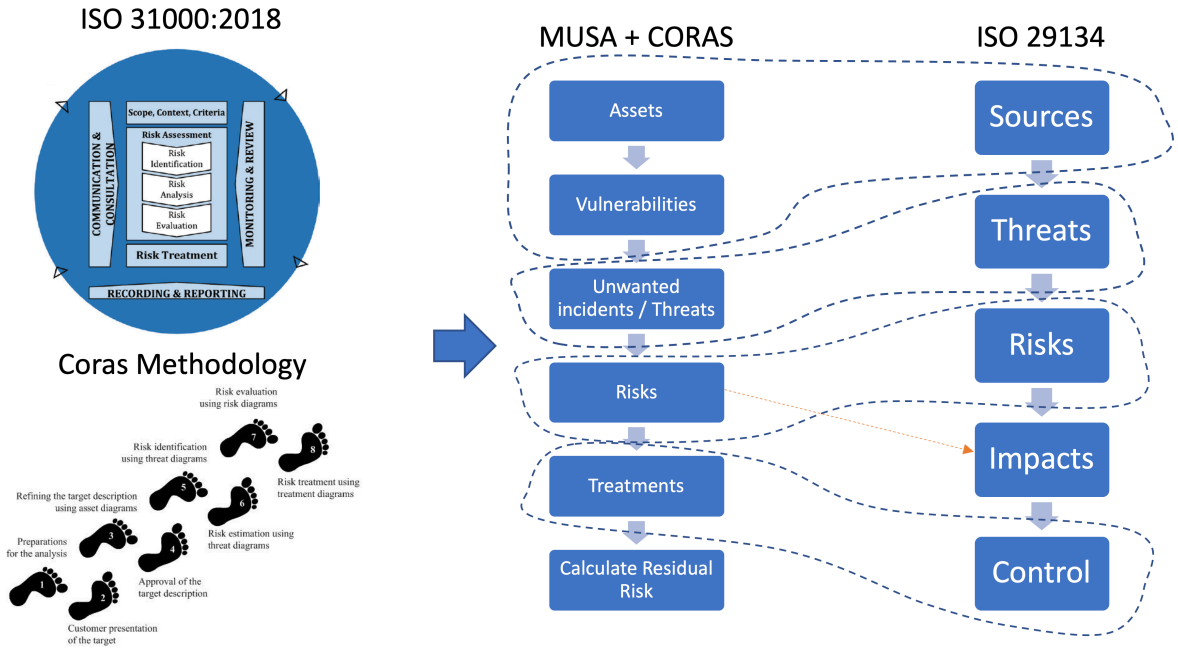


Figure 6. Outline of Risk Assessment module flow with MUSA framework (Design Time) (Extracted from MUSA D3.3)

Based on this analysis, in Figure 7, we propose a methodology for risk management in ENACT. In this figure we do not only depict the different steps of our methodology, but we also analyse the actors playing a central role in each of those steps in a DevOps-driven company.

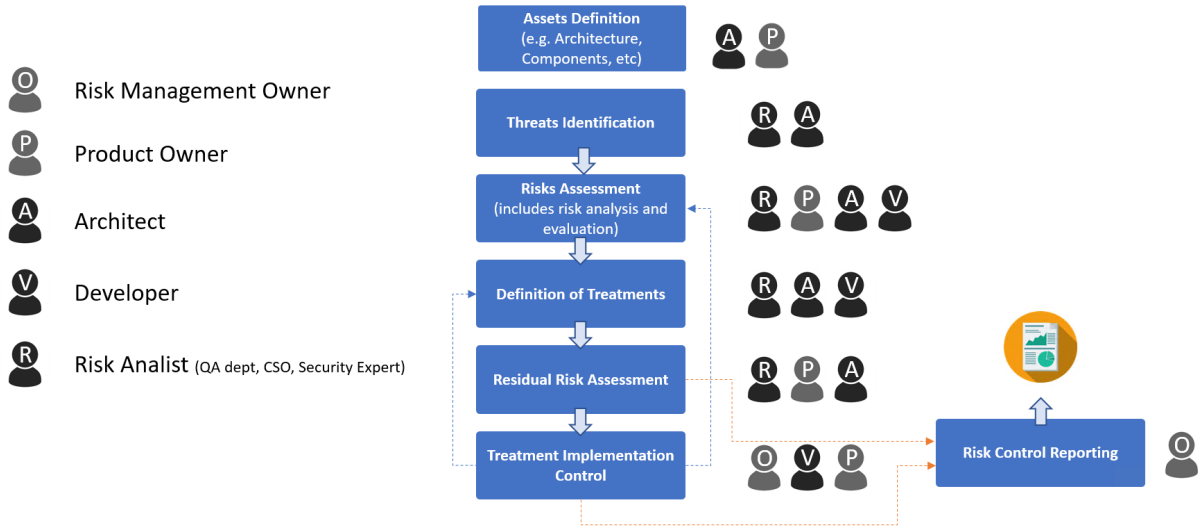


Figure 7. Risk Management methodology for ENACT's Risk Management enabler.

Our methodology, inspired by the previous analysis, can be summarize in 6 main steps. We add a seventh step to emphasize the need for reporting. The 6 steps are:

- **Assets definition**: most risk methodologies recognize the need to explicitly define our assets. This is usually an essential part of the methodology as the risks are analysed with respect the impact they may have on these assets. In ENACT, our assets will be the components of our IoT system. Instead of allowing the user to define them, which may be a hard task in a complex and distributed IoT system, we are planning to allow the tool to consume Genesis Models as described later on in Subsection 3.2.1. However, there may be other types of assets related to business aspects, such as reputation for instance. During the execution of ENACT project we will study how to deal with these aspects. One option would be to mimic what we already did in MUSA, using OWASP to define the likelihood and consequence. OWASP includes some of the most common business considerations when evaluating a particular risk. As mentioned before, we have not included a step to describe the vulnerabilities associated to a particular asset, as we consider this one an optional step in our methodology. However, ENACT's tool should be able to provide the means for an organization to define the vulnerabilities related to a component of an architecture or a subset of components. Please note that new risks may arise when combining different types of components in a system and this is hardly managed with existing technology.
- **Threats identification**: in this step, users are encouraged to identify threats that may affect the components in the described system. Detecting vulnerabilities in the previous step may be also helpful for threat identification. Previous definition of vulnerabilities may make some threats evident and it also allows for completeness checks at the end, by checking for vulnerabilities that have not been mapped to the current list of defined threats.
- **Risk Assessment**: risk assessment is composed of two different steps: risk analysis, where risks are evaluated in terms of likelihood and consequence, and risk evaluation, where risks are accepted, or they are classified as risks that need to be mitigated.
- **Definition of Treatments**: mitigation actions are defined in the form of treatments. A treatment can act as a mitigation action of different risks and a risk may require several

treatments. Deciding what is the minimum number of treatments required to mitigate a risk may not be straightforward and our tool will support it.

- **Residual Risk Assessment**: once the mitigation controls are defined, the residual risks need to be reassessed. This involves again two steps: risk analysis, where likelihood and consequence are updated after the application of the control(s), and risk re-evaluation, where risks are analysed again, and they are classified as accepted or further mitigation actions required.
- **Treatment Implementation Control**: finally, we add a last step in the methodology that goes beyond many of the methodologies defined before. In particular, it involves the control of the implementation of the mitigation actions (or controls) proposed in the previous step. This step will be connected to the enablers generated in WP4, to collect evidences from security and privacy monitoring and control in order to match them to treatments and risks.

We foresee at least the following roles involved in the usage of the ENACT Risk Management enabler:

- **Architect**: architect is a key actor in the execution of risk control processes. Architect will support definition of assets (and in particular, definition of the architecture), identification of threats, assessment of risks, definition of treatments and assessment of the residual risk after the application of these treatments.
- **Developer**: the developer will be involved in the correct implementation of treatments and their continuous control. They may also be involved in the risk assessment process and the definition of treatments. Their role in the risk control process may significantly change depending on the type of organization.
- **Risk Management Owner**: overall responsible for the risk management process. For instance, in the case we need to control risks related to privacy, the organization may need to appoint a DPO, other C-level executives may share their responsibilities (*e.g.*, the CEO). Within the risk control process in our tool the Risk Manager Owner will be responsible for controlling treatment implementation, reporting risk related issues (*e.g.*, to the company's board), or merging the results of risk analysis with the corresponding authorities, auditors or other members of the company board.
- **Product Owner**: the product owner, or project manager, usually conducts this process for a particular product or project of her responsibility. The product owner should be involved in several phases of the risk analysis process, including: the identification of sources (*e.g.*, hacker), the definition of assets, the assessment of risks (the first time and also the reassessment after the application of treatments), the control of treatment implementation and finally supporting the DPO in preparing documentation related to a DPIA, in case it is necessary.
- **Risk analyst**: the risk analyst role represents someone appointed to control the overall risk assessment process, bringing specific expertise in risk management. While the role may exist in any type of organization, small companies may not have staff with particular expertise in risk management and this role may be played by an architect or an external consultancy firm. In large enterprises, this role may represent a much wider

subset of roles including staff in the QA department, a CSO or a security expert in general. Risk analyst will be specially involved in identifying risk sources and threats, performing the risk assessment, defining treatments and calculating residual risks after their application.

2.3 Synthesis

In the following, we evaluate how our approach addresses the requirements defined in D1.1 (in particular, those associated with the Risk Management enabler in D2.1) and the current status with respect to those requirements. Please note that the text in the description of requirements in those previous deliverables has been refined after discussions with WP1 and WP2 partner to iterate over the original requirements to improve clarity. Updated texts have been underlined.

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.	This feature is partially implemented. Users are free to express any type of risks and there is a mechanism to connect related risks.
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.	An initial proof of concept has already been implemented, with a Kanban-like presentation to show the status of the risk analysis process and a mock-up of a dashboard to detect exceptions related to the risk analysis process.
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.	The first implemented version of the 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.	The initial pilot developed allows to specify the type of mitigation actions. Next step includes adding evaluation dimensions for each type of treatment (including for instance cost, implementation time, etc).
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.	Not implemented yet.
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.	Not implemented yet.
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	Not implemented yet.

		subset of treatments to consider a specific risk mitigated.	
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	Partially implemented in the current proof of concept through the treatment mitigation dashboard.
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.	Not implemented yet.
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.	Not implemented yet.

3 Continuous orchestration and deployment of SIS

This section presents the orchestration and deployment enabler and is an extended version of the following two ENACT research papers [2, 3].

Since SIS typically operate in a changing and often unpredictable environment, the ability of these systems to continuously evolve and adapt to their new environment is decisive to ensure and increase their trustworthiness, quality, and user experience. In particular, there is an urgent need for supporting the continuous orchestration and deployment of SIS over IoT, edge, and cloud infrastructures.

In the past years, multiple tools have emerged to support the building as well as the automated and continuous deployment of software systems with a specific focus on cloud infrastructures (*e.g.*, Puppet, Chef, Ansible, Vagrant, Brooklyn, CloudML, etc.). However, as identified in our literature reviews very little effort has been spent on providing solution tailored for the delivery and deployment of applications across the whole IoT, edge, and cloud space, especially at IoT devices level [4-6]. Cloud and edge solutions typically lack languages and abstractions 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 Internet [4-6]. In addition, they typically do not consider trustworthiness aspects such as security, privacy, resilience, reliability, and safety.

To address these challenges, we have developed a framework for the continuous deployment of SIS. In this section, we present our GeneSIS Framework and show how it facilitates the development and continuous deployment of SIS, allowing decentralized processing across heterogeneous IoT, edge, and cloud infrastructures. GeneSIS includes: *(i)* a domain-specific

modelling language to model the orchestration and deployment of SIS; and *(ii)* an execution engine that supports the orchestration of IoT, edge, and cloud applications as well as their automatic monitoring and deployment across IoT, edge, and cloud infrastructure resources.

The main contributions of GeneSIS that we present in this section are the following:

- It enables to cope with the heterogeneity across the IoT, edge, and cloud infrastructures and control the orchestration and continuous deployment of SIS that span across this space. A special focus has been to tackle challenges imposed by IoT infrastructures that typically include devices with no or limited access to Internet.

- The same language and tool are used for the continuous deployment of SIS (including the monitoring of the deployed system - i.e., monitoring if hosts are still reachable and if software component are still running, and the dynamic adaptation of a deployment - i.e., modifying how a SIS is deployed) providing a unique model-based representation of the SIS for both design- and run-time activities (*i.e.*, for developers and operators).

- It enables 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 mechanisms.

In the remainder of the section, subsection 3.1 describes the overall approach of the GeneSIS Framework. Subsection 3.2.1 presents the GeneSIS Modelling language while Subsection 3.2.2 details the supporting execution engine. Subsection 3.2.3 details how we extended ThingML for monitoring the execution flow of ThingML programs and how this is integrated in GeneSIS. Then, Subsection 3.3 shows the integration of the GeneSIS framework with existing IoT platforms, *e.g.*, FIWARE or SMOOL. Finally, Subsection 3.4 evaluates how our approach addresses the requirements defined in 3.1.1 and the current status with respect to the use case requirements as defined in D2.1.

3.1 Overall presentation of the enabler

In this section we first introduce a motivating example evolved from the smart building case study provided by Tecnalia to highlight in a single scenario the main requirements for GeneSIS. Then we present the overall GeneSIS approach.

3.1.1 *Motivating example*

The example is about a smart building that needs support for continuous deployment of its SIS together with security and privacy mechanisms. The smart building has several IoT gateways that control IoT field devices inside and outside the building. Figure 8 shows two representative IoT gateways: gateway1 (*RaspberryPi1*) that receives sensors' data (*i.e.*, temperature, fire, light) and camera data, and gateway2 (*Rpi2*) that sends commands to control the actuators, *e.g.*, heating, window blinds, LED-lights. For simplicity, we only discuss here a representative part of the smart building, which is about an IoT Energy Efficiency application that gets access to sensors' data to make decisions and send commands to control the actuators.

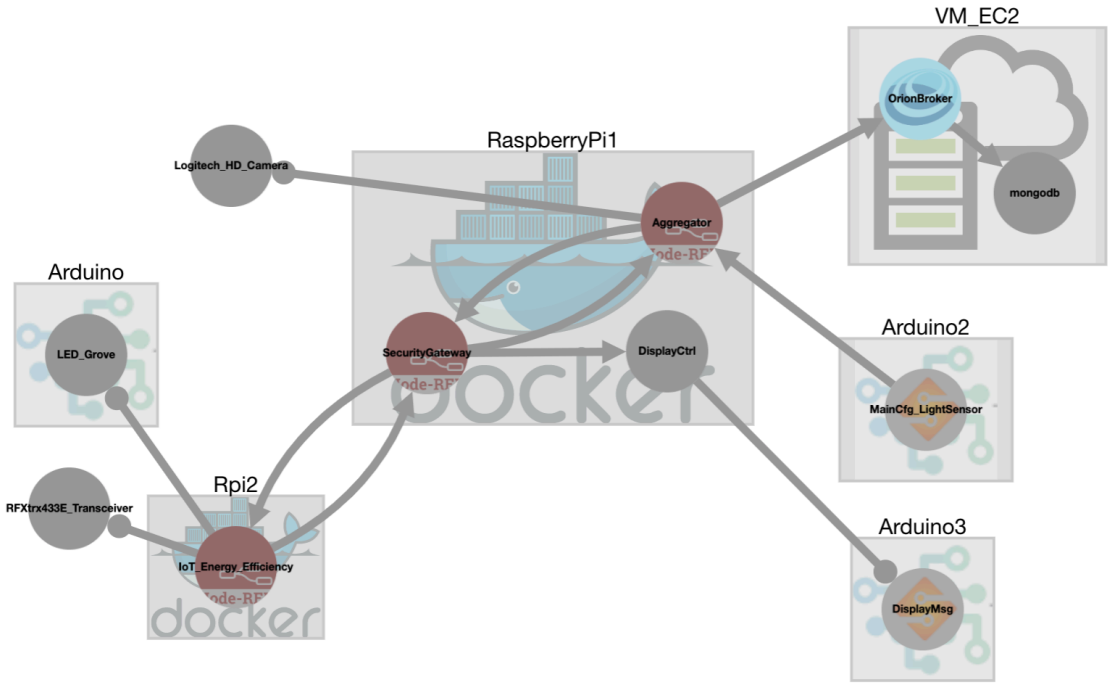


Figure 8. An example of a GeneSIS deployment model of a smart building application

Gateway1 hosts two main applications *Aggregator* and *DisplayCtrl* and a *security gateway*. The *Aggregator* reads data from the light sensor and also connects to the HD camera. Devices are either directly connected to Gateway1 such as the camera, or indirectly connected via serial communication from an Arduino controlling the light sensor. Gateway1 shares sensor data with the *IoT Energy Efficiency* application and stores it in the FIWARE Orion Broker hosted in the Cloud. The *IoT Energy Efficiency* application uses the sensors data to make decisions and sends commands to control the actuators. 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, it will turn on the LED-lights and/or open the blind if it becomes too dark. The application will also switch off the light if there is no person in the room after a certain time, *e.g.*, based on the video analysis application. On the other hand, if the room temperature is high, the IoT Energy Efficiency app may need to close the window blinds to prevent sunlight heating the room. Therefore, the actuator to control the blinds can be accessed concurrently (*e.g.*, the blind is used to control both the temperature and the light level) for different behaviours, which have conflicting goals and effects (*i.e.*, actions optimizing temperature may be hindered by actions optimizing light level). A device controller (*i.e.*, components whose role is to manage access to actuators) is added to the SIS to handle this problem.

The main requirement for GeneSIS deployment model regarding the support for specifying security and privacy is the following. There is a software component that requires to be deployed together with a specific security component providing a certain security and privacy capability. For example, the IoT Energy Efficiency application that reads sensor data must only do so according to an access control policy, or context-based access control mechanisms. It is important to note that the IoT applications can only receive sensors' data that they are allowed

to access and can only send commands to the actuators that they are allowed to control, in a dynamic context. For example, for security (and privacy) reasons, the IoT Energy Efficiency application can receive sensors data about temperature and light, but not live video from cameras. The IoT Energy Efficiency application can control heating system or window blinds, but NOT in case of fire alarm. For privacy reasons, camera data cannot be sent out of the Gateway1. The main requirement for GeneSIS is that when deploying the IoT applications, we also need to deploy security and privacy mechanisms together with the policies that must be enforced for those IoT applications.

More advanced requirements may be also needed such as the constraints of the deployed security and privacy controls. Constraints for deploying security modules must be considered by GeneSIS such as how far the host of a security module is from the sensors or actuators. For example, the security module may need to execute on a local node to the Gateway1 to ensure the performance of the authentication mechanism.

This example motivates for 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. This includes supporting the deployment of software components on devices with limited access to Internet. 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.

- **Security and privacy (R5)**: Supporting the specification of the security and privacy mechanisms that should be involved in the SIS (including their orchestration and how they can be deployed) is required.

- **Safety (R6):** The identification and management of actuation conflicts should be facilitated, in particular, to ensure authorisation policies in actuation.

Abstraction and infrastructure independence (R2) and automation (R4) are justified by the need for deploying the system on an infrastructure leveraging the IoT (*i.e.,* Arduino board), edge (*i.e.*, Raspberry PI), and cloud (*i.e.*, Amazon EC2) spaces. Separation of concerns (R1), automation and adaptation (R4), and actuation conflicts (R6) are justified by the need for dynamically adding a new software component to manage the access to the blinds 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 (*i.e.*, the RFXtrx433E transceiver) and white-box devices (*e.g.*, Raspberry PI). Finally, the support for security and privacy mechanisms (R5) is justified by the involvement in the system of actuators whose access should be controlled, and private data must be protected.

In the following sections, we present GeneSIS and how it addresses these requirements.

3.1.2 *Overall approach*

The objective of GeneSIS is to support the orchestration and deployment of IoT systems whose software components can be deployed over **IoT, edge, and cloud infrastructures**. The target user group for our framework is thus mainly software developers and architects. Figure 9 depicts the overall GeneSIS approach.

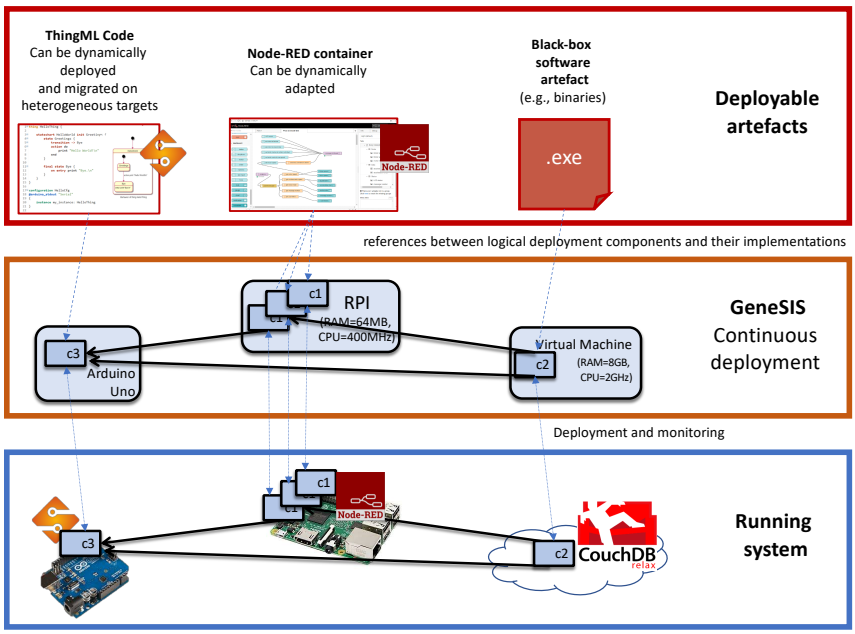


Figure 9. An overview of the GeneSIS approach

To deploy an application on the selected target environment, its components need to be allocated on host services and infrastructure. More precisely, what needs to be allocated are the implementations of those components. This is often referred as deployable artefact. 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) [6]. As depicted in the top layer of Figure 9, at the current moment, GeneSIS consumes as input three types of deployable artefacts:

- **Blackbox deployable artefact:** This refers to deployable artefacts that cannot be modified by GeneSIS (*e.g.*, GeneSIS can deploy a binary but cannot modify it). Our framework is agnostic to any development paradigm and technology, meaning that the developers can design and implement their blackbox deployable artefact based on their preferred paradigms and technologies. GeneSIS is also agnostic to any specific business domain.
- **ThingML source code:** ThingML [7, 8] is a domain specific language for modelling distributed IoT systems including the behaviour of the distributed components in a platform-specific or -independent way. From a ThingML code, the ThingML compiler can generate code in different languages, targeting around 10 different target platforms (ranging from tiny 8-bit microcontrollers to servers). This is particularly interesting for GeneSIS as, from a deployment model, the GeneSIS execution engine can identify the host to which a ThingML source code should be allocated and thus generate code in the relevant language before compiling and deploying it on the host. **This also provides GeneSIS with the ability to seamlessly migrate or deploy a ThingML program from one host to another.**
- **Node-RED container:** Using Node-RED, one can build an application as assembly of components executed in a Node-RED container, which can be dynamically adapted. **This provides GeneSIS with the ability to dynamically tailor an application to best fit its deployment.**

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 [9]. A deployment model is thus a connected graph that describes software components along with targets and relationships between them from a structural perspective [10]. A deployment configuration or deployment model typically includes the description of how its software components are integrated and communicate with each other. This is often referred to as software composition or orchestration. Software components represent either the deployable artefact and/or the resources on top of which of them are deployed.

GeneSIS includes: *(i)* a domain-specific modelling language to specify deployment model – *i.e.*, the orchestration and deployment of SIS across the IoT, edge, and cloud spaces; and *(ii)* an execution engine to enact the actual orchestration and deployment of SIS.

Because it is not always possible for the GeneSIS execution engine to directly deploy software on all hosts (*e.g.*, tiny devices do not always have direct access to Internet or even the necessary facilities for remote access), the actual action of deploying the software on the device has to be delegated to a host (*e.g.*, a gateway) locally connected to the device. The GeneSIS execution environment handles this problem by *(i)* **generating a deployment agent** responsible for deploying the software on the device with limited connectivity and *(ii)* **deploying** it on the host locally connected to the device with limited connectivity.

Finally, the GeneSIS execution environment is also responsible for monitoring the deployment and the status of the deployed SIS. It is worth noting that the information monitored from a

running system are fed back into its GeneSIS deployment model, providing a unique model-based representation for both design- and run-time activities.

3.2 Technical presentation and highlights

In this section we present the GeneSIS Modelling language before we detail its supporting execution engine. Compared to D2.1, the main evolution in the GeneSIS modelling language consisted in extending the language with new concepts to support the modelling and deployment of security and privacy mechanisms. In particular, and as detailed in Section 3.2.1, we added the concepts of *port* and *capabilities.* These provide the ability to specify, for each component, the capabilities (in term of security and privacy, execution support, hardware) it *offers* and *demands*. A deployment model is valid when all *demands* match *offers.* As detailed in Section 3.2.2, the main evolutions of the GeneSIS execution engine are: *(i)* support for the concept of ports and capabilities, *(ii)* extension of ThingML with monitoring and debugging mechanisms and its deep integration with GeneSIS, *(iii)* development of the GeneSIS deployment agent.

3.2.1 *The GeneSIS Modelling Language*

One of the objectives when we developed the GeneSIS modelling language was to keep it with minimal set of concepts, but still easily extensible. Our language is inspired by component-based approaches in order to facilitate separation of concerns and reusability (addressing R1). In this respect, deployment models can be regarded as assemblies of components. The type part of the GeneSIS modelling language metamodel is depicted in Figure 10.

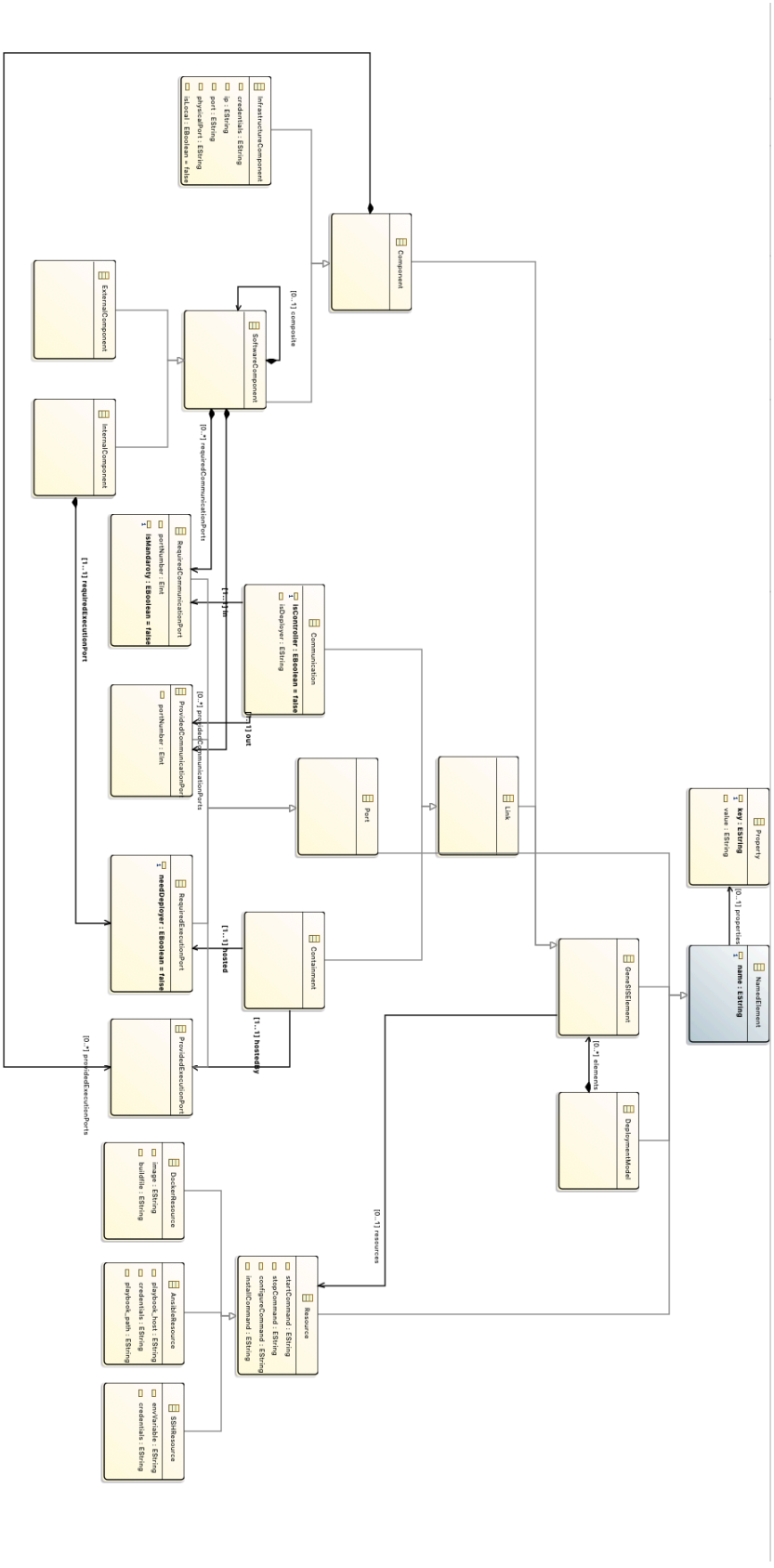


Figure 10. Metamodel of the GeneSIS modelling language

In the following, we provide a description of the most important classes and corresponding properties in the GeneSIS metamodel as well as sample models in the associated textual syntax. The textual syntax better illustrates the various concepts and properties that can be involved in a deployment model and that can be hidden in the graphical syntax.

A Deployment Model consists of *SISElements*. All *SISElements* have a *name* and a unique *id*entifier. In addition, they can all be associated with a list of *properties* in the form of key-value pairs. The two main types of *SISElement*s are *Component*s and *Link*s.

A *Component* represents a reusable type of node that will compose a *Deployment-Model*. A *Component* can be a *SoftwareComponent* representing a piece of software to be deployed on a host (*e.g.*, an Arduino sketch can be deployed on an Arduino board). A *SoftwareComponent* can be an *InternalComponent* meaning that it is managed by GeneSIS (*e.g.*, an instance of Node-RED to be deployed on a Raspberry Pi), or an *ExternalComponent* meaning that it is either managed by an external provider (*e.g.*, a database offered as a service) or hosted on a blackbox device (*e.g.*, RFXCom transceiver) (addressing R3). A *SoftwareComponent* can be associated with *Resource*s (*e.g.*, scripts, configuration files) adopted to manage its deployment life-cycle (*i.e.*, download, configure, install, start, and stop). In particular, there are three main predefined types of resources: *Docker-Resource* (see Listing 1), *SSH-Resource*s, and *AnsibleResources*.

Listing 1. An example of Internal component

```
{
      "_type": "/internal",
      "name": "Orion",
      "properties": [],
      "id": "bf1c8d43-b19e-49c7-969d-26b34e73e2e9",
      "provided_execution_port": [],
      "docker_resource": {
      "name": "f3e3feba-056e-46a7-9225-5b9edf5f1820",
      "image": "fiware/orion:2.2.0",
      "command": "-dbhost mongodb",
      "links": ["mongodb:mongodb"],
      "port_bindings": {
          1026": "1026"
      },
      "devices": {
                "PathOnHost": "",
                "PathInContainer": "",
                "CgroupPermissions": "rwm"
      },
      "required_execution_port": {
                "name": "needDocker",
                "needDeployer": false
      },
      "provided_communication_port": [{
                "name": "OrionNGSIv2 API",
                "port_number": "1026"
      }]
```

```
        "required_communication_port": [{
                "name": "requiresMongo",
                "port_number": "27017",
                "isMandatory": true
        }]
}
```

An *InfrastructureComponent* provides hosting facilities (*i.e.*, it provides an execution environment) to SoftwareComponents. The properties *IP* and *port* represent the IP address and port that can be used to reach the InfrastructureComponent (see Listing 2). The property *needDeployer* depicts that a local connection is required to deploy a SoftwareComponent on an *InfrastructureComponent* via a *Physical-Port* (*e.g.*, the Arduino board can only be accessed locally via serial port, see Listing 2). An *InfrastructureComponent* can expose a set of *hardwareCapabilities,* which represent the interfaces toward specific hardware facilities attached to the component (*e.g.*, a light sensor is plugged to the Arduino). This is important as *(i)* the software component that will use the hardware facility must know how to access it and *(ii)* in case a software component is using a specific interface for accessing a hardware facility, we must ensure that the required interfaces match what is offered by the *InfrastructureComponent.*

Listing 2. An example of Infrastructure component

```
{
        "_type": "/infra/device",
        "name": "ardui",
        "properties": [],
        "id": "dd3f5ac5-7723-449b-a57e-8c5d1d62252d",
        "provided_execution_port": [{
                "name": "arduino"
        }],
        "ip": "127.0.0.1",
        "port": [],
        physical_port": "/dev/ttyACM0",
        device_type": "arduino",
        "needDeployer": true
}
```

Components are connected through two kinds of ports. A communication port represents a communication interface of a component. A *ProvidedCommunicationPort* provides a feature to another component (*e.g.*, FIWARE Orion provides a REST interface, see Listing 3), while a *RequiredCommunicationPort* consumes a feature from another component (*e.g.*, FIWARE requires a MongoDB interface, see Listing 3). Only internal components can have a *RequiredCommunicationPort* since they are managed by GeneSIS. The property *isMandatory* of *RequiredCommunicationPort* represents that the *InternalComponent* depends on this feature (*e.g.*, the service hosted on RaspberryPi2 will not work if the communication with RFXtrx433E is not properly set up). The property *portNumber* represents the logical port that can be used to interact with the component.

An execution port represents the execution interface of a component (*i.e.*, the execution environment offered by the component for other components). A *ProvidedExecutionPort*

represents that the component provides execution environment facilities (*e.g.*, an Arduino board provides an execution environment for Arduino Sketches, see Listing 2), while a *RequiredExecutionPort* represents that the internal component requires an execution environment from another component (*e.g.*, FIWARE Orion requires hosting from a Docker engine, see Listing 3). A *RequiredExecutionPort* may require *securityCapabilities* (*i.e.*, a set of security mechanisms are required) (addressing R5), *executionCapabilities* (*i.e.*, a specific execution environment is required for the component to execute), and *hardwareCapabilities* (*i.e.*, some hardware facilities must be available at a certain location). By contrast, a *ProvidedExecutionPort* may offer *securityCapabilities* and *executionCapabilities.* For a deployment model to be valid, all the required capabilities must match a provided capability.

There are two main types of *Links*. 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 *Hosting* can be associated with *Resource*s specifying how to configure the components so that the contained component can be deployed on the container component. A *Communication* represents a communication binding between two *SoftwareComponents*. A *Communication* can be associated with *Resource*s specifying how to configure the components so that they can communicate with each other. The property *isController* indicates that the *SoftwareComponent* associated to the *in* attribute is controlled by the other (*e.g.*, all messages going to the Arduino should pass through the service hosted on RaspberryPi, see Listing 3) (addressing R6). Finally, the property *isDeployer* specifies that the *InternalComponent* hosted on the *InfrastructureComponent* with the *needDeployer* property should be deployed from the host of the other *SoftwareComponent* (*e.g.*, the artefact to be executed on the Arduino will be deployed from the RaspberryPi). This property is important as several host 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.

Listing 3. An example of Communication

```
{
      "name": " ctrlRFXtrx433E_Transceiver",
      "properties": [],
      "src": /IoT_Energy_Efficiency/64d0fff3-a8fc-408d-8370-9cbc28ce9d23",
      "target": "/RFXtrx433E_Transceiver/f7eb6490-b91a-475a-be9d-d6671d26f426",
      "isControl": true,
      "isDeployer": false,
      “isLocal”: false
}
```

It is worth noting that we applied the **type-instance** pattern [11] to *SoftwareComponents* thus facilitating the definition and reuse of generic types of components (addressing R1). As a result, components can remain device- and platform-independent or specialized into device- or platform-specific components (addressing R2).

3.2.2 *The GeneSIS execution engine*

From a deployment model specified using the GeneSIS Modelling language, the GeneSIS deployment execution engine is responsible for: *(i)* deploying the *SoftwareComponents*, *(ii)*

ensuring communication between them, *(iii)* provisioning cloud resources, and *(iv)* monitoring the status of the deployment.

3.2.2.1 Overall architecture

As depicted in Figure 11, the GeneSIS execution engine can be divided into two main elements: *(i)* the facade and *(ii)* the deployment engine.

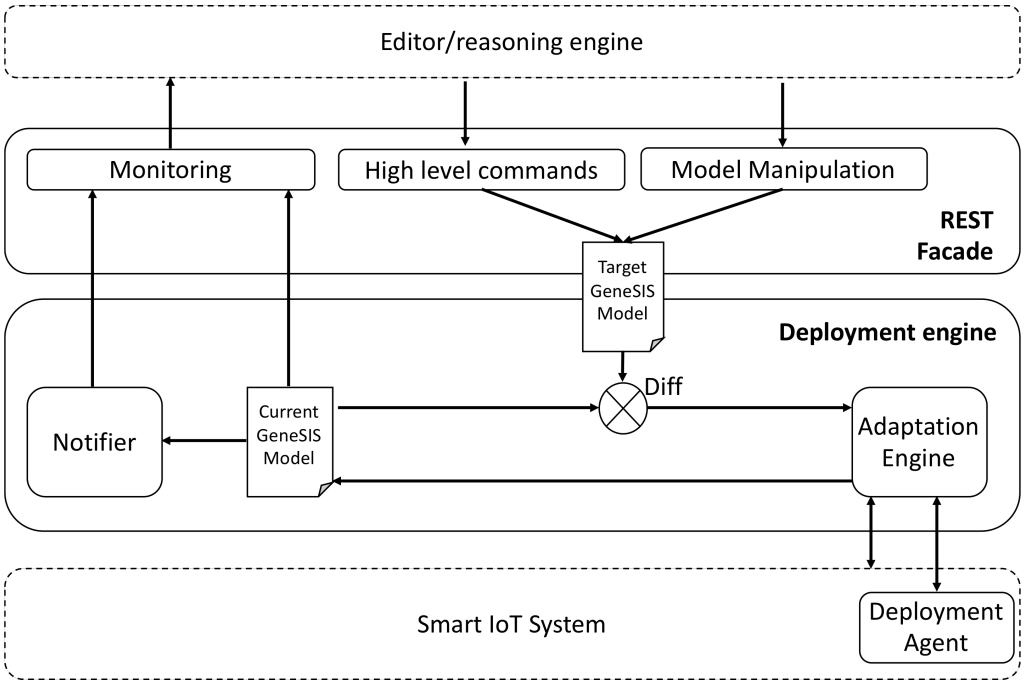


Figure 11. Models@Run-time architecture of GeneSIS

The facade provides a common way to programmatically interact with the GeneSIS execution engine via a set of three APIs. The monitoring API offers mechanisms for remote third parties (*e.g.*, reasoning engines) to observe the status of a system. Third parties can either consume the whole GeneSIS model of the running system enriched with runtime information or subscribe to a notification mechanism.
The high-level commands API exposes a pre-defined set of high-level commands that avoid direct manipulation of the models (*i.e.*, the model is automatically updated when the command is triggered).
At the current moment, this API only includes a migrate command that supports the migration of an *InternalComponent* from one host to another. Finally, the model manipulation API provides the ability to load a new target deployment model. In the future, it is also planned to provide support for the atomic MOF-level modifications of the deployment model.

GeneSIS follows a declarative deployment approach. From the specification of the desired system state, which captures the needed system topology, the deployment engine computes how to reach this state (Addressing R4). At the current moment, the deployment engine only computes a single adaptation plan, which may not always be optimal (i.e., not necessarily the fastest deployment of the one using lowest amount of bandwidth). In future work we will consider leveraging former work [12] to address this issue.

The GeneSIS deployment engine implements the Models@Run-time pattern to support the dynamic adaptation of a deployment with minimal impact on the running system. Models@Run-time [13] is an architectural pattern for dynamic adaptive systems that leverage models as executable artefacts that can be applied to support the execution of the system. Models@Run-time enables to provide abstract representations of the underlying running system, which facilitates reasoning, analysis, simulation, and adaptation. A change in the running system is automatically reflected in the model of the current system. Similarly, a modification to this model is enacted on the running system on demand. This causal connection enables the continuous evolution of the system with no strict boundaries between design- and run-time activities (addressing R2).

Our engine is a typical implementation of the Models@Run-time pattern. When a target model is fed to the deployment engine, it is compared (see Diff in Figure 11) with the GeneSIS model representing the running system. Finally, the adaptation engine enacts the adaptation (*i.e.*, the deployment) by modifying only the parts of the system necessary to account for the difference and the target GeneSIS model becomes the current GeneSIS model.

Finally, the deployment engine can delegate part of its activities to deployment agents running on the field (see Section 3.2.2.2 for more details).

A deployment process typically consists in the following steps.

1. **Check infrastructure:** This step consists in checking if the hosts specified in the deployment model are reachable (*e.g.*, is the docker remote API accessible at the address specified in the deployment model). For cloud resources, the objective is to check if the API offered by the cloud provider can be accessed.
2. **Provision and instantiate resource:** In the case of cloud solutions, this step consists in provisioning the cloud resources based on few constraints (*e.g.*, min CPU, min Disk, min RAM) and running the proper execution environment (*i.e.*, virtual machine image) as specified in the deployment model. For container technologies, this step consists in pulling the image of the container and running it with the set up specified in the deployment model (*e.g.*, access to file system, specifying open ports).
3. **Installation and configuration:** This step consists in running scripts and commands to configure and install software on the host. This includes ensuring that the software components that form the deployment topology can communicate with each other.
4. **Start:** this step consists in starting the deployed software.

In the following subsections, we detail the specific deployment support that is offered for two *InternalComponents* natively supported by GeneSIS: the Node-RED and ThingML components, namely. These are the deployable artefacts presented in Section 3.1.2, for which classical deployment approaches do not offer specific support. Yet, these nodes are represented as regular *InternalComponent* and GeneSIS is not bound to any of them (*i.e.*, GeneSIS can be used without these nodes).

Node-RED Components -- dynamic adaptation of the application behaviour

Node-RED [14], an open source project by IBM, uses a dataflow programming model for building IoT applications and services.

Provided with a visual tool, Node-RED facilitates the tasks of orchestrating IoT devices, wiring them up to form an IoT application. More precisely, a Node-RED application takes the form of one or more *flows*, which are composed of a set of *nodes* and *wires*. A *node* is a piece of software written in JavaScript that typically executes when a message is received from a *wire*. Node-RED can run at the edge of the network because of its light footprint. Thanks to the large community behind Node-RED, a large set of Node-RED nodes are available off-the-shelf making it easy to implement new applications.

The Node-RED Admin API can be used to remotely administer an instance of Node-RED[10]. In particular, it enables the dynamic loading of a flow or the dynamic modification of the running flow. Node-RED also implements the Models@Run-time pattern, it is thus possible to add or remove nodes without modifying the rest of the flow.

When deploying a Node-RED *InternalComponent*, GeneSIS leverages the Node-RED Admin API in order to dynamically instantiate the necessary nodes within a flow to ensure the communications with the rest of the components in the GeneSIS model. For instance, in the context of our motivating example, the service responsible for managing the accesses to the Arduino is implemented using Node-RED. Its deployment proceeds as follows. An instance of the Node-RED runtime is deployed and started using Docker. Once started, GeneSIS automatically instantiates and configures: *(i)* a "serial port communication" node to communicate with the Arduino board and *(ii)* "web socket out" (see Figure 8)

ThingML Components -- deployment across heterogeneous platforms

ThingML is an open source IoT framework that includes a language and a set of generators to support the modelling of system behaviours and their automatic derivation across heterogeneous and distributed devices at the IoT and edge end. The ThingML code generation framework has been used to generate code in different languages, targeting around 10 different target platforms (ranging from tiny 8 bit microcontrollers to servers) and 10 different communication protocols [15]. ThingML models can be platform specific, meaning that they can only be used to generate code for a specific platform (for instance to exploit some specificities of the platform); or they can be platform independent, meaning that they can be used to generate code in different languages.

The deployment of a ThingML *InternalComponent* by GeneSIS, not only consists in the deployment of the code generated by ThingML on a specific platform, but also in the actual generation of this code. The GeneSIS deployment engine proceeds as follows. It first identifies the platform on which the ThingML *InternalComponent* should be deployed. Then it consumes the ThingML models attached to the component and use ThingML to generate the code for the identified platform. If required, the generated code is further built and packaged before being deployed. Thanks to this mechanism, a ThingML *InternalComponent* can easily be migrated from one host to another. In other words, this means that the same ThingML code can be dynamically migrated from one device and platform to another without necessarily relying on a virtualization technology for lower footprint.

[10] https://nodered.org/docs/api/admin/methods/

3.2.2.2 The GeneSIS Deployment Agent

It is not always possible for the GeneSIS execution engine to directly deploy software on all hosts. Indeed, tiny devices, for instance, do not always have direct access to Internet or even the necessary facilities for remote access (in such case, the access to 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 must be delegated to the gateway locally connected to the device. Within our example, this is the case of the Arduino device whose software code can only be updated via the RaspberryPi gateway.

The GeneSIS deployment agent aims at addressing this issue (addressing R4). It is implemented as a Node-RED application. We decomposed the deployment procedure into four steps resulting in four groups of Node-RED nodes (see Figure 12). The flow of components (a.k.a. nodes) that will form the deployment agent is dynamically generated and deployed by the GeneSIS deployment engine based on the target host. It is worth noting that only the deployment node is mandatory and needs to be in the agent. Indeed, compilation and communications are activities that could be run anywhere on the IoT, edge, and cloud space. In addition, the other components can be distributed across different instances of Node-RED. We present below these four groups of components.

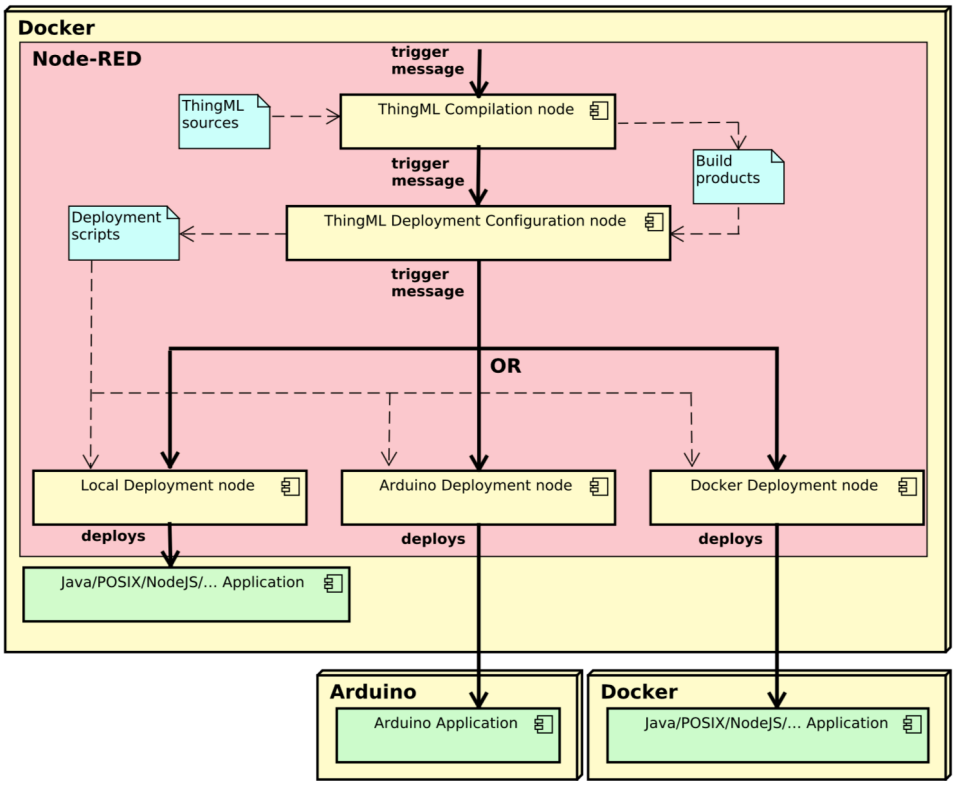


Figure 12. The deployment agent nodes

Code generation nodes: The aim of this type of node is to generate, from source code or specification languages, the code or artefact to be deployed on a target device. In the context of our motivating example, we created a ThingML compilation node, which consumes ThingML models and generates code in a specific language. The desired language is specified as a property of the node (*e.g.*, Arduino sketch in our example). The code generation is achieved by

using the ThingML compiler. In order to trigger a compilation, code generation nodes consume as input a *start compilation* message. Once the compilation is successfully completed, they should send a *generation success* message that includes the location of the generated code. Finally, a *compile on start* property can be set to true enabling to trigger the compilation when the node is instantiated. By contrast, the deletion of an instance of the node results in the deletion of the generated code.

Deployment configuration nodes: This type of node aims at preparing the actual deployment of a software component (being generated by 1. or not). This typically consists in generating configuration files. For instance, we created a ThingML Docker deployment configuration node that generates a "docker-compose" file as well as the relevant Dockerfile files depending on the target device. These nodes typically consume messages from the code generation nodes – *i.e.*, *generation success* messages that include details about the location of the artefact to be deployed. The retrieval of such a message triggers the actual generation of the configuration file. Once this process is completed, it generates a message containing the location of both the artefact to deploy and the configuration files. Removing an instance of configuration nodes results in the deletion of all the configuration files it has generated.

Deployment nodes: This type of node aims at enacting the deployment of a software component on a specific target. In the context of our motivating example, we created an Arduino deployment node that *(i)* build and upload an Arduino sketch on the Arduino board using the Arduino CLI[11] and *(ii)* install the libraries required for its proper execution. Similarly, we created a Docker deployment node. These nodes typically consume messages from the configuration nodes and do not produce any output. Removing an instance of a deployment node results in the termination of the deployed software (*e.g.*, killing a docker container, deploying a dummy Arduino sketch).

Communication nodes: After deployment, it can be important to communicate with the deployed software artefacts, for instance to monitor the status of a deployment. Communications nodes are regular Node-RED I/O nodes such as serial port for Arduino board or HTTP requests for REST services.

Thanks to this modularity, components from each of these groups can be seamlessly and dynamically composed for different types of deployments. The following scenario illustrates the benefits of such modularity. Considering our motivating example, in case of failure of the Arduino board, the *InternalComponent* named *DisplayMessage* could be temporarily migrated from the *Arduino* to the *RaspberryPi* by dynamically recompiling the ThingML code to Java and redeploying it on the *RaspberryPi*.

The development and operation of applications deployed by an agent and running on IoT devices such as Arduino boards is typically challenging as it is not always possible to access the logs or the systems output. To address this issue, we extended ThingML and the deployment of ThingML programs via GeneSIS with the necessary mechanisms to enable the remote debugging of ThingML programs as well as the run-time monitoring of its execution flow.

[11] https://playground.arduino.cc/Learning/CommandLine

3.2.3 *Model-based, Platform-independent Logging of the deployed SIS*

A typical application of Model-Driven Software Engineering [16] [17] (MDSE) is to provide abstraction on top of heterogeneous targets with the aim to facilitate the design, development, and operation of complex systems. With this objective in mind, a set of approaches propose to improve productivity by automatically generating code in different target languages from a common abstraction [18] [15]. However, one recurring challenge in those approaches is how to properly log, monitor and debug the generated programs (hereafter called target programs). Indeed, to fully benefit from the approach, such logging should be performed in a uniform way that relates to the concepts of the original abstraction level.

A number of mature tools for logging, monitoring and debugging already exist for most programming languages, and provided that fully operational code can be generated, those tools can be used as-is to generate insight about each individual target programs. However, in the cases where there is no direct 1-to-1 mapping between the original abstraction and each target language *i.e.*, if an original concept needs to be decomposed and recombined into lower-level constructs, those platform-specific tools will typically fail relating to the original concepts. As a result, the management of these tools may become overwhelming.

ThingML is a domain specific language for modelling the behaviour of distributed systems. From a ThingML specification, compilers can generate fully operational code for different languages, targeting around 10 different target platforms. ThingML is asynchronous by nature, and the main pattern to implement the behaviour of a program is composite statecharts *à la* UML. None of the target languages provide language-level support for statecharts, and while JavaScript and Go are asynchronous by nature, through the JavaScript event loop and Go routines, C and Java are mostly synchronous, with asynchronous mechanisms only available via libraries. This hinders the use of mature tools for logging ThingML generated programs, as discussed before. Instead, one way of logging the execution of programs generated from ThingML would be to extend the existing code generators, so as they automatically weave-in the extra instrumentation that is needed to bridge this abstraction gap. While this approach is fully possible, it comes at a hefty price:

- All the code generators need to be updated, which is a non-trivial task. Given that each generator is already a rather complex piece of software, great care, not to say extreme reluctance, is often the rule applied by developers and maintainers of code generators when it comes to extensions going beyond the sole objective of generating code for the existing concepts of the modelling language.
- This implies that a solid test suite needs to be implemented to ensure that logs are consistent across all the supported languages.
- To reduce the implementation effort, one can directly link to specific logging frameworks, so as to avoid maintaining a generic and extensible framework. While this might be an acceptable option for developers willing to use the selection solution, this forces the others to modify the compilers if they want to use another solution.

Rather, we propose a radically different approach, where existing compilers are left totally unchanged. This approach relies on *(i)* a platform-independent framework for logging, modelled existing ThingML concepts, *(ii)* a set of annotations to control which aspects of a

ThingML model should be logged, *(iii)* a set of endogenous ThingML-to-ThingML model transformations, automatically introducing the necessary instrumentation that integrates with the logging framework, and *(iv)* a set of interchangeable platform-specific drivers to transport the logs into different back-ends. Our empirical assessment on three different target languages indicated that this logging approach implies, in most cases, a reasonable overhead at runtime: +1-3% on execution time for Java and Go (however, +18% in JavaScript) and +7-9% on RAM.

Section 3.2.3.1 gives an overview of our platform-independent logging approach for heterogeneous target, while Section 3.2.3.2 details the model transformations involved in the automatic weaving of the logging instrumentation. Section 3.2.3.3.1 and Section 4.2.3.3.2 evaluate our approach, respectively from a quantitative and qualitative points of view.

3.2.3.1 Overall approach

Our main objective is to provide an automated, platform-independent and easy to use logging mechanism to 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.

ThingML [19][15] is a textual modelling language for heterogeneous and distributed reactive systems. It is fundamentally built around a sub-set of the UML: statecharts and components communicating through asynchronous message-passing. In addition, the ThingML language comes with a first-class, platform-independent action language, providing a classical set of actions and expressions found in most imperative languages *i.e.*, variables, functions and function calls, numerical and Boolean algebra, control structure (if, while, for-loop) and so on, as well as two more specific statements: *(i)* to send asynchronous messages (the reception of messages being handled in the statechart), and *(ii)* to mix ThingML statements with code from the target language (typically to implement drivers or wrap existing libraries). ThingML models can automatically be compiled to Java, Go and different dialects of C and JavaScript.

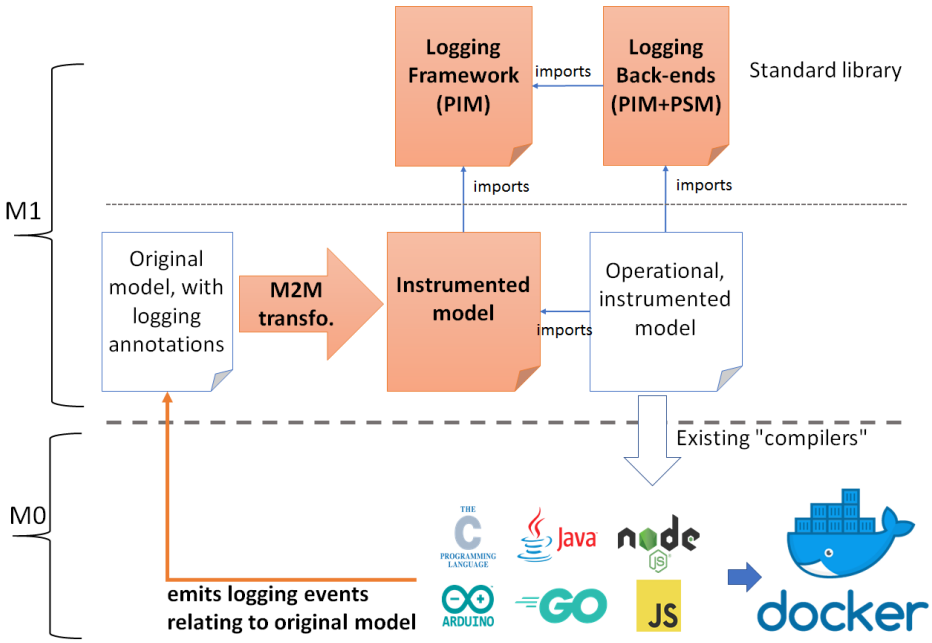


Figure 13. Overview of our model-driven, platform-independent logging approach.

Figure 13 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[12]. 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.

3.2.3.2 Logging as a set of model transformations

Our logging approach is built around:

- a generic logging framework, which provides an abstract, platform-independent logging API, described in sub-Section 4.2.3.2.1,

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

- a set of model transformations, responsible for weaving the required logging instrumentation and to interact with this framework, described in the remaining sub-sections.

The model transformations create, update and delete ThingML elements, which we will illustrate on simple examples, using a rather standard "diff" syntax: [+] for showing additions/updates and [-] for showing deletions.

ThingML Logging Framework

Our logging approach allows logging five key information of any ThingML program, as specified in the API shown in Figure 14:

- **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:
  - 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.

```
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 14. List of logging messages

```
component fragment Logger includes LogMsgs {

  provided port log {
    receives function_called, property_changed,
      message_lost, message_handled, message_sent
    receives log_on, log_off
  }

  abstract function log_function_called(
    inst : String, fn_name : String,
    ty : String, returns : String, params : String
  )
  ... //other log_* functions
  statechart init ON {

    state ON {
      internal event d : log?function_called
      action
        log_function_called(
          d.inst, d.fn_name,
          d.ty, d.returns, d.params
        )
      ... //other events reacting on log?* message
      transition -> OFF
      event log?log_off
    }

    state OFF {
      transition -> ON
      event log?log_on
    }

  }
}
```

Figure 15. Interface to be included by the logging back-ends

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, as shown in

Figure 15. In the *ON* state, the abstract logger will handle all the incoming logging events and delegate them to abstract functions, such as *log_function_called* having the same signature than the logging messages. Concrete logging back-ends will simply include this abstract component (called *fragment* in ThingML) and implement the abstract functions. In the *OFF* state, all incoming logging events are simply discarded.

A component that needs logging will include the *WithLog* fragment defined in Figure 16. This component will then be able to send the logging messages through a *log* port. In addition, the component to be logged needs to define a *DEBUG_ID* property to identify the component. This identifier will be passed as the first argument of the logging messages *i.e.*, the *inst* parameters in Figure 15.

```
component fragment WithLog includes LogMsgs {

  readonly property DEBUG_ID : String

  required port log {
    sends function_called, property_changed,
      message_lost, message_handled, message_sent
  }

}
```

Figure 16. Interface to be included by the components to be logged

The *WithLog* fragment can be imported and used manually by developers. This usage, though fully possible and legitimate, will not be addressed in this document. Rather, our approach promotes the use of the following annotations:

- *@monitor*, with possible values *functions*, *properties* or *events*. This annotation only applies to component types.
- *@monitor "not"*, which can be applied to specific functions, properties or messages.

These annotations allow developer to specify what they want to log, and filter out specific elements that s/he do not want to include. These annotations will be used to select which of the model transformations, described in the remainder of this section, should be applied and where they should be applied. The goal of those transformation is to weave instrumentation code, with no side-effect, that is as close as possible to what the developer would write if s/he used the logging framework manually.

Logging ThingML properties

Logging updates on properties is rather trivial, but can have side effects if not implemented correctly, especially if the property is assigned with the result of an expression involving a function call (*a = c()*), as shown in Figure 17.

```
    property a : Int32
    ...
    //each property assignment is logged like that:
[+] readonly var old_a : String = a as String
    a = c()
[+] readonly var new_a : String = a as String
[+] log!property_changed(DEBUG_ID,
[+]                      "a", "Int16",
[+]                      old_a, new_a)
```

Figure 17. Effects of instrumenting property assignments

The variable assignment *a = c()* is now surrounded by two local variables, which store the values of the variable before and after the update, without calling *c()* multiple times, and serialize them into strings. Finally, the logging message is sent, containing information of the updated variable, including the old and new values.

Logging ThingML functions

Instead of instrumenting function calls, we rather instrument the bodies of functions. This strategy has several advantages:

- less instrumentation code needs to be woven into the ThingML model, assuming the given function is called more than once.
- function calls can either be an action *i.e.*, a standalone one-line statement, or an expression (as in the variable assignment of Figure 17) potentially included in an arbitrarily complex action. In the latter case, this implies extracting the function calls out of the action where they are contained, store their return values into local variables, update back the action by replacing function calls by variable references, etc. Though this can be implemented, it would make the transformation more complex.

Figure 18 shows the result of this transformation. First, the actual parameters of the function are stored and serialized into a string. Then, if the return type of the function is not void, every *return* statement is instrumented in a quite similarly to how property assignments are instrumented (see previous sub-section). Ultimately, a logging event is emitted, describing the function, its parameters and return value.

```
//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 18. Effects of instrumenting functions

The instrumented function has the very same signature than the original function, and no further update is needed.

Logging ThingML events

Logging messages sent by a component requires to extract the parameters of the message into separate variables, to avoid side effects. Those variables are used in the updated send action and in the logging message, as show in Figure 19.

```
//consider a state CURRENT
state CURRENT {
  //consider an event changing the state to NEXT
  //this will be logged like this:
  transition -> NEXT
  event e : p?m
  guard ... //unchanged
  action do
[+]  readonly var params : String = e.a as String
[+]                      + "," + e.b as String
[+]  log!message_handled(
[+]    DEBUG_ID,
[+]    "CURRENT", "NEXT",
[+]    "p", "m", params
[+]  )
    ... //unchanged former behavior
  end
}
```

Figure 19. Effects of instrumenting the emission of messages

An event is consumed if a handler (transition or internal transition) reacting on that event is successfully triggered. Logging consumed messages thus requires instrumenting the action of existing handlers, as shown in Figure 20. For a given handler, the instrumentation records the parameters of the message, if any, and sends a logging event. If the handler already defined an action, the content of this action is appended after the logging instrumentation.

```
//consider a state CURRENT
state CURRENT {
  //consider an event changing the state to NEXT
  //this will be logged like this:
  transition -> NEXT
  event e : p?m
  guard ... //unchanged
  action do
[+]  readonly var params : String = e.a as String
[+]                      + "," + e.b as String
[+]  log!message_handled(
[+]    DEBUG_ID,
[+]    "CURRENT", "NEXT",
[+]    "p", "m", params
[+]  )
    ... //unchanged former behavior
  end
}
```

Figure 20. Effects of instrumenting messages that are handled

According to the usual semantics of composite statecharts, events are consumed in-depth first. If the current state cannot consume an incoming event, then the event is delegated to its parent state, which might consume it or not. If not, the process repeats, potentially up to the top-level statechart. If the top level statecharts does not consume the event, it is discarded. For a given event, we distinguish three cases:

- the original top-level statechart already reacted to this event through a handler with no guard: the event will always be handled, and there is no need for instrumentation as the event will never be discarded.
- the original top-level statechart defines no handler for this event *i.e.*, if the event is not handled by a nested state, it will be discarded. In this case, the instrumentation will consist in an unguarded handler, which reacts on this event and emits a *message_lost* logging event.
- the original top-level statechart defined one guarded handler, or more, for this event. In this case, the instrumentation should only catch the event if the existing guard(s) evaluate(s) to false, as shown in Figure 21.

In Figure 21, we consider that the original statechart was reacting on event *p?m* - *i.e.*, a message *m* received on a port *p*. This message contains an integer parameter *a*. More precisely, the statechart reacts to this event through two guarded internal transitions, which will catch the event iff *a>100* or *a<100*. An experienced eye would notice that this event will never be caught if *a == 100*, which is exactly what we want to log by creating a similar internal transition, also reacting on *p?m*, which will trigger iff the other two internal transitions do not. This is achieved by guarding this new transition with the Boolean conjunction (*and*) of the negation (*not*) of the existing guards on that event *p?m*.

```
statechart init INIT {
    //consider the original statechart reacts
    //on e : p?m [e.a > 100]
    internal event e : p?m
    guard e.a > 100
    action ...

    //and reacts on e : p?m [e.a < 100]
    internal event e : p?m
    guard e.a < 100
    action ...

[+] internal event e : p?m
[+] guard not (e.a > 100) and not (e.a < 100)
[+] action do
[+]   readonly var params : String =
[+]                 "a=" + (e.a as String)
[+]   log!message_lost(DEBUG_ID, "p", "m", params)
[+] end

  state INIT { ... }
}
```

Figure 21. Effects of instrumenting messages that are discarded

3.2.3.3 Evaluation

Our logging approach is implemented as an extension to the ThingML framework and is available under the Apache 2.0 Open-source License. It is implemented in about 1000 lines of Java code organized as follows:

- MonitorAspect: An interface defining a *monitor* method, which abstracts the currently available monitoring/logging aspects:
    - **PropertyMonitoring**: implementing the model transformation presented in Section 4.2.3.2.1.
    - **FunctionMonitoring**: implementing the model transformation presented in Section 4.2.3.2.2.
    - **PropertyMonitoring**: implementing the model transformations presented in Section 4.2.3.2.1.
- **MonitorGenerator**: this class browses the ThingML model and based on the annotations available in the model will instantiate and execute the different monitoring aspects.

The scripts used in the quantitative evaluation (Section 4.2.3.3.1), as well as all the input data (ThingML models) and the data generated by those scripts (instrumented ThingML models, generated code for Java, JavaScript and Go, and logs resulting from the execution of this code) are available under the Apache 2.0 Open-source License. The dashboard we developed as part

of the qualitative evaluation (Section 4.2.3.3.2) is also available under the Apache 2.0 Open-source License.

Quantitative impact of logging instrumentation

In this experiment, we measure the runtime overhead of our approach, by measuring the RAM consumption and the execution of a medium-sized ThingML program. Three versions of this program are benchmarked:

- **no**: Original program without logging instrumentation
- **off**: Instrumented program, but with the actual logging turned off. This means that the program will emit log events, which will be received and simply discarded by the logging back-end.
- **on**: Instrumented program, with logging turned on. For this experiment the logging back-end will simply print to the standard output, which is re-directed to a file.

The results of this section have been produced by executing the four scripts contained in *ThingML-logs-xp/src/main/bash*:

- *00_build_ThingML.sh*: builds a special version of ThingML, also including our logging tool, and exposes it as a Docker container.
- *01_instrument_models.sh*: weaves the logging instrumentation (for all properties, functions and events) into the base ThingML model.
- *02_generate_code.sh*: generates code (using 1) for Java, NodeJS and Go, from the base ThingML model (**no**) and the instrumented ThingML model, with logging **off** and logging **on**.
- *03_run_in_docker.sh*: runs the produced Java, NodeJs and Go code into separate containers, 100 times in each mode (**no**, **off** and **on**) and collects logs.

The input ThingML model already contains instrumentation to monitor the memory and total execution time of the program, which is used to produce metrics for all 900 executions of the target programs: three modes (**no**, **off** and **on**) and three languages (Java, NodeJS, Go), executed 100 times each.

Impact on memory

Figure 22 shows the memory consumption for the three different languages, in the three different modes. Rather unsurprisingly, we observe an increased memory consumption: having the log instrumentation (depicted as **off** in the figure) consumes more memory than not having it (depicted as **no** in the figure) and turning on the actual logging (depicted as **on** in the figure) again increases the memory consumption. The memory consumption is about +7% in **on** mode compared to **no** mode for Java, about +8% for NodeJS, and +9% for Go. The memory consumption increases rather linearly along the three modes in NodeJS and Go. We observe a different pattern in Java, where most of the overhead is already present in **off** mode.

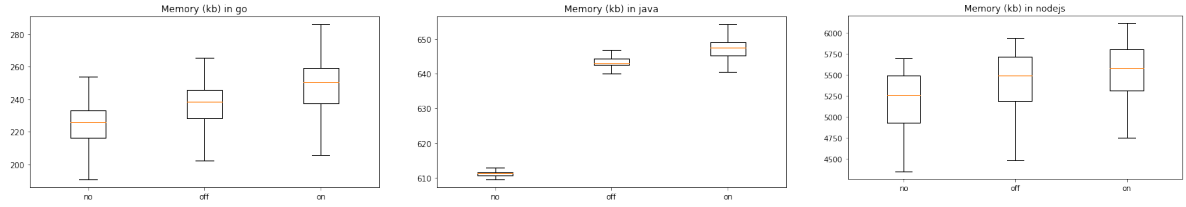


Figure 22. Use memory in Java, JavaScript and Go, without log instrumentation (no), with log instrumentation inactive (off) and active (on)

Impact on execution time

Figure 23 shows the execution times for the three different languages, in the three different modes. Rather unsurprisingly, the original programs (no instrumentation) yield the fastest execution times. We observe two different patterns:

- In NodeJS, the overhead of the logging instrumentation, when the logging back-end simply discard all logging events (**off** mode) is negligible. However, the overhead when the logging back-end is active is (**on** mode) is rather large: about +18%.
- In Go and Java, the overhead in **off** mode (+2% in Go and +3% in Java) is larger than the overhead in **on** mode (+1% in Go and +1% in Java). In any case, this overhead is rather well contained.

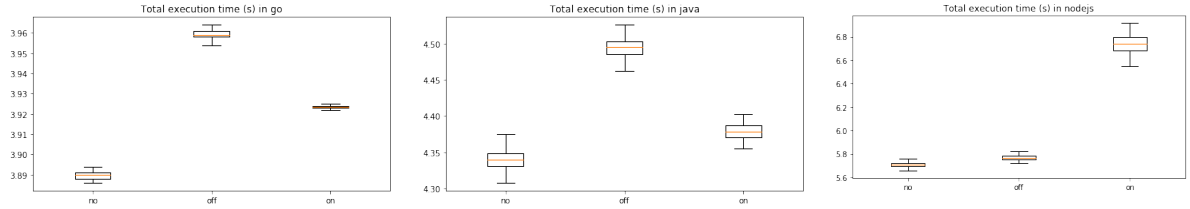


Figure 23. Execution times in Java, JavaScript and Go, without log instrumentation (no), with log instrumentation inactive (off) and active (on)

Explaining the large overhead on the execution time if NodeJS in **on** mode requires more investigations, in particular to determine if this overhead is the fact of:

- the NodeJS interpreter, which we deem rather unlikely,
- the chosen logging back-end (*i.e.*, prints to the standard output redirected to a file), which we deem rather likely,
- the logging instrumentation, which we deem very unlikely given the negligible overhead in **off** mode, where logging events are emitted by our test program and received by the logging back-end, and then simply discarded, or
- the experimental setup itself (*e.g.*, the fact that we execute NodeJS within Docker containers), which we deem very unlikely, given the other two languages are not affected by this issue, and that Docker should in principle be oblivious from what actually runs within containers.

A more important issue is that the C code generated from the instrumented ThingML model cannot be compiled as-is by GCC or other C compilers. We plan to overcome this issue for deliverable D2.3.

Qualitative evaluation

We created the web-based dashboard shown in in Figure 25. This dashboard receives a stream of raw log events and provides developers with details about the status of their target program, in terms of ThingML concepts.

The communication between the ThingML logging framework and the dashboard is made using MQTT. Rather than directly printing logs to the standard output, as we did in Section 3.2.3.3, we now use a MQTT logging back-end, partly generated by the ThingML framework [15]. The log messages are received by this MQTT logging back-end, automatically serialized into JSON (as shown in Figure 24 and published on a MQTT topic.


```
{
   "inst":"game",
   "prop_name":"bx",
   "ty":"Int16",
   "old_value":"8695",
   "new_value":"8508"
}
```


Figure 24. Example of a log message

MQTT has been selected because as it follows the Publish-Subscribe pattern as it is scalable and offers great space, time, and synchronization decoupling [20]. Scalability is important as events can be published at a high frequency (*e.g.*, up to 6 messages per milliseconds in our simple example). The raw log events are published on different topics based on the categories defined in Section 3.2.3.2.

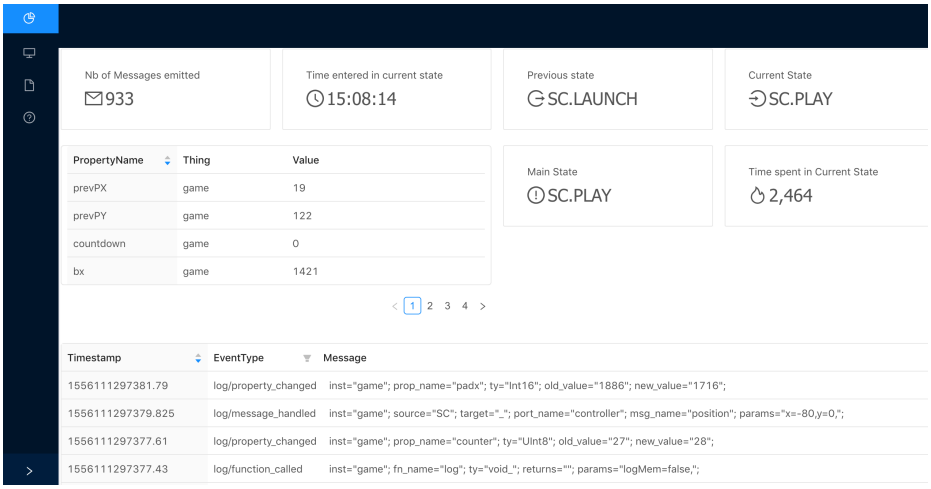


Figure 25. Screenshot of the ThingML logging dashboard

Using those raw log events, we were easily able to provide developers with the following information:

- The current value for the monitored properties of the ThingML model (see the top table in Figure 25).
- The current state (in term of the ThingML state machine) in which the program is.
- The previous state, in which the program was just before the current one.
- The time spent in the current state in milliseconds.
- The time when the program entered in the current state.
- The state in which the program has spent most time.

- The number of messages emitted by the program since it started.

In addition, all the log events are displayed in a table and can be sorted and filtered (see bottom table in Figure 25). Our systematic and automatic logging approach, combined with this dashboard can be seen as a sort of models@runtime approach [21], where the execution of programs generated from ThingML can be traced so as to dynamically re-construct the original ThingML model.

In the following section we discuss the relationship between GeneSIS and the major IoT platforms and commercial solutions.

3.3 Integration with existing platforms

Real IoT systems are seldom developed from scratch but rather build upon off-the-shelf components, legacy sub-systems. In addition, they can rely on the wide range of IoT platforms that have been developed over the past decade. We investigated the integration of GeneSIS and ThingML with the main IoT platforms and commercial solutions and in particular, the integration with FIWARE, SOFIA (the SMOOL variant), and Microsoft IoT Hub.

Integration with FIWARE

FIWARE is defined as a *"framework of open source platform components which can be assembled together and with other third-party platform components to accelerate the development of Smart Solutions."*[13]. We decided to integrate GeneSIS with the FIWARE Orion Context Broker as it is the central element of the FIWARE platform. Orion is the only mandatory component of any "Powered by FIWARE" platform. It allows managing the entire lifecycle of context information, including the possibility to create context elements and to manage them through updates and queries. It is important to note that integrating the Orion Context Broker in a SIS seamlessly provides it with access to the whole FIWARE ecosystem and in particular to the Generic Enablers.

GeneSIS supports the deployment and orchestration of SIS integrated with the Orion Context Broker in two ways. First, GeneSIS can manage the deployment, configuration, and adaptation of Orion as any other software component. As depicted in Figure 26, a specific component type has been created and is available by default when starting GeneSIS. More precisely, this GeneSIS component is a subtype of *Internal Software Component.* It is worth noting that this component exposes a mandatory *Required Communication Port* (meaning that Orion will not work properly if the dependency is not accessible) as a MongoDB datastore is required for Orion to execute properly. The example of deployment model involving Orion depicted in Figure 26 is available at the following address: https://gitlab.com/enact/GeneSIS/tree/master/docs/examples.

[13] https://www.fiware.org/developers/catalogue/

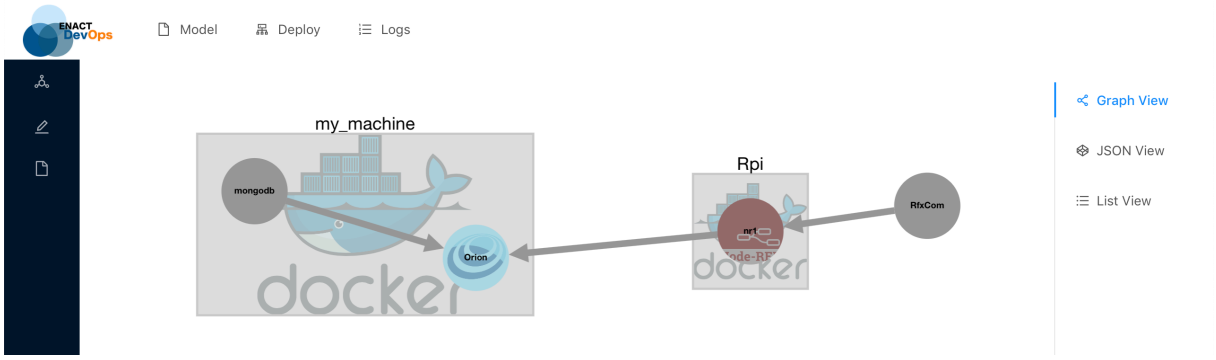


Figure 26. Deployment model involving the FIWARE Orion Context Broker

Second, in order to facilitate the interactions between the software components of a SIS and Orion, we created a simple REST-based proxy that can be placed in front of an instance of Orion. It is implemented using Node-RED (see Figure 27) and can thus be easily dynamically adapted.

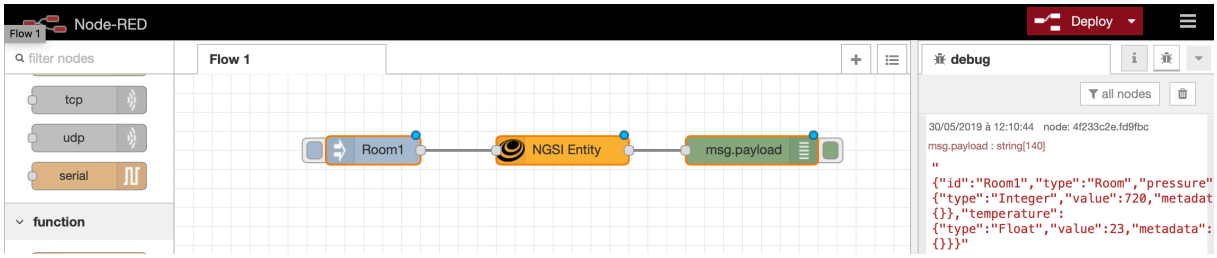


Figure 27. Querying the Orion Context Broker using Node-RED

Integration with SMOOL

SMOOL [22] is an IoT middleware that aim at providing a publish/subscribe communication infrastructure to facilitate the creation of IoT applications. SMOOL is developed by TECNALIA and leverages the Smart Space Access Protocol (SSAP) developed in SOFIA. SMOOL applications consist of a set of Knowledge Processors (KPs) that exchange data via a Semantic Information Broker (SIB). SMOOL comes with a KP generation wizards. The wizard can be used to generate a 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 the smart application [22].

A SMOOL KP can be deployed by GeneSIS as any other software component. In addition, 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. Details about this integration can be seen in the following video: https://www.youtube.com/watch?v=mfT_AwfkXNc

Integration with Microsoft IoT Hub

Microsoft IoT Hub[14] is a platform to connect, monitor, and manage Edge and IoT devices. More precisely, it provides mechanisms to enable communications between IoT, Edge, and Cloud

[14] https://azure.microsoft.com/en-us/services/iot-hub/

devices together with mechanisms to support the reliable deployment of software components over large set of edge devices. When combined with the Azure IoT DevKit, it is also possible to deploy and update firmware on IoT devices with limited resources. However, to do so, it is required to run agents on the edge devices (in the form of docker containers) or specific software component for smaller IoT devices (for instance see: https://github.com/Azure/azure-iot-arduino) thus creating what can be considered as a vendor lock-in.

The integration of GeneSIS with the Microsoft IoT Hub is currently under development. We foresee two types of integration:

1. Using GeneSIS to trigger new IoT Hub deployment. This would also allow breaking the vendor lock-in by enabling the deployment and management of software components via the Microsoft IoT Hub together with other technologies.
2. Deploying GeneSIS on edge devices using Microsoft IoT Hub and using it to manage small devices. This will help achieve the remote deployment and management of the IoT systems without a direct IP-based connection to the main edge device, which in turn improves the scale of IoT systems that GeneSIS can achieve.

3.4 Synthesis

In the following, we evaluate how our approach addresses the requirements defined in (i) Section 3.1.1 and (ii) the current status with respect to the use case requirements as defined in D1.1.

- **Abstraction and Infrastructure independence (R2):** By leveraging model-driven engineering techniques, 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):** 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. The GeneSIS deployment agent enables the deployment of software component on devices with limited access to Internet. By applying the Models@Run-time pattern, and thanks to its Facade, GeneSIS provides developers and autonomic managers with the means to enact adaptations of the deployment of SIS.
- **Security and privacy (R5):** Regarding security and privacy mechanisms, using the GeneSIS it is possible to specify the security and privacy capabilities provided and required by a component.

- **Safety (R6):** Regarding actuation conflicts, the GeneSIS modelling language facilitate the identification of concurrent accesses to actuators and allow electing a software component as the controller of an actuator, meaning that all accesses to the actuator should be done through that component.

Table 2. Requirements from D2.1

ReqID	Requirement	Description	Status at M15
UC1-3 **R1**	Scalability	GeneSIS should be able to deploy SIS involving hundred sensors/actuators.	**Not yet available.** Scalability has not been a main concern so far but will be for D2.3. Largest deployment involved 15 nodes. Integration with Microsoft IoT Hub will help fulfilling this requirement.
UC1-3 **R2**	Scalability	The GeneSIS modelling language should be able to represent deployments involving hundred sensors/actuators.	**Not yet available.** Scalability has not been a main concern so far but will be for D2.3. Largest deployment involved 15 nodes.
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.	**Partially covered.** A first 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 provide 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.	**Partially covered.** 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.
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.	**Partially covered.** Integration with the CloudML provisioning engine is ongoing.
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	**Partially covered.** GeneSIS implements the models@runtime pattern, which provides a mean to enhance deployment models with runtime informations. At the

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

Table 3. 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	**Ongoing.** GeneSIS has not yet been applied to the on-board systems. However, we demonstrated the deployment on at least 2 gateways, 2 IoT devices, 1 cloud resource.
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	**Ongoing.** GeneSIS has not yet been applied to the on board systems. However, we demonstrated the deployment on at least 2 gateways, 2 IoT devices, 1 cloud resource.
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"	**Partially covered.** Deployment on IoT, Edge and cloud resources has been demonstrated. We are currently working on the support for multiple deployment.
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)	**Ongoing.** Update of software component has been demonstrated using Ansible. Work is currently done with WP4 and the diversifier at runtime to manage multiple upgrades in a more generic way.
TO1.3	eHealth	**Automatic Pairing of devices with Gateway after reset.**	**Not started.**

		At least 5 different devices automatically paired with Gateway after reset	
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 in lab context.** Integration has been achieved between ThingML and SMOOL. This has now to be applied in the context of the use case in collaboration with WP1.
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.	**Ongoing.** GeneSIS has been extended to support the specification security and privacy concepts (i.e., specifying which S&P mechanism should be used). Validation is ongoing. Deployment of actuation conflict managers is already supported.

4 Identifying, analysing and managing actuation conflicts

4.1 Overall presentation of the enabler

4.1.1 *Illustration and Motivation*

Since SIS are not limited to collect sensors data but also act on the physical environment, new software challenges appear for supporting their development and the operation. One of them is the *management of actuation conflicts* when different applications interact concurrently on shared devices or within a shared physical environment (e.g. Figure 28). Then, tools must handle these interactions so as to prevent non-anticipated evolutions of the physical environment.

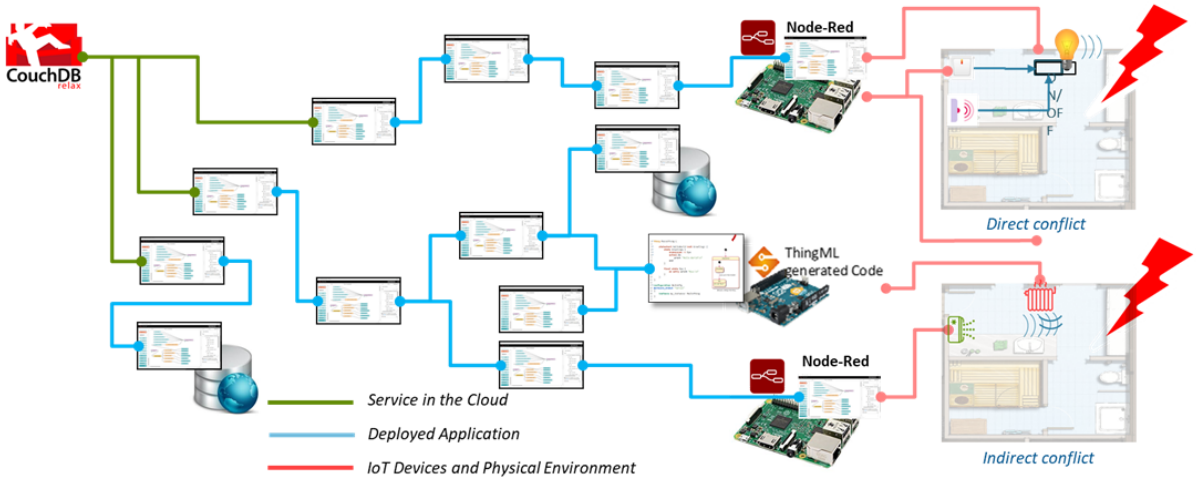


Figure 28: SIS and Actuation Conflits

A *direct conflict* occurs when simultaneous antagonist accesses are triggered to a shared software resource. In DevOps for IoT, these resources are mostly located in front of the actuation systems and a conflict occurs when at least two applications, a priori independent, try to access a shared actuation system.

For instance, imagine a first application switching the light on when it detects a presence in the room. Then we consider a second application switching the light off when the TV is in use. This situation may result in a blinking light, but this is even not sure!

Sometimes actuation conflicts are more insidious, and we talk about *indirect actuation conflicts*. They appear when two applications act on separate actuation systems whose effects interfere with each other through the physical environment. Effects resulting from these actions may become non-anticipable. In such a case, conflicts are not detectable from the software application models.

For instance, in the TECNALIA KUBIK smart building, blowers manage heating, ventilation and cooling. If one considers two blowers acting in the same room, both can blow hot air, cold air, or can be stopped. Since both devices are independent and do not communicate with each other, it is possible that one blower is blowing hot air while the other one is blowing cold air. The resulting behaviour is obviously counterproductive and needs to be avoided. Managing such situation is not trivial. What is the right strategy to adopt? Stop one blower when the other is doing the opposite action?

Then two kinds of questions arise in solving actuation conflict. First, how the system must behave under conflict? This is often an end-user concern. Second, how to be sure that the conflict resolution is effective? This is more often a developer concern. So, Actuation Conflict Management Enabler (ACM Enabler) must provide user-friendly test and validation tools for designing the actuation conflict manager.

In previous deliverables we identified a list of requirements. Some are regarding actuation conflict management in use case providers requirements (Table 2). Others are more technical requirements for actuation conflict management (Table 1).

Table 4: Actuation conflict management requirements as defined in D2.1

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.
R2	Usability	Shall integrate a user-friendly tool for actuation conflict manager design, based on simple and interpretable modelling frameworks.
R3	Trustworthiness (reliability)	Shall help identifying direct and indirect actuation conflicts
R4	Reusability	Shall permit the reusability of solutions already designed for similar cases
R5	Trustworthiness (reliability)	Shall aid in the design of the conceptual model of the conflict controller (*e.g.*, Formal test & verification)
R6	Trustworthiness (reliability)	Shall provide tools for testing actuation conflict managers through an operational model (intended to validate conflict resolution solutions in an operational context)
R7	Trustworthiness (reliability)	Shall manage actuation conflicts despite black box components in different targeted platforms
R8	Adequacy	Conflicts resolution at run-time shall be automated as much as possible.
R9	Trustworthiness (safety)	Shall provide actuation conflict alerts during the deployment of a SIS.
R10	Monitoring & trustworthiness (safety)	Shall continuously monitor behavioural drift to assess deployed solutions.
R11	Monitoring & traceability	Shall trigger a new actuation conflict manager development from the quantitative behavioural drift assessment value/threshold.
R12	Scalability	Shall provide tools allowing managing hundreds of sensors and actuators, thus tens of actuation systems.
R13	Scope	Actuation conflicts management tools shall support several targets ranging from IoT, Edge to cloud.
R14	Integration	Actuation conflicts management tools shall support different kind of frameworks (GeneSIS, ThingML, Node-Red) and middleware (SMOOL, SOFIA2, etc.).

Table 5: Actuation Conflict Management Enabler Requirements in D1.1

DO-3.3.1 **DO-3.3.4**	SW updates conflict	INDRA, Rail Use Case	Prioritization of the orders must be done
DO-3.3.2	Monitoring Interface	INDRA, Rail Use Case	The Context Monitoring and Actuation Conflict Management Enabler and the Monitoring Enablers must have a GUI.
DO-3.3.3	Low delay alerting	INDRA, Rail Use Case	Low delays of alerting to the rail operator in order to avoid critical accidents
DO-3.3.6	Conflicts in actuator resource - inter IoT systems	TECNALIA, Smart	The Conflicts Enabler should be able to identify and avoid conflicts in colliding commands sent to same actuator by two IoT apps.

		Building use case	
DO-3.3.7	Conflicts in physical variable actuation - inter IoT systems	TECNALIA, Smart Building use case	The Conflicts Enabler should be able to identify and avoid conflicts in commands sent to actuators affecting the same physical variable, by two IoT apps at the same time.

4.1.2 *Overall Approach*

The ACM Enabler provides a set of tools to assist developer for detecting and solving actuation conflicts. These tools depend on different models provided as inputs to the ACM Enabler. These models are: (1) GeneSIS deployment model, (2) different kinds of internal models of GeneSIS components (*i.e.*, model of deployable artefact's architecture), and (3) physical environment models for the specific case of indirect actuation conflict. In ACM Manager V1, we assume that internal models of the deployable artefacts are assemblies of components like in Node-Red. A physical environment model is only a list of actuators, which interact with each other though the physical environment.

In order to reason on a unique model dedicated to actuation conflict management, a first tool computes a common "*Workflow and Interaction Model for Actuation Conflict Management*" (WIMAC) from the different input models as described above. WIMAC V1, restricted by above assumptions, is defined by the metamodel depicted in Figure 43. WIMAC is used to detect direct and indirect actuation conflicts (both kind of conflicts introduced in section 4.1.1) and solve actuation conflicts by inserting ACM solutions before deployment. They propose two approaches for assisting the developer in managing actuation conflicts:

- A first approach aims at providing some predefined solutions for simultaneously solving a large set of conflicts *in a large-scale SIS*.
- A second approach aims at providing tools for designing a new and reliable solution for a specific and problematic actuation conflict, hereafter called *custom ACM*. The behaviour of the custom ACM is specified and modelled as a Finite State Machine (FSM) for facilitating model checking.

In the first approach, the actuation conflict detection and solving tool is based on graph and model transformation algorithms. Detection and transformation are defined from a set of rules. These rules correspond to some actuation conflict patterns used to (1) detect conflicts within WIMAC and (2) apply corresponding transformations to insert off-the-shelf actuation conflict managers (ACM). Thereby, a new WIMAC is computed, providing a new GeneSIS deployment model and required ACM components.

In the second approach, a design workflow is proposed for designing new and reliable actuation conflict managers aiming at addressing specific and problematic cases. The workflow consists in the following stages:

(1) A new actuation conflict manager (ACM) is described from some Extended ECA rules (Event-Condition-Action), where actions not only govern ACM outputs but may also govern ACM internal state evolution.

(2) The transformation of a set of Extended ECA rules into a required equivalent Mealy FSM is applied, enabling ACM formal verifications. Then, most of the well-known model checkers can be used to verify formal properties such as safety (example in [ressouche2011.enw]).

(3) Finally, the corresponding software components are generated according to the targeted execution platform (*e.g.*, Node-RED, ThingML, …), and instantiated into a new extended GeneSIS deployment model.

Figure 29 illustrates the different steps for managing actuation conflicts with the Enabler. A first step consists in the synthesis of a WIMAC from a GeneSIS model, GeneSIS components internal models (*e.g.*, Node-RED assembly, ThingML) and from the physical environment model. Components that are part of the software orchestration are represented in blue, red components are participating to the actuation system, green components are provided services in the cloud. The second step is based on conflict patterns recognition and associated transformation rules to insert ACM in WIMAC at a large-scale level. Red and green colour filled components depict two different inserted ACM. The third step is required when developer must customize safe and reliable ACM for a problematic conflict point. Thanks to the tools introduced in section 4.2.2, a new ACM is designed and inserted (blue color filled component in Figure 29). Finally, the new WIMAC is then deployed including the implementation of the inserted ACM.

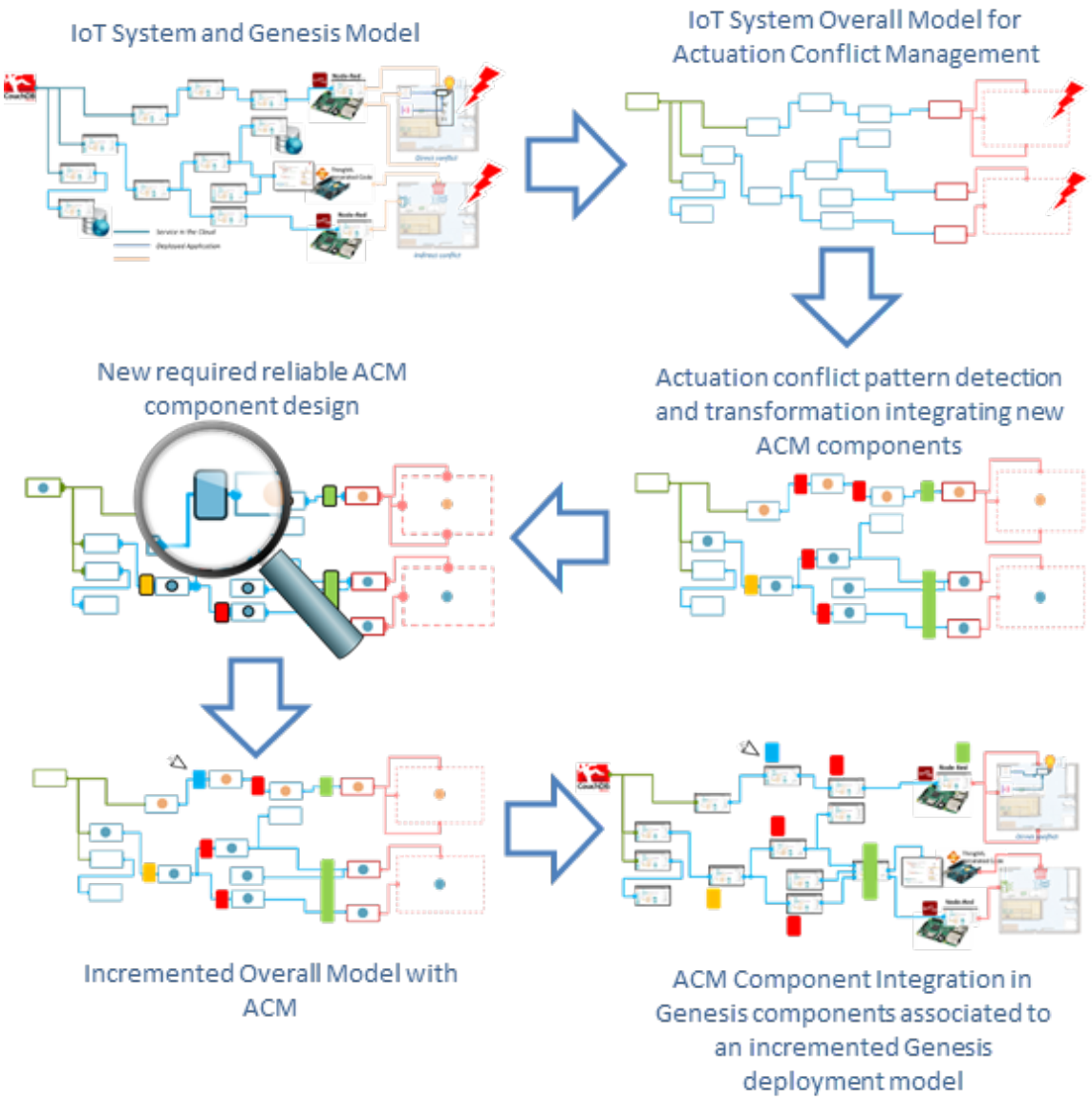


Figure 29: Overall Actuation Conflicts Management Workflow

4.1.3 *Highlights*

The first released set of tools for actuation conflict management in SIS follows several innovative principles associated to the tools:

- *A new language to describe a Workflow and Interaction Model for Actuation Conflict Management (WIMAC):* Actuation conflict management Enabler requires a WIMAC used to detect and manage actuation conflicts. This model is generated automatically from (1) the SIS deployment model (written with the GeneSIS modelling language) (2) the set of GeneSIS components internal models (including but not limited to Node-Red and ThingML) and (3) a model of the impacts of the actuations on the physical environment. To the best of our knowledge there is no language similar to WIMAC focusing on the management of actuation conflicts. WIMAC is one of the main concepts of the actuation conflict management Enabler. Its metamodel will be incremented according to the use case

providers and developers' feedback. A first version of the metamodel is described in section 4.2.3.

- *A generic tool for Actuation Conflict Detection and Solving for large scale SIS:*

  The first challenge, SIS are generally not limited to small sets of sensors and actuators. Because they are often large-scale systems.it consists in providing a tool for deploying ACM solution in a systematic manner (see section 4.2.1).

  This tool must be generic. The tool and its associated algorithms must support possible modifications of WIMAC and new actuation conflict patterns resulting from further experiments. The labelled graph transformation algorithm (*i.e.*, graph rewriting) in use allows to define WIMAC transformations that can be written independently of WIMAC version.

- *Designing Safe and Reliable Custom Actuation Conflict Manager (ACM):*

  The second challenge consists in managing actuation conflicts that may occur and requiring special attention. Tools are then required for designing a custom and reliable actuation conflict manager. This challenge is close to that of the control of hybrid systems[15], that requires some specific tools for designing and testing a reliable controller using sensors and actuators for controlling a local physical environment (see section 4.2.2).

  These two simultaneous challenges justify the innovative approach mixing large scale actuation conflict management, thanks to predefined off-the-shelf solutions, and local conflict management, thanks to some specific and reliable solutions for the problematic cases.

Next section about technical presentation and highlights will detail the different tools that cooperate in ACM Enabler V1.

4.2 Technical presentation

In this technical presentation, Section 4.2.1 describes the tool to detect and solve actuation conflicts at a large-scale level. The solving methodology is only based on predefined actuation conflict solutions. Section 4.2.2 describes the tool that supports the design of specific and reliable ACM components. Finally, section 4.2.3 illustrates the use of our ACM Enabler V1.

4.2.1 *Actuation Conflict Detection and Solving for large scale SIS*

The challenge is to provide a tool for detecting and solving actuation conflicts based on our WIMAC, assuming that SIS are most of the time large-scale systems. So, one needs to find a generic and scalable approach based on generic algorithms that can be applied to WIMAC and the various actuation conflict types.

Thus, we define an appropriate way to modify WIMAC by adding ACMs on the right conflict points. ACMs can either be effective at resolving conflicts, or just *forward components* used for monitoring activity at the conflict point (denoted by "monitors" in the sequel). The method consists in defining some patterns representing typical conflicting types. Then, the transformation rules associated to these patterns are applied on WIMAC model for instantiating

[15] A hybrid system is a dynamical system that exhibits both continuous and discrete dynamic behavior.

ACMs. Numerous tools for graphs and model's transformation are available on the market. Among these tools, a domain independent one, called AGG, is selected so as to be able to handle heterogeneous models.

In the current release of the tool, the "Attributed Graph Grammar" (AGG) is used together with its development environment (called "AGG") which supports an algebraic approach for applying graph transformations. It provides us with the capability to define conflict patterns in the form of an attributed graph and their associated transformation rule also in the form of an attributed graph. Then, AGG manages a set of transformations applied upon corresponding conflict patterns detection.

AGG may seem like a quite heavy solution to apply few transformation rules such as described in the example below. Nevertheless, our objective is to allow developer to add new transformation rules for being able to address some new actuation conflict patterns and to apply a greater number of rules.

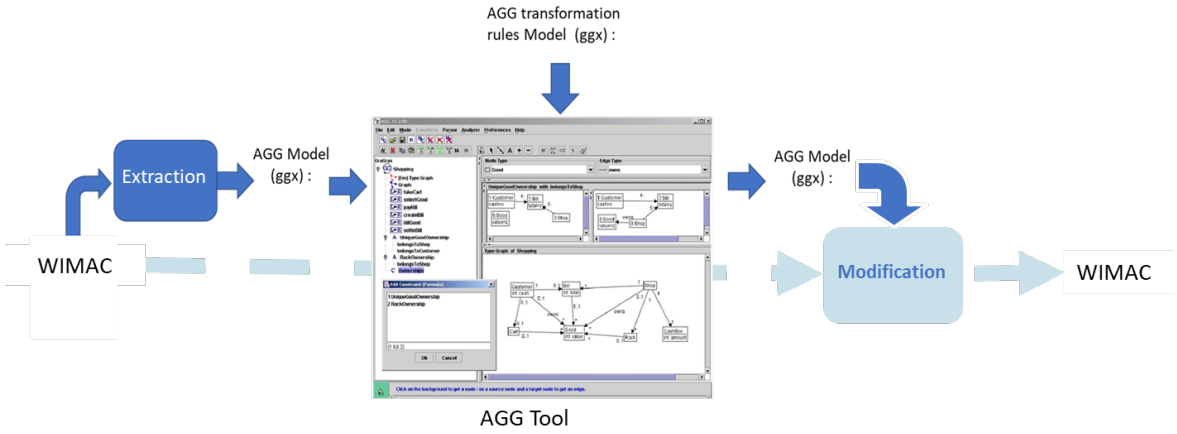


Figure 30: Actuation Conflict Detection and Solving in a large-scale level with AGG

As depicted in Figure 30, in order to integrate AGG into the workflow, AGG model is extracted from WIMAC and after conflicts solving, new AGG model allow to modify WIMAC /. Because WIMAC is likely to evolve during ENACT lifecycle, the aforementioned tools are not dedicated to WIMAC as described in section 4.2.3.

The AGG model is built upon *Entities* representing types of nodes and edges. Each entity has an ID and a name. This model can be modified by *Rules*, composed of two graphs. The first graph is called LHS (Left Hand Side) and represents a sub-graph (a pattern) to be searched for in the AGG model. The second graph is the RHS (Right Hand Side) and represents the transformation to be applied on the AGG model when the pattern described in LHS has been found (Figure 31). The AGG model can be serialized into an XML file.

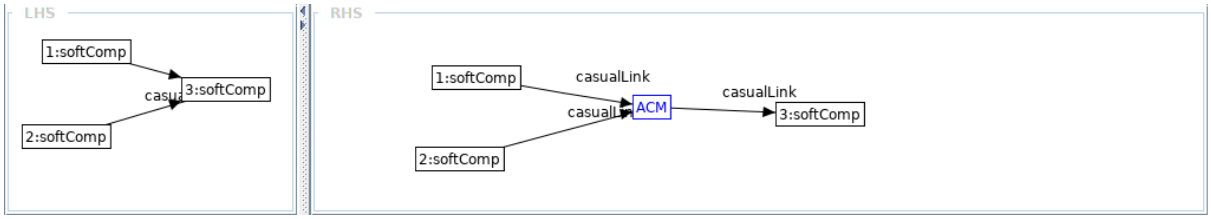


Figure 31: A graph rewriting rule (LHS is search pattern and RHS is how to transform the identified pattern)

As depicted in Figure 31, during an AGG transformation, ACM components are added, resulting from the RHS transformations. WIMAC is first transformed into its AGG counterpart, further complemented with transformation rules and resulting into a *.ggx* project file. It is worth noting that transformation rules are flexible, *i.e.*, they can be added to an existing project, removed or modified on the fly. The transformation process is started as soon as a project file is made available. During this process, transformation rules are applied until none can be applied anymore, resulting in a modified AGG graph. It is worth noting that, in this context, different algorithms will be investigated in order to prevent, for instance, cyclic transformations that may result from AGG transformation process, etc.

In the current version of the ACM Enabler, three main conflict management rules can be applied to the AGG model. The first rule (Figure 32) allows to handle direct conflict **preventing two software components to access the same resource** except if, obviously, this resource is an ACM. The second rule (Figure 33) handles **indirect conflicts by adding an ACM before resources** thereby, preventing counterproductive effects to be produced in the environment through actuators. Finally, the third rule Figure 34 allows to **add monitors to specific resources for further investigations**.

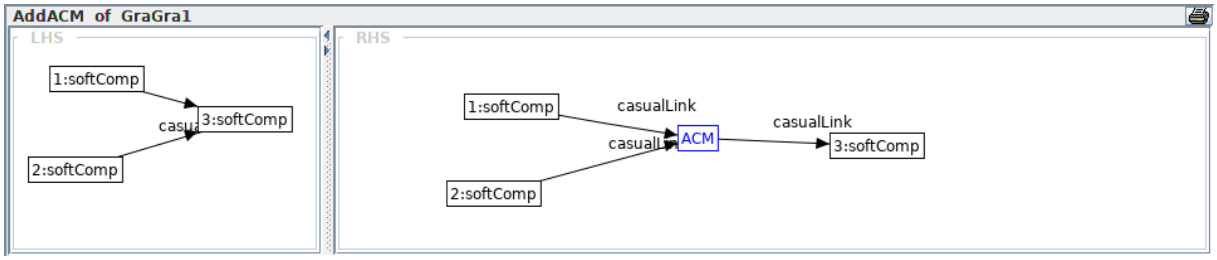


Figure 32: AGG rule for direct conflict

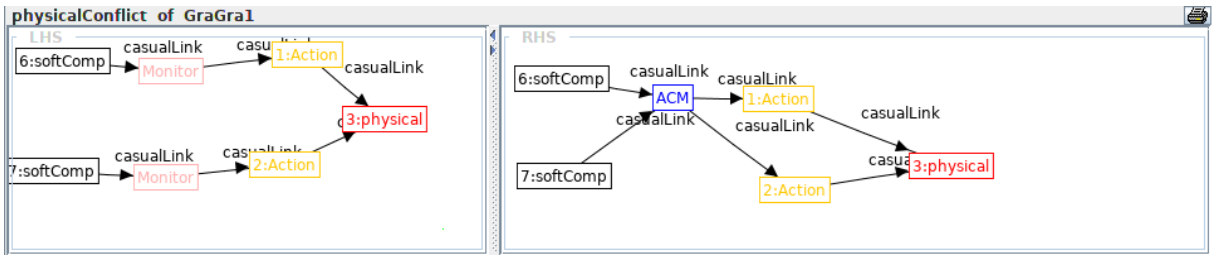


Figure 33: AGG rule for indirect conflict

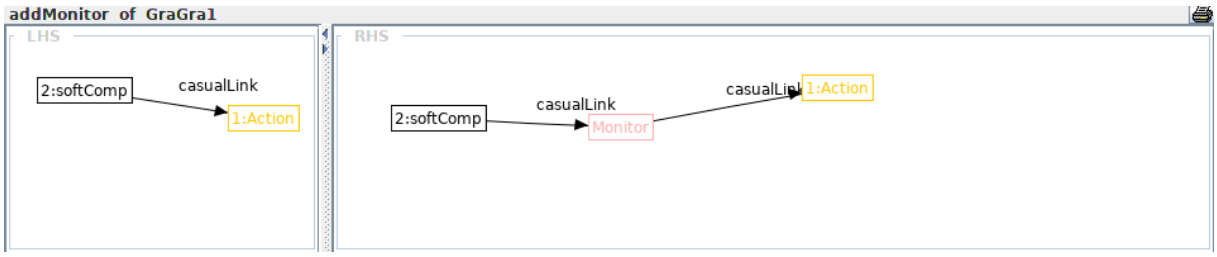


Figure 34: AGG rule to add monitor before actuator (action node)

At the end of the transformation process, the modified AGG model can further be transformed to its WIMAC counterpart.

In the case of blowers (see section 4.1.1), the two independent blowers are indirectly conflicting since both modify the temperature of a shared physical environment. The associated WIMAC model is depicted in Figure 35 where each blower is represented by a software component and an action via actuators, each action being linked to a shared physical process. This model is transformed into its AGG counterpart depicted in Figure 36.

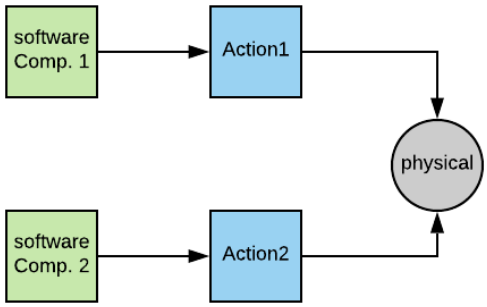


Figure 35: WIMAC model for indirect conflict type

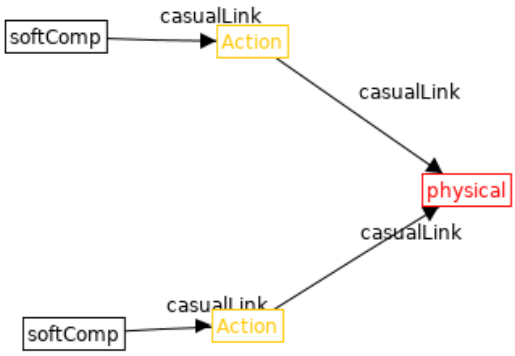


Figure 36: AGG graph corresponding to the WIMAC model depicted in Figure 35.

Once imported, the transformation process is started. It tries to apply the first rule (Figure 32, for direct conflict), but the pattern does not match since, in that particular case, there is no direct conflict. Similarly, the second rule cannot be applied at this step but the third can (Figure 34). Because the algorithm tries to applied all the rules until a last stable graph is reached, the second rule (Figure 33) can thus be finally applied, thereby instantiating an ACM, resulting in the graph depicted in Figure 37.

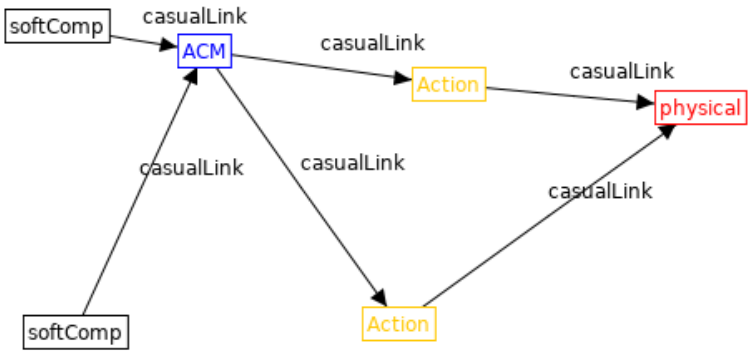


Figure 37: Resulting AGG graph after applying all possible rules.

4.2.2 *Designing Safe and Reliable Custom Actuation Conflict Manager*

The second main challenge for actuation conflict management consists in providing tools for designing reliable actuation conflict managers (*i.e.*, custom ACM) intended to address specific and problematic cases. Because a custom ACM must be validated by our tools chain, we choose to describe it as a finite state machine like Mealy machine [23]. Designing custom ACM may be quite complex but new ACM becomes off-the-shelf ACM for future actuation conflict solving.

4.2.2.1 Actuation conflict Manager Design facilities

In this section, we introduce a toolkit aiming at supporting the design of reliable ACMs and targeting problematic conflict points.

The proposed approach is divided into different modelling levels. The first level focuses on the behavioural logic, which is represented by a finite state machine (FSM). FSM is then described with a modelling language, such as State Chart XML (SCXML), compatible with numerous formal verification tools. The second level focuses on generating FSM execution engine (*i.e.*, implemented ACM Component). FSM execution time is bounded thus keeping the response time of the conflict manager within acceptable delays.

However, designing an FSM can be quite a complicated task, the number of transitions and states being potentially high. A first challenge is then to provide developers with a user-friendly interface. Because of its popularity for End-user programming, especially in the smart building domain, we chose kinds of Event-Condition-Action (ECA) rules to specify ACM. An ECA rule can be read as follows: “On this Event, if the Condition is true, then do this Action”.

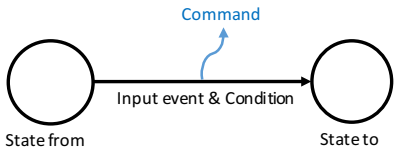
Unfortunately, ECA formalism is close to combinatory logic, free from the concept of state. Thus, FSM states have to be defined when designing FSM from ECA.

This motivates the introduction of Extended ECA programming models which natively define internal states. With Extended ECA rules in hands, we propose to translate them into the Mealy FSM, which can be described through SCXML format. This format is specifically designed for modelling Finite State Machine and possibly provides room for further evolutions through hierarchical features, parallel state and intern metamodel creation.

4.2.2.2 Extended ECA rules to Mealy Finite State Machines

Each extended ECA rule contains all the necessary elements used to define state-transitions in the Mealy FSM. Each state-transition is defined by:

- The *statefrom*, the state-transition starts from,
- The *stateto*, the state-transition ends to,
- An *input event* (possibly multivariate) required to initiate the state-transition,
- A *condition* to be satisfied to grant the state-transition,



- A *command* (Action) to be emitted while transiting from the starting state to the ending state.

A custom ACM contains as many Mealy FSM as devices trying to modify a shared resource. A shared resource can either be an actuator (direct conflict, Figure 38) or a physical property whose value can be modified through different actuators (indirect conflict, see Figure 38 and Figure 39).

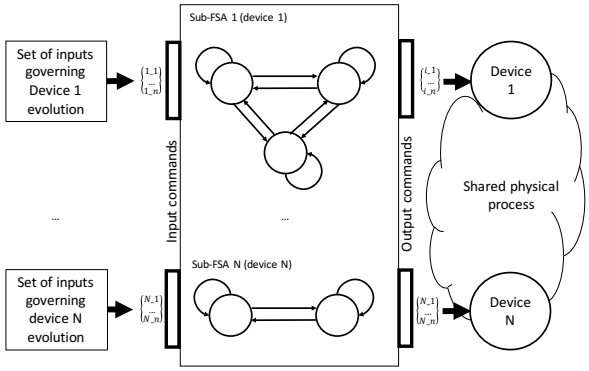


Figure 38: Custom ACM for indirect conflicts mgmt.

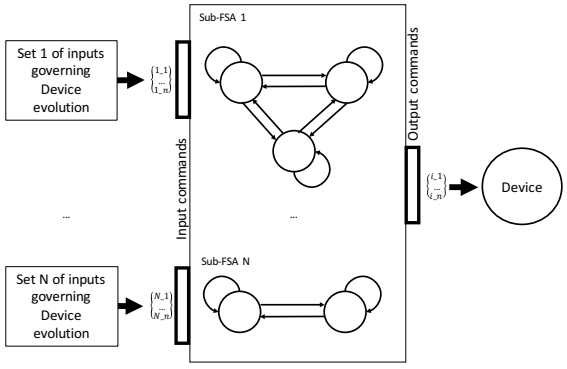


Figure 39: Custom ACM for direct conflicts mgmt.

Each Mealy FSM represents the evolution of the commands to be sent to the actuator(s) in response to the input events, potentially inhibited if the defined state-transition condition is not satisfied.

An extended ECA rule is defined as follows:

```
ON EVENT IF CONDITION DO ACTION
```

Where:

```
EVENT          : Input(<value>)       OR INIT()
CONDITION      : State(<state_name>)
ACTION         : State(<state_name>) OR Output(<value>)
```

For instance, a state-transition from the state “s_1” to the state “s_2” can be described by:

```
ON Input(i) IF State(“s_1”) DO State(“s_2”)
```

Then, a command emission can be described by:

```
ON Input(i) IF State(“s_1”) DO Output(o)
```

Finally, one can define an initial state to start with by:

```
ON Init() IF true DO State(“s_1”)
```

Let us use the examples provided in section 4.1.1 to illustrate how to generate a Mealy FSM from an Extended ECA programming model. The Extended ECA code snippet below is an implementation of an indirect conflict manager inherent to blowers trying to modify the temperature of a shared physical environment. The main (yet simple) idea underlying this implementation is to prevent counterproductive effects to be produced in the environment by

filtering commands before sending them to the blowers. For instance, if a “doCold” command is required to be sent to a blower while the current state of the other blower is “Warm”, then the command is inhibited.

```
/*---------------
/* INITIALIZATION
/*---------------
ON Init() IF true DO State(“doingWarm_1”) //Blower 1
ON Init() IF true DO State(“doingWarm_2”) //Blower 2

/*----------------------------
/* BLOWER 1 extended ECA rules
/*----------------------------

ON Input(“doWarm_1”) IF State(“doingWarm_1”) DO State(“doingWarm_1”)
ON Input(“doWarm_1”) IF State(“doingCold_1”) DO State(“doingWarm_1”)
ON Input(“doCold_1”) IF State(“doingCold_1”) DO State(“doingCold_1”)
ON Input(“doCold_1”) IF State(“doingWarm_1”) DO State(“doingCold_1”)

ON Input(“doWarm_1”) IF State(“doingWarm_1”) DO Output(“doWarm_1”)
ON Input(“doCold_1”) IF State(“doingCold_1”) DO Output(“doCold_1”)
ON Input(“doWarm_1”) IF State(“doingWarm_2”) DO Output(“doWarm_1”)
ON Input(“doCold_1”) IF State(“doingCold_2”) DO Output(“doCold_1”)

/*----------------------------
/* BLOWER 2 extended ECA rules
/*----------------------------

ON Input(“doWarm_2”) IF State(“doingWarm_2”) DO State(“doingWarm_2”)
ON Input(“doWarm_2”) IF State(“doingCold_2”) DO State(“doingWarm_2”)
ON Input(“doCold_2”) IF State(“doingCold_2”) DO State(“doingCold_2”)
ON Input(“doCold_2”) IF State(“doingWarm_2”) DO State(“doingCold_2”)

ON Input(“doWarm_2”) IF State(“doingWarm_2”) DO Output(“doWarm_2”)
ON Input(“doCold_2”) IF State(“doingCold_2”) DO Output(“doCold_2”)
ON Input(“doWarm_2”) IF State(“doingWarm_1”) DO Output(“doWarm_2”)
ON Input(“doCold_2”) IF State(“doingCold_1”) DO Output(“doCold_2”)

/*-------------------------------------------------
/* Example of priority rules (blower 2 prioritized)
/*-------------------------------------------------
ON Input(“doWarm_2”) IF State(“doingCold_1”) DO Output(“doWarm_2”)
ON Input(“doCold_2”) IF State(“doingWarm_1”) DO Output(“doCold_2”)

ON Input(“doWarm_1”) IF State(“doingCold_2”) DO Output(“doCold_1”)
ON Input(“doCold_1”) IF State(“doingWarm_2”) DO Output(“doWarm_1”)
```

By parsing extended ECA rules, one can incrementally build the custom ACM depicted in Figure 40.

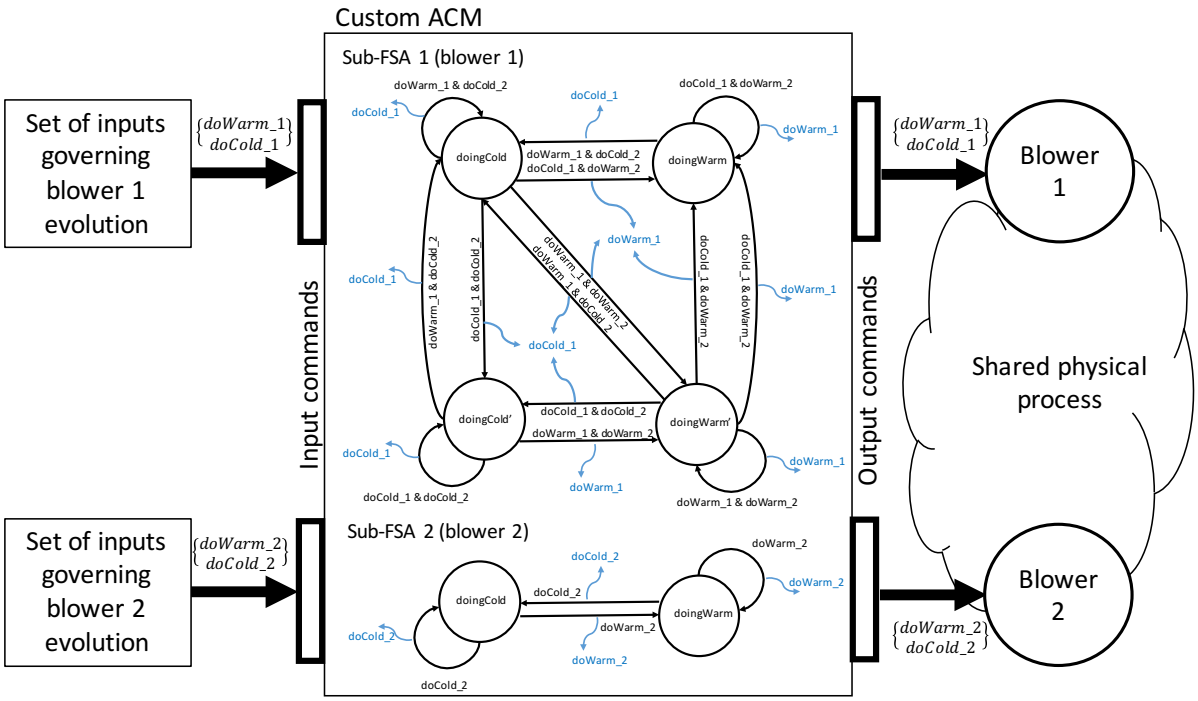


Figure 40: Custom ACM from extended ECA rules

4.2.2.3 Validation tools and ACM component generation

Before instantiating an ACM into the AGG, some formal verifications (providing mathematical proof of correctness) are needed to ensure its behavioural reliability. For this purpose, the Mealy FSM built from Extended ECA rules is transformed into SCXML format further used as input to NuSMV [24].

In a short-term roadmap, logical constrains specifications (*e.g.*, two outputs are incompatible) will be added. These logical constrains may also be described using extended ECA rules and an equivalent Mealy FSM, called ACM observer. Contrary to ACM, ACM observer consumes as inputs the inputs and outputs of ACM. Its output is either “OK” or “KO” indicating when logical constrains are verified or not.

Both Mealy FSM feed NuSMV and then some formal properties can be verified using its model-checker. For example, formal verification of the ***Safety*** property “ensures that something bad (KO) will never happen”.

This formal verification approach is close to the one described in [25].

After the verification of the ACM logic, the custom ACM implementation is done through a generic execution engine (see Figure 41). Our FSM execution engine is an event driven. Thus, a generator must compute input events before triggering one FSM transition and generating the corresponding outputs.

In ACM Enabler V1, the targeted platform is a Node-Red one.

In future works, tools for testing FSM with their execution engine before deployment will be developed to ensure that the logical constrains remain true independently from their implementation. Then different generic execution engine policies may be tested to automatic choose the most reliable one for one FSM and the corresponding custom ACM.

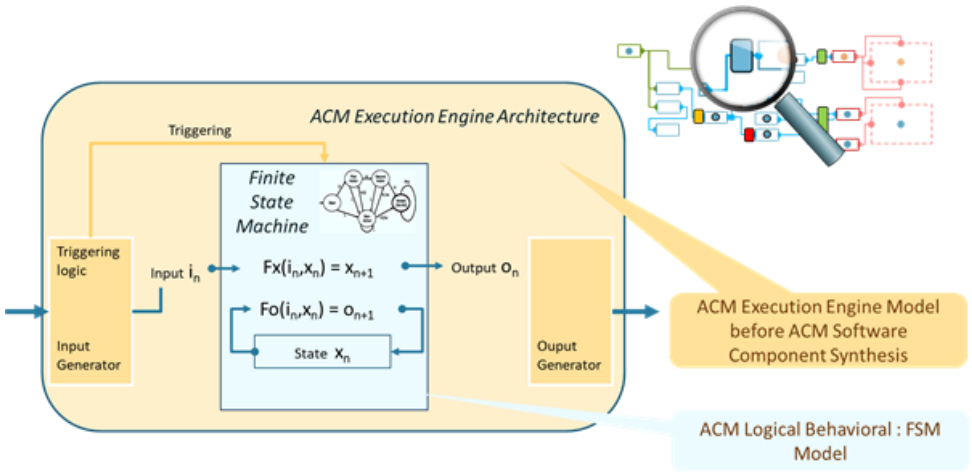


Figure 41: Execution engine for a Finite State Mealy Machine and custom ACM software component

In summary, the proposed approach for designing a custom ACM is depicted in Figure 42. First extended ECA rules allow to specify an ACM logic. This set of rules are thus transformed into a Mealy FSM. Other sets of ECA rules describe some constrains such as undesired ACM inputs/outputs. They are then transformed into a second Mealy FSM. From both FSM, a Model Checker allows to prove (or not) that constrains are validated.

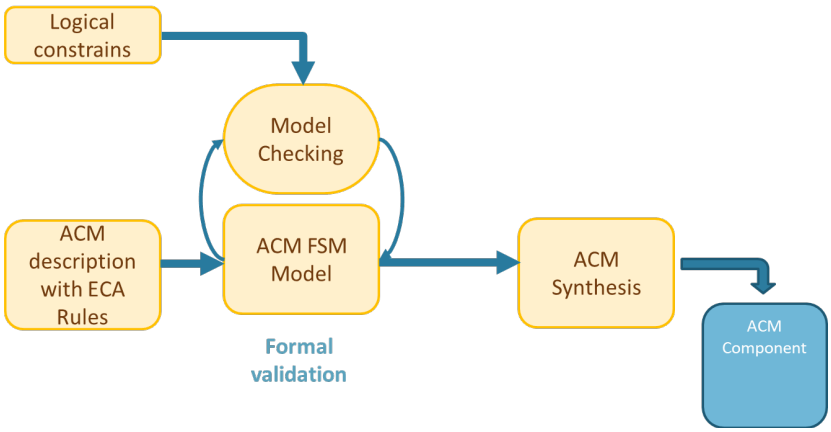


Figure 42: Designing Safe and Reliable Custom Actuation Conflict Manager Workflow

4.2.3 *ACM Enabler V1 Illustration*

In the first release of the ACM Enabler, an actuation conflict corresponds either to concurrent commands applied on shared actuators (direct conflict) or to concurrent commands applied to distinct actuators whose effects impact a shared physical environment (indirect conflict).

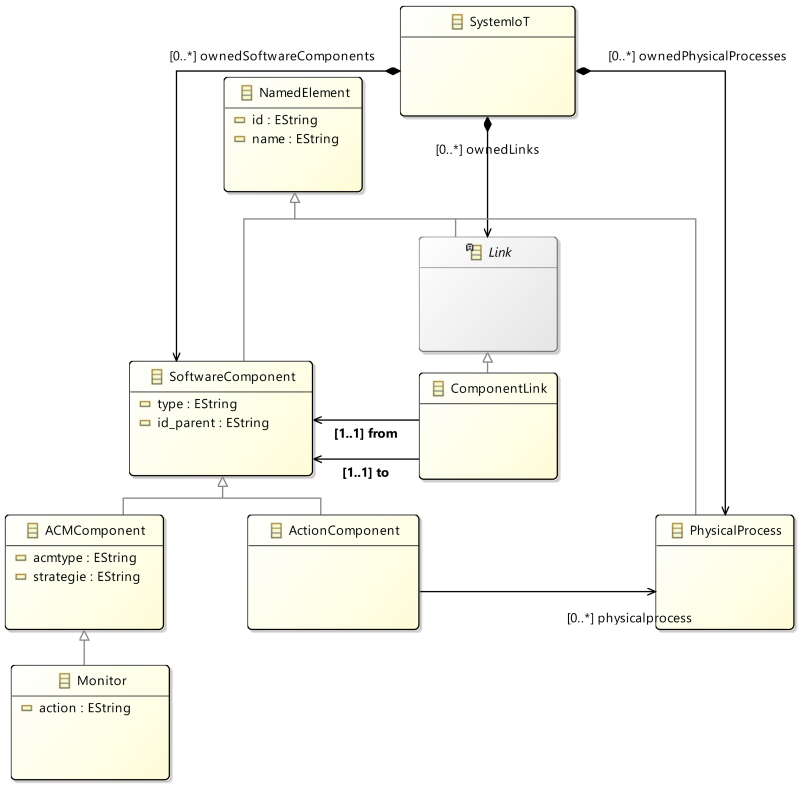


Figure 43: WIMAC meta-model V1

The main entities in the current WIMAC metamodel are the following (see Figure 43):

(1) ***SoftwareComponents*** –In V1, software components are black-box components. Thus, actuation conflict management is solved externally, *i.e.*, by instantiating *ACM components* without modifying existing software components.

(2) ***ActionComponents*** are Software components controlling transducers that modify the physical process, *e.g.*, actuators,

(3) ***Physical Process*** corresponds to a bounded part of the physical environment, *e.g.*, temperature in a room. In V1, the models of the physical processes of interest allow to detect indirect conflicts inherent to *ActionComponents.*

Given this meta-model, a concrete implementation is illustrated in the sequel.

As presented in section 4.1.1, we consider two different applications whose purpose is to control a shared light. The first application (*App1*) switches the light on when a presence is detected in the room while the second application (*App2*) switches the light off when the TV is in use. Both applications (Figure 44) act on the same device, *i.e.*, the light. In this context, the lack of conflict management may result in the light to blink perpetually.

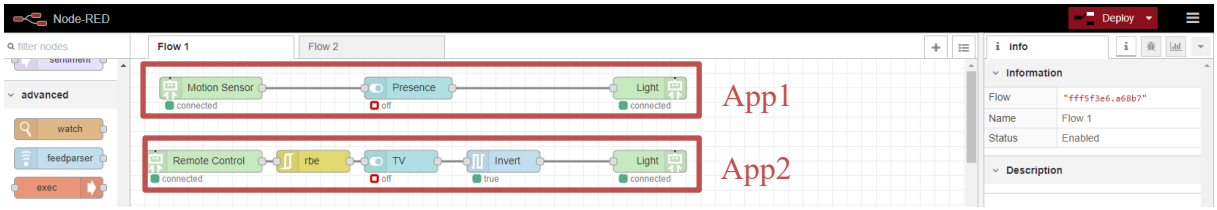


Figure 44: Two applications acting on the same device "Light"

Following the process described in 4.2.1, the associated WIMAC model, with an off-the-shelf ACM instantiated, is depicted in Figure 45.

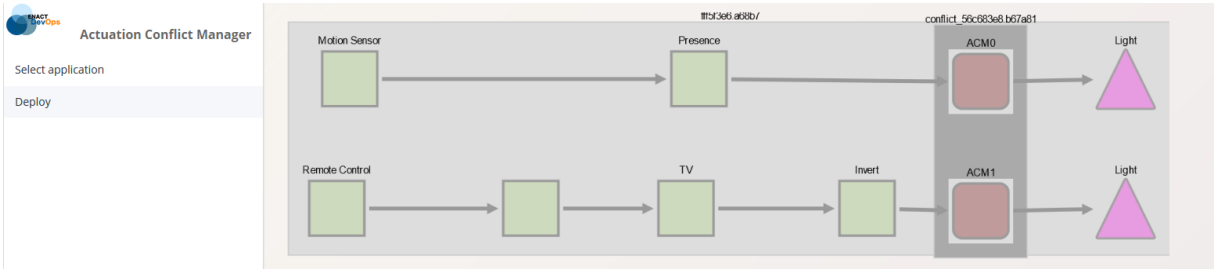


Figure 45: Actuation conflict identification and resolution

From the application model depicted in 4.1.1, there is no way to infer that both lights do actually refer as explained in 4.2.1 (illustrated by Figure 35 and Figure 36) identified by an ID in the model of the physical environment and "links" describes the fact that both lights interact with each other in the physical process:

```
{
        "physical_processes": [{
                        "id": "56c683e8.b67a81",
                        "name": "PP_LightLivingRoom"
                }
        ],
        "links": [{
                        "from_id": "1a0c80b8.1c9e9f",     // Light#1
                        "to_id": "56c683e8.b67a81"        // Acts on this process
                }f, {
                        "from_id": "57f66f91.3c12",       // Light#2
                        "to_id": "56c683e8.b67a81"        // Acts on the same process
                }
        ]
}
```

This allows to identify a potential conflict, hence the instantiation of the ACM in front of the lights in the WIMAC model. For this example, the ACM corresponds to an AND gate. The resulting Node-Red application is depicted in Figure 46.

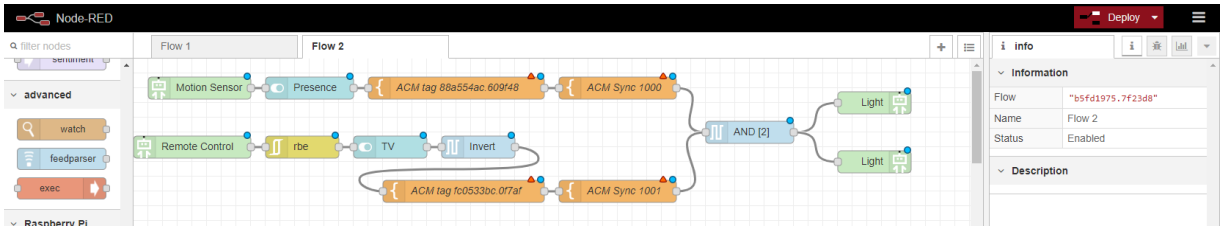


Figure 46: Generated application model with actuation conflict management

4.3 Synthesis

In the sequel, we do evaluate how the proposed approach addresses the requirements defined in Section 4.1.1 with respect to the use case requirements described in deliverables D2.1 and D1.1.

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.	**Partially covered.** ACM Enabler V1 is a chain of tools with a step by step design methodology. Enabler will be enhanced for reusing of custom ACM and for testing FSM execution engine before deployment.
R2	Usability	Shall integrate a GUI tool for actuation modelling support, based on simple and interpretable modelling frameworks.	**Ongoing.** Large-scale Actuation Conflict Management uses AGG tool. This tool provides a simple yet efficient 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 environment *tal actuation conflicts	**Partially covered**. ACM Enabler V1 provides an actuation conflict management considering indirect conflicts that may occur through a shared physical environment. The environment model is simple in this first release. Future work will introduce more complex physical environment models.
R4	Reusability	Shall permit the reusability of solutions already designed for similar cases	**Partially 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. Future work will introduce a knowledge base used to store ACMs incremented with semantic metadata, thereby facilitating ACMs search through semantic queries.
R5	Trustworthiness (reliability)	Shall aid in the design of the conceptual model of the conflict controller (*e.g.*, Formal test & verification)	**Partially 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 controller through an operational model (intended to validate conflict resolution solutions in an operational context)	**Not yet available.** 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 black box modelled components.	**Available**. ACM Enabler V1 manages actuation conflicts despite black box components (see WIMAC model V1).
R8	Adequacy	Conflicts resolution at run-time shall be automated as much as possible.	**Not yet available.** ACM Enabler V1 automatically detects actuation conflicts before deployment and provides developers tools to solve it.
R9	Trustworthiness (safety)	Shall provide actuation conflicts alerts during the deployment on ENACT platform.	**Not yet available.** ACM Enabler V1 automatically detects actuation conflicts before deployment and provides developers tools to solve it.
R10	Monitoring & trustworthiness (safety)	Shall continuously monitor behavioural drift to assess deployed solutions.	**Not yet available.** This is part of the roadmap with UDE.
R11	Monitoring & traceability	Shall trig a new development cycle from the quantitative behavioural drift assessment value/threshold.	**Not yet available.** It is planned in the roadmap and requires WP3 and WP2 Tools to be merged to the common ENACT DevOps platform.
R12	Scalability	Shall provide tools allowing to manage hundreds of sensors and actuators, thus tens of actuation systems.	**Partially covered**. Large scale Actuation Conflict Detection and Solving is one of the focus of the ACM Enabler V1. Further tests are planned in the roadmap.
R13	Scope	Actuation conflicts management tools shall support several targets ranging from IoT, Edge to Cloud.	**Partially covered**. Leveraging on GeneSIS and WIMAC models, the actuation conflict management is compatible with several targets ranging from IoT, Edge to Cloud. In V1, ACMs are instantiated as nodes in Node-Red middleware embedded in a Docker container running on Raspberry PI.
R14	Integration	Actuation conflicts management tools shall support different kind of frameworks (GeneSIS, ThingML, Node-Red) and middleware (SMOOL, SOFIA2, etc.).	**Partially covered**. In V1, ACMs are instantiated as nodes in Node-Red middleware embedded in a Docker container running on Raspberry PI. ACMs are deployed thanks to GeneSIS. SMOOL and FIWARE middleware will be addressed in a future version of the tool.

DO-3.3.1 **DO-3.3.4**	INDRA, Rail Use Case	The orders must be prioritized	**Available.** In V1, 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.	**Not yet available.** It requires WP2 and WP3 tools to be integrated.

DO-3.3.3	INDRA, Rail Use Case	Low delays of alerting to the rail operator in order to avoid critical accidents	**Available**. In V1, 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 Conflicts Enabler should be able to identify and avoid conflicts in colliding commands sent to same actuator by two IoT apps.	**Available.** In V1, direct actuation conflicts are detected and solved.
DO-3.3.7	TECNALIA, Smart Building use case	The Conflicts 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.	**Available.** In V1, indirect actuation conflicts are managed, considering counterproductive effects that may be produced by different actuators sharing a common environment.

Work in progress is focused on evaluating ACM Enabler V1 performances on use cases provided by TECNALIA.

Following this status, several improvements are planned in the roadmap as follows:

(1) Possible WIMAC evolutions are currently being investigated. A first evolution consists in considering grey-box component models describing interactions between their input and output ports. A second envisioned evolution consists in introducing analytical models of the physical environment based, for instance, on differential equations.

(2) For the time being, application and deployment models' part of WIMAC are flattened leading any hierarchical structure to get lost. A possible evolution would consist in conserving existing hierarchical structures upon application and deployment model integration into WIMAC.

(3) Off-the-shelf ACMs are, for the time being, merely stored into a repository. A possible evolution consists in incrementing ACMs with semantic annotations relying on a common ontology and store the result into a Knowledge Base (KB). Doing so, one can leverage semantic web tools like SPARQL [26] (SPARQL Protocol and RDF Query Language) for searching relevant ACMs in the KB.

(4) Finally, custom ACMs formal verifications can be complemented with tests on the concrete ACMs. For instance, Discrete Event System Specification (DEVS) [27] models of the ACM components would allow to test if all the validated properties are preserved on the concrete ACM components.

5 Test and Simulation for SIS

5.1 Overall presentation of the Enabler

Within this section we cover the Test and Simulation ENACT enabler for SIS. Given the fact that this work has been taken over by Beawre as the continuation of work of CA Technologies, former partner of ENACT, and the fact that the partner recently joined the project, we will be covering the motivation and technical presentation based on the current scope of work.

5.1.1 *Motivation*

Software testing is a crucial step of any software development process, even more so in the DevOps software development cycle. In IoT oriented application development however, 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. Even more so in IoT environments, developers need to test their applications to ensure trustworthiness factors are met and well concluded.

In IoT based applications, access to devices, sensors and actuators with a specific environment where they will be deployed might not be trivial, or it can be limited due to many factors. Networks of physical 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 hence the revenue generated by the system— 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 requirements completion, providing developers with testing environment to —at least— start and execute testing of their applications. When it comes to IoT application testing, simulation tools allow developers to have an initial testing platform that enables them to develop their applications before putting them into a production IoT devices network. This way, the impact of the application development on IoT systems is minimized. The shortcoming of the simulators is apparent when the application is relying on externally deployed or distributed network of sensors and/or actuators measuring providing input to the core of the application where the actuators are necessary for the sole application functioning. These scenarios are covered by IoT testbeds.

IoT testbeds also play a relevant role when it comes to testing applications. Testbeds offer a deployed network of IoT devices. Developers can upload their applications onto these networks and test their software in a real environment. IoT-Lab[16] and SmartSantander[17] 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 custom environment and more control over it. Furthermore, simulators avoid the need of having a physical network of devices.

In the recent years, both academia and the commercial market offered solutions in the IoT simulation field. Although the area is the same, their approach is entirely different. Academic solutions implement cutting-edge technology in the form of proofs-of-concept, which are usually not ready for production systems demands. Furthermore, 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.

All of the above showcases 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 sensors and actuators data which does make sense in the given scenario but also stresses the boundaries of the scenario in order to detect potential problems. In Table 6, the scope of common IoT testing is listed. ENACT Test and Simulation Enabler addresses all of the categories apart from

[16] https://www.iot-lab.info/
[17] http://www.smartsantander.eu/

the ones bounded to the components and communication testing which are purely connected to the hardware testing and as such are considered out of the scope of purpose.

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 handing • 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 • Users Roles and Responsibility & its Usage Pattern
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

Table 6. Example test condition of IoT testing scopes

5.1.2 *Overall approach to Test and Simulation*

ENACT approach to Test and Simulation is trying to fill in the gap of how the application of distributed of IoT testing is usually performed. Based on the conclusions of challenges of Test and Simulation described in D2.1 section 5.1.1, the aim of the enabler is to provide means to test the trustworthiness of the IoT application by providing believable, constrained and reason backed sensors and actuators feedback data into the application. We believe the approach we will explain in detail in section 5.2 addresses main challenges and provides a solid baseline for the extension on-top of the enabler to advance the simulation and test capabilities.

Contrary to initial direction of the enabler, the current approach is to be agnostic to the data stream model format and cover the range of the scenarios where the major player in the space could be used. The approach dictates that the real environment communication traffic can be caught and modelled in order to produce stable variant stream of data for the application that in effect can be used for application testing.

Table 7 showcase the scope of the testing capabilities ENACT Test and Simulation tool can cover.

IOT elements Testing Types	Sensor	Application	Network
Functional Testing	True	True	False
Usability Testing	True	True	False
Security Testing	True	True	True
Performance Testing	False	True	True
Compatibility Testing	True	True	False
Services Testing	False	True	True
Operational Testing	True	True	False

Table 7. Scope of the ENACT Test and Simulation Capabilities

By allowing wide range of testing types, sensors and actuators, traffic models can be used by users to compose testing scenarios where the range of tests of the most common IoT testing problem are performed. For instance, one of such examples is IoT scale where the specific behaviour of the modelled traffic from the device can playback faulty data testing rigidity and composition of the application at faulty scenarios.
Range of common pre-recorded simulated devices will be released with the enabler in order to ease the adoption.

5.2 Technical presentation and highlights

Within this section we will introduce the two main categories that compose the Test and Simulation Enabler. Figure 47 depicts modules composing the tool as well as the inputs and outputs to which the enabler is targeted.

Actual Devices refer to a live system device where the traffic can be captured but also testbeds or existing virtual devices to which the tool can be pointed. Ideally, these are the devices that represent the stable state of the required environment. If such devices are not available, the Recording Storage of devices will be available to the developers in order to choose from the

most common devices used in IoT environments. The set of the most common devices is determined by the project use cases.

IoT Application refers to the codebase of the application which can be instantiated on the test server and which expects the inputs from the devices in order to perform actions.

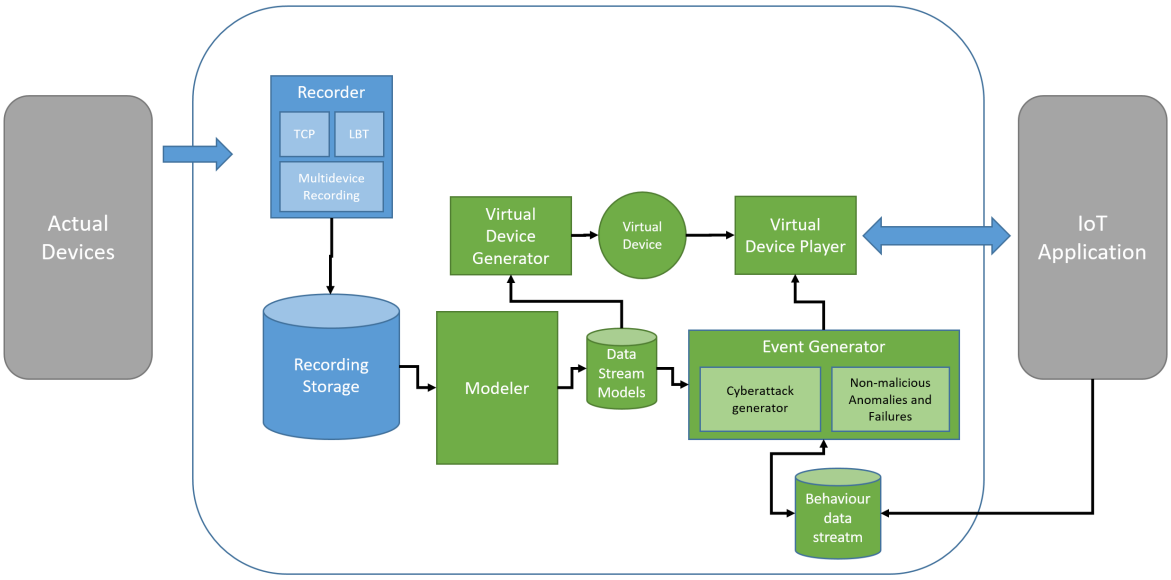


Figure 47. ENACT Test and Simulation Enabler architecture.

From a high-level perspective, Testing and Simulation enabler is envisioned as a package that comprises two main differentiated modules.

First module handles the Simulation itself, extracting the behaviour of actual devices and modelling it to create virtual devices that later on will be played back. Architectural information of the system to be simulated can be passed as input so the simulator will know what devices need to point to in order to sniff its traces and extract its behaviour. This is carried from the GeneSIS model and is graphically represented in the tool in order to ease selection of what devices should be modelled by the simulator.

In parallel, the second main module will manage Events Generation. This module will provide most of the value related to trustworthiness. It can simulate both malicious and non-malicious events that could modify the normal operation of applications or systems. It enables developers to anticipate potential issues and allow them to build a more resilient and robust applications, as well as to anticipate security problems regarding cyberattacks.

During simulation, developers are able to monitor the simulation and to evaluate the results for testing once the simulation is finished. To that matter, developers can upload a description of the alerts and tests to assess during the simulation. In the next section we will introduce the immediate plans for the tool as well as the scope of the final deliverable.

5.3 Future Plans

This section covers the future plans for the enabler, covering the base plan as well as the specific points of innovation which ENACT is aiming to deliver in the enabler.

The baseline aim for the project is to deliver exploitation ready tools for testing and simulating IoT environments, to this extend the enabler will be released open-source under ETL license with Eclipse Foundation back up in order to reach the appropriate communities. The tool is deployable though container which encapsulates SaaS application for IoT Test and simulation. The UI of the Test and Simulation tool is coherent with the rest of the tools of the ENACT framework. On-top of the planned schedule of work set of innovation has been identified and it currently undergoing definition and implementation, these are:

- **Multi device recording** – in order to represent the state of the application, the testing should consist of coherent information's across the sensors, such that the data sent to the application is coherent and represents actual state of the recorded environment. To address this, a multi-device recording capabilities are planned that will allow capturing the data streams from multiple devices at the same time, annotating it with the timestamp so the analysis of dependencies can be performed. This will also enable the multi-device model to be formed capturing the dependencies between the environment.
- **Risk analysis extension –** assessing likelihood and impact of a risky situation is often subjective in nature or it does represent the state of the knowledge of the actor involved in the risk analysis process. By allowing the expressed technical risky situation to be simulated, we want to provide means of assurance of appropriate level of likelihood and impact on the IoT application. We find the concept to be extremely powerful with great potential to be exploited by the communities and commercially. Risky situations are meant to be matched to the testing scenarios and the impact and likelihood analysis report will be presented to the risk analysis user for further analysis and validation. This is solely possible due to the shared architectural model provided by GeneSIS.
- **Model Variance control –** the data streams models are based on the actual recording of the devices, such as TCP or Bluetooth traffic. Model Variance Control will be able to produce new models based on the existing one with the variance of output specified by the actor using the enabler. This will allow for mimicking edge case scenarios as well as emit fault or attack like events to the application.
- **Event Generator for specific purposes –** two main purposes are considered within the ENACT Test and simulation tool for automatic variance introduction as the aim is to produce specific scenarios against which the application can be tested, these are:
    - **Cyberattack events –** where the sole purpose is to take control of the application component in order to extract the data.
    - **Non-malicious Anomalies and Faults –** where the aim is to produce faulty state of the application making it unavailable or behaving in an unprecedented way.
- **Recorder Interoperability –** CA was intending to use Knowthings.io as a backbone of the Test and Simulation tool. The main major step is to support multiple recording streams in order to interoperate between the data stream sources and enhance recording capabilities. For the scope of the project, we aim to support three traffic recorders, these are:
    - **Knowthinks.io legacy / WireShark –** the project is not currently active, but the project is directly interoperable with WireShark type of TCP output capture. ENACT Test and Simulation will support both formats.

- **iFogSim**[18] **–** is a toolkit for Modelling and Simulation of Resource Management Techniques in Internet of Things, Edge and Fog Computing Environments. Interoperability with iFogSim would allow us to simulate the environments which are extremely difficult to record, such as sensors deployed in extreme scenarios
- **MtM IoT –** commercial solution of Montimage, newly joined as a partner in the ENACT consortium, capable of distributed IoT application and systems monitoring and traffic analysis, it would aim to provide a replacement for responsibilities of Knowhtings.io

Initial version covering the first approach of the mentioned capabilities is planned to be release in M21 of the project.

5.4 Synthesis

In the following section, it showcased 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 below Intelligent Transportation System use case, is referred to as UC1, and the Digital Health use case, referred to as UC2.

ReqID	Requirement	Description	Coverage
UC1 **R1**	Scalability	The simulator should easily scale the number of components that are involved in the simulation.	Covered by the approach where the data stream model recording, or event model can be indefinitely replicated and targeted against the application in order to simulate the system at scale.
UC1 **R2**	Component modelling	The simulator should model virtual devices that reproduce the behaviour of real devices.	The approach is to record the IO of the component which in effect can reproduce the behaviour.
UC1 **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	The enabler is capable of covering TCP and Low Power Bluetooth traffic, enabling the recorder interoperability allows for an easy extension of the data streams models to be captured from other type of communication.

[18] Harshit Gupta, Amir Vahid Dastjerdi , Soumya K. Ghosh, and Rajkumar Buyya, iFogSim: A Toolkit for Modeling and Simulation of Resource Management Techniques in Internet of Things, Edge and Fog Computing Environments, Software: Practice and Experience (SPE), Volume 47, Issue 9, Pages: 1275-1296, ISSN: 0038-0644, Wiley Press, New York, USA, September 2017.

		amplifier circuits is being developed by BOSC. Particular IC is not chosen yet.	
UC1 **R4**	Simulation of multiple communication protocols	The simulator should simulate several communication protocols: – IEEE 802.15.4 ZigBee – IEEE 802.3	To be implemented.
UC1 **R5**	Simulation of dynamic geographical position	The simulator should simulate changes of geographical position of the system, taking into account possible mobile network disconnections, and other possible situations derived of the position changes of the system physical platform.	Covered by the approach as this is the model of the sensor data. It is also covered by the event generator which aims to introduce the faulty scenario into the system.
UC1 **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.	As above.
UC1 **R7**	Attack simulation	The simulator should generate possible attacks to the system. Attacks include data poisoning, device disconnection, or device hijacking.	Event generation described in the section 5.3 Future plans aims to address exactly that requirement. Current status: under development.
UC2 **R1**	Scalability	The simulator should easily scale the number of components that are involved in the simulation.	As per UC1 R1.
UC2 **R3**	Simulation of multiple sensor events	The simulator should generate signals from multiple types of sensors listed in D1.1	Multi device recording capabilities described in the section 5.3 Future plans aims to address exactly that requirement. Current status: under development.
UC2 **R4**	Simulation of multiple communication protocols	The simulator should simulate several communication protocols as described in D1.1	Interoperability ensures such capabilities. For the scope of the project, TCP and Bluetooth will be covered as a baseline. Use cases will be also assisted to record the other type of protocols with the existing data stream recorders.
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.	As per UC1 R5
UC2 **R7**	Attack simulation	The simulator should generate possible attacks to the system. Attacks include data poisoning, device disconnection, or device hijacking.	As per UC1 R7
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.	Test and Simulator Enabler is capable of sending the data to any application it is pointed to, regardless of it being instantiated during the test scenario run pointed to already existing application.
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 the approach. Currently in the development phase.

6 Conclusion

WP2 aims at providing enablers for the development and deployment of SIS. WP2 will provide enablers with capabilities to *(i)* manage risk, *(ii)* orchestrate and deploy software components, *(iii)* identify, analyse and manage actuation conflict, and *(iv)* test and simulate provided services.

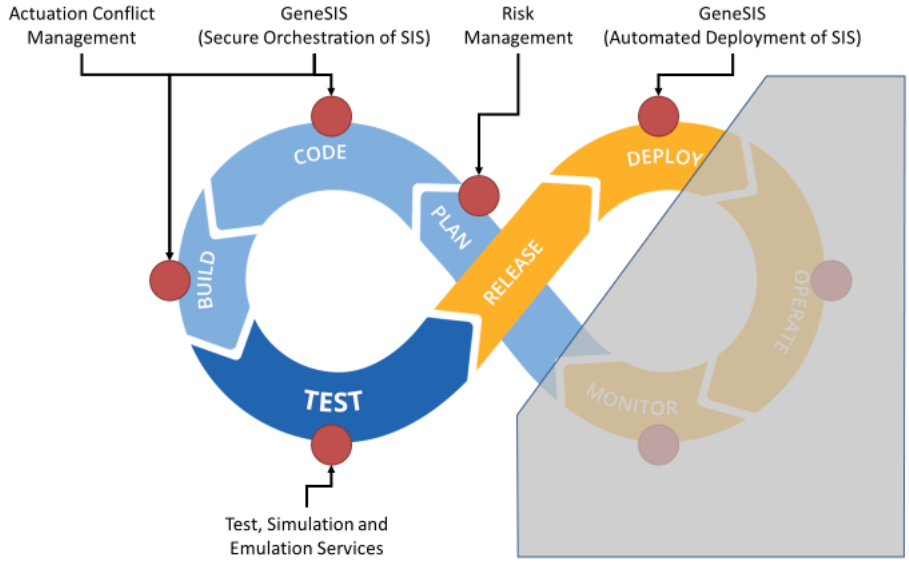


Figure 48. WP2 Enablers for the Dev part of the DevOps cycle

Based on the conceptual designs described in D2.1 we revised the conceptual designs in D2.2 and provided prototypical implementations for all WP2 enablers, except the test and simulation enabler which is ongoing. These first versions of the enablers will serve as a baseline for the final enablers delivery in D2.3. From now, a major focus will be on interacting and supporting case study partners and to account for and evolve the languages and tools accordingly.

Appendix A

1 Risk Management -- User guide

In the following section we summarize the main instruction to run and configure the ENACT Risk Management tool. The code is open source and available under: https://gitlab.com/enact/risk-management

1.1 Installation

The tool follows the SaaS model and hence needs to have a prerequisite of database available to in order to start it. Currently supported database is MSSQL version 2014 onwards.

The typical scenario of run would involve restoring the database schema with running the docker container holding the application logic and API's.

1.1.1 *Database setup*

You can find the database schema dump under root of the repository.

1. Copy the database schema to your local drive

```
wget -L https://gitlab.com/enact/risk-
management/database_schema.sql
```

2. Restore the database schema to the database

```
Sqlcmd -U <user_name> -i
<file_location>/database_schema.sql
```

1.1.2 *Run the docker container with Risk Management tool*

1. Download the DockerFile:

```
wget -L https://gitlab.com/enact/risk-
management/raw/master/Dockerfile
```

2. In the repository where you downloaded the DockerFile, build your image:

```
docker -t risk-management build .
```

3. Run the docker container (Depending on how you plan to use Risk Management tool, remember to open the proper ports, and pass the database connection string cf. https://docs.docker.com/engine/reference/run/). Please note that the database server should be available to docker image.

```
docker run --network=”host” -e DbConnection=”
Server=sqlserver;Database=xxxx;User Id=xx;Password=xxx;”
-p 8080:8080 risk-management
```

4. Access the Risk Management web interface

Once Risk Management tool has started, you can access the web interface at the following address:

```
http://localhost:8080
```

2 GeneSIS – User guide

In the following section we summarize the main instructions to install, configure, start and use GeneSIS. More details can be found in the GeneSIS GitLab repository: https://gitlab.com/enact/GeneSIS

2.1 Installation

The main README file at the root of the GeneSIS Gitlab repository details how to setup and start GeneSIS from: *(i)* Git, *(ii)* the GeneSIS official Docker image, and *(iii)* the Docker Build file. In the following we provide an overview of these instructions.

2.1.1 *Pre-requisite:*

- Node.js v7
- npm v4
- Java v8

In order to deploy docker containers on a host, please remember to turn on the Docker Remote API on the target host. On Raspberry Pi, you can install docker using:

```
curl -sSL https://get.docker.com | sh
```

and configure it as follows:

- Create a file called:

```
/etc/systemd/system/docker.service.d/startup_options.conf
```

- Add to the file:

```
# /etc/systemd/system/docker.service.d/override.conf
[Service]
ExecStart=
```

```
ExecStart=/usr/bin/dockerd -H tcp://0.0.0.0:2376 -H
unix:///var/run/docker.sock
```

- Reload the unit files:

```
sudo systemctl daemon-reload
```

- Restart Dockerd:

```
sudo systemctl restart docker.service
```

2.1.2 *From git:*

If you want to run the latest code from git, here is how to get started:

1. Clone the code:

```
git clone https://gitlab.com/enact/GeneSIS.git
cd GeneSIS
```

2. Install the dependencies:

```
npm install
```

3. Run GeneSIS:

```
npm start
```

4. Access the GeneSIS web interface

Once GeneSIS started, you can access the GeneSIS web interface at the following address:

```
http://your_pi:8880
```

2.1.3 *From DockerFile:*

You may build your own Docker image of GeneSIS by using our DockerFile. This image will run the latest code from git.

5. Download the DockerFile:

```
wget -L
https://gitlab.com/enact/GeneSIS/raw/master/Dockerfiles/D
ockerfile
```

6. In the repository where you downloaded the DockerFile, build your image:

```
docker -t genesis build .
```

7. Run the docker container (Depending on how you plan to use GeneSIS, remember to open the proper ports, cf. https://docs.docker.com/engine/reference/run/).

```
docker run -p 8880:8880 genesis
```

8. Access the GeneSIS web interface

Once GeneSIS started, you can access the GeneSIS web interface at the following address:

```
http://your_pi:8880
```

2.1.4 *From the public Docker image:*

1. Pull the image:

```
docker pull nicolasferry/genesis
```

2. Run the docker container (Depending on how you plan to use GeneSIS, remember to open the proper ports, cf. https://docs.docker.com/engine/reference/run/).7

```
docker run -p 8880:8880 genesis
```

3. Access the GeneSIS web interface

Once GeneSIS started, you can access the GeneSIS web interface at the following address:

```
http://your_pi:8880
```

2.2 Tutorials and examples

- A set of examples of deployment models: https://gitlab.com/enact/GeneSIS/tree/master/docs/examples
- A set of six tutorials gradually explaining how to use GeneSIS is available at https://gitlab.com/enact/GeneSIS/tree/master/docs/tutorial and include instructions for:
  - Deploying a single instance of Node-RED.
  - Deploying a single ThingML Component.
  - Deploying two instances Node-RED.
  - Deploying via Ansible.
  - Deploying via SSH.
  - Deploying multiple nodes including a node requiring a deployment agent.

3 Actuation conflict manager – User guide

In the following section we summarize the main instructions to install, configure, start and use Actuation Conflict Manager. More details can be found in the Actuation Conflict Manager GitLab repository: https://gitlab.com/enact/actuation_conflict_manager

3.1 Installation

The main README file at the root of the Actuation Conflict Manager Gitlab repository details how to setup and start Actuation Conflict Manager from Git. In the following we provide an overview of these instructions.

3.1.1 *Pre-requisite:*

- Node.js(tested with 8.11.2 and 10.15.1)
- Npm (tested with 5.6.0 and 6.4.1)
- Java 8
- rethinkdb https://www.rethinkdb.com/docs/install/

3.1.2 *Installation from git:*

If you want to run the latest code from git, here is how to get started:

1. Clone the code:

```
git clone https://gitlab.com/enact/actuation_conflict_manager.git
cd actuation_conflict_manager
```

2. Install the dependencies:

```
cd acm-app
npm install
cd ../acm-repository-gateway
npm install
```

3.1.3 *Run from git sources:*

1. Start rethink database:

```
./rethinkdb.exe
```

2. Start AGG Gateway server

```
cd acm-app\agg
java -jar agg-gateway-0.1.0.jar -cp agg_V21_classes.jar
```

3. Start actuation conflict manager application:

```
cd acm-app
npm start
```

Once Actuation Conflict Manager started, you can access the Actuation Conflict Manager web interface at the following address:

```
http://localhost:3333
```

3.2 Examples and tutorials:

A set of examples/tutorials to use Actuation Conflict Manager is available at the following address:
https://gitlab.com/enact/actuation_conflict_manager/tree/master/docs/examples

- Using ACM with Conflict with a single Node-RED instance
- Using ACM with Conflict with two Node-RED instances
- Using ACM with with a GeneSiS deployment model

4 Test and simulation – User guide

References

1. Lund, M.S., B. Solhaug, and K. Stølen, *Model-driven risk analysis: the CORAS approach*. 2010: Springer Science & Business Media.
2. 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.
3. Morin, B. and N. Ferry, *Model-based, Platform-independent Logging for Heterogeneous Targets*, in *MODELS*. 2019, IEEE/ACM.
4. 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.
5. 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.
6. 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.
7. Fleurey, F. and B. Morin. *ThingML: A Generative Approach to Engineer Heterogeneous and Distributed Systems*. in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW 2017), Gothenburg, Sweden, 5-7 April 2017*. 2017.
8. Morin, B., et al. *A Generative Middleware for Heterogeneous and Distributed Services*. in *2016 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE)*. 2016.
9. Dearie, A. *Software deployment, past, present and future*. in *Future of Software Engineering, 2007. FOSE'07*. 2007. IEEE.
10. Bergmayr, A., et al., *A Systematic Review of Cloud Modeling Languages.* ACM Computing Surveys (CSUR), 2018. **51**(1): p. 22.
11. Atkinson, C. and T. Kühne, *Rearchitecting the UML infrastructure.* ACM Transactions on Modeling and Computer Simulation (TOMACS), 2002. **12**(4): p. 290-321.
12. Ferry, N., et al. *Towards Meta-adaptation of Dynamic Adaptive Systems with Models@ Runtime*. in *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, Port, Portugal, February 19-21, 2017*. 2017.
13. Blair, G., N. Bencomo, and R.B. France, *Models@ run. time.* Computer, 2009. **42**(10): p. 22-27.
14. O'Leary, N. and D. Conway-Jones, *Node red-a visual tool for wiring the internet of things.* Retrieved July, 2017. **4**: p. 2017.

15. Harrand, N., et al. *Thingml: a language and code generation framework for heterogeneous targets*. in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. 2016. ACM.
16. Schmidt, D.C., *Model-driven engineering.* COMPUTER-IEEE COMPUTER SOCIETY-, 2006. **39**(2): p. 25.
17. Brambilla, M., J. Cabot, and M. Wimmer, *Model-driven software engineering in practice.* Synthesis Lectures on Software Engineering, 2012. **1**(1): p. 1-182.
18. Kleppe, A.G., et al., *MDA explained: the model driven architecture: practice and promise*. 2003: Addison-Wesley Professional.
19. Morin, B., N. Harrand, and F. Fleurey, *Model-based software engineering to tame the IoT jungle.* IEEE Software, 2017. **34**(1): p. 30-36.
20. Eugster, P.T., et al., *The many faces of publish/subscribe.* ACM computing surveys (CSUR), 2003. **35**(2): p. 114-131.
21. Bencomo, N., et al., *Models@ run. time: foundations, applications, and roadmaps*. Vol. 8378. 2014: Springer.
22. Noguero, A., A. Rego, and S. Schuster, *Towards a Smart Applications Development Framework.* Social Media and Publicity. **27**.
23. *Handbook of Model Checking*. 2018, Springer.
24. Chan, W., et al., *Model checking large software specifications.* IEEE Transactions on Software Engineering, 1998. **24**(7): p. 498-520.
25. Ressouche, A., J.-Y. Tigli, and O. Carrillo. *Toward Validated Composition in Component-Based Adaptive Middleware*. 2011. Berlin, Heidelberg: Springer Berlin Heidelberg.
26. Hartig, O., *Foundations of RDF* and SPARQL* : (An Alternative Approach to Statement-Level Metadata in RDF)*, in *AMW 2017 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web, Montevideo, Uruguay, June 7-9, 2017.*, D.S. Juan Reutter, Editor. 2017, Juan Reutter, Divesh Srivastava.
27. Zeigler, B.P., T.G. Kim, and H. Praehofer, *Theory of Modeling and Simulation*. 2000: Academic Press, Inc. 510.