Low-Code Engineering for the Internet of Things

Felicien Ihirwe

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

Department of Information Engineering, Computer Science and Mathematics

Ph.D. Program in Information and Communication Technology
Curriculum: Emerging computing models, software architectures, and intelligent systems

XXXV cycle

This dissertation is submitted for the degree of
Doctor of Philosophy

Low-Code Engineering for the Internet of Things

SSD INF/01

Supervisor
Prof. Davide Di Ruscio

Co-Supervisors
Prof. Alfonso Pierantonio
Dott. Simone Gianfranceschi

Doctoral Program Supervisor
Prof. Vittorio Cortellessa

Doctoral Dissertation of:
Jean Felicien Ihirwe

A.Y. 2021/2022

ACKNOWLEDGMENT

First and foremost, I would like to express my heartfelt gratitude to my supervisor, Prof. Davide Di Ruscio, for his unwavering support from the beginning to the finish of my doctorate studies. Your insights and expertise have been invaluable in shaping my ideas and strengthening my technical, research, and writing skills. I want also to express my most sincere appreciation to my co-supervisors Prof. Alfonso Pierantonio and Simone Gianfranceschi for your effort and the trust you have invested in helping me reach this milestone.

I'd like to express my gratitude to Prof. Andrea Polini and Prof. Gerson Sunyé for carefully reading the preliminary version of this thesis and providing helpful comments and recommendations. I also thank the entire Lowcomote supervisory board for their feedback and constructive criticism, which has helped me to reshape my thoughts. I recognize and appreciate the whole Intecs Solution community for providing me with a friendly and supportive atmosphere, especially Silvia Mazzini for his continuous support even after she retired. I would also want to thank Prof. Eric Umuhoza for his guidance and encouragement even before the beginning of this adventure till the end, all of my colleagues and friends from the Lowcomote community, the DISIM department colleagues at the University of L'Aquila, and my friends.

I'd like to extend my sincere tribute to my mother for her continuous encouragement and counsel toward my greater success. Finally, my special thanks goes to my girlfriend, entire family, especially my siblings, who have believed in me and always encouraged me to go for it.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 813884

Less expectation, more exploration
and trust.

Abstract

The Internet of Things (IoT) technologies are often seen as the main drivers of the current technological revolution, which devotes the most priority to improving the well-being of humanity. IoT is typically regarded as a powerful network of systems that integrates several heterogeneous and independently networked devices working together to achieve a shared purpose. Engineering such systems requires efficient tools to deal with the intrinsic complexities while offering means to increase system reliability by limiting future repair costs. Low-Code Development Platforms (LCDPs) unravel the opportunities to advance the simplicity of how new applications are developed in different business application domains. However, in the IoT domain, systems are complex, multi-layered, and highly heterogeneous in all aspects, not to mention the large amount of data being collected and processed concurrently. Even though there is a convenient push toward coping with such complexities, there still needs to be a massive gap regarding the actual development techniques that support early system analysis, deployment, and run-time management. Low-Code Engineering (LCE), on the other hand, aims to tackle such issues by extending the development knowledge present in LCDPs to a more sophisticated era of "Low-Code Engineering Platforms (LCEPs)" by injecting into it the theoretical and technical concepts present in Model-Driven Engineering (MDE), Cloud Computing, and Machine Learning. These platforms target more sophisticated domains such as IoT, industrial automation, data science, recommender systems, etc. This dissertation addresses such challenges by first presenting the current state of the art of Low-Code Engineering Platforms (LCEPs), which gives a better understanding of what LCEPs are and their differences with respect to existing LCDPs, particularly in the IoT domain. We also highlight how MDE plays a significant role in the LCE's evolution. Then, we examine the current limitations, open challenges, and opportunities of existing IoT Engineering platforms in realizing such an initiative. While evaluating the quality of such complex platforms could be challenging, we propose the software product quality model for evaluating the static and dynamic quality properties of such engineering platforms.

The complexity behind the automated realization of IoT systems can be extremely daunting. One efficient approach is to adopt Domain-Specific Languages (DSLs). DSLs are tailored to the specific domain to pave the way for the domain experts to define the system's behavior based on their expertise. This dissertation presents CHESSIoT, a platform that integrates high-level visual DSLs, software development, safety analysis, and deployment mechanisms for engineering multi-layered IoT systems. With CHESSIoT, users may conduct various engineering tasks on system and software models to enable earlier decision-making. This is achieved in a unique environment that combines multi-staged designs, most notably the system-level, functional, and deployment architectures. The physical architecture specifically contains the high-level system building blocks and their interconnections suitable to perform both early qualitative and quantitative safety analysis by employing logical Fault-Trees (FTs). On the other hand, the software model is equipped with the system's functional behavior suitable for generating platform-specific code ready to be deployed on low-level IoT device nodes. Additionally, the framework supports modeling of the system's deployment, which would ultimately be used to generate deployment artifacts. To facilitate run-time management of deployed services, the tool offers means for defining run-time service provisioning modules through which deployment rules are defined and configured. To demonstrate the effectiveness of our proposed approach, throughout this dissertation, different comparative assessment was conducted to highlight the potential contribution of our approach in relation to existing approaches. Finally, we used the implications from the conducted research studies as well as experiments from running examples to tackle potential research questions as well as demonstrate the capabilities of our supporting tool.

Keywords: *Low-Code Developments Platform, Low-Code Engineering Platform, Model-Driven Engineering, Internet of Things, System modeling, Safety analysis, Code generation, System Deployment*.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The current advancement in software engineering is transforming how we live our daily lives by providing a smoother living experience by automating essential lifestyle tasks. IoT technologies are one of the critical drivers of such a technological revolution which devotes the most priority to enhancing the well-being of humanity. However, because of the competitive challenges imposed by digitalization, [6], businesses are always looking for innovative approaches that can tackle complex problems and reduce time to market, as well as the costs associated with development while maintaining the optimum software quality.

A typical IoT system is regarded as a powerful multi-layered network of systems that integrates many heterogeneous, independently networked systems working together to achieve a shared purpose. In general, a typical IoT system consists of multiple layers. For instance, at a low level, we have the edge layer, which is made up of devices and sensors that collect data from the physical world and communicate it to the next layer. The fog layer serves as an intermediate layer between the edge devices and the cloud, enabling local data processing and decision-making. It can perform various tasks, including data filtering, aggregation, analytics, and even running certain applications locally. Finally, the cloud layer is a centralized repository where data from all devices is stored and analyzed. This layer can also include services such as data storage, analysis, and management. Such systems demand various development skills, from handling tiny microcontrollers to more extensive and complex cloud-based systems. Each layer is crucial to enable the system to operate efficiently and offer users valuable insights and automation capabilities [7].

Engineering such systems is challenging and complex primarily due to the ever-increasing heterogeneity in every aspect that needs to be combined to fully produce a well-sensed and functional system [8]. Model-Driven Engineering (MDE) seeks to support the automation of the software development process by employing models as the primary artifact in the development of complex systems. Through high-level abstractions, MDE can provide a unique means for representing many aspects of heterogeneous systems all in one place thanks to modeling languages, specifically Domain-Specific Modeling Languages (DSMLs). Tackling such heterogeneity is essential to look at every system sub-component as a black box, where both the physical characteristics and the software that manages them are highly linked [9]. These sub-systems can be designed, developed, tested, and analyzed independently, and later they can be integrated to form a fully functioning system.

Cloud-based modeling is one of the relevant topics in the MDE community due to the induced possibilities of designing, developing, analyzing, and deploying applications seemingly with reduced efforts. This has also been recently favored by the increasing adoption of Low-Code Development Platforms (LCDPs). LCDPs aim to tackle the shortage of highly skilled professional software developers by enabling end-users with no or limited programming background (referred to as citizen developers in LCDP terminology) to contribute to software development processes without sacrificing the productivity of professional developers [10]. LCDPs have been especially successful in developing domain-specific applications in four market segments: database applications, mobile applications,

process applications, and request-handling applications [11]. IoT is expected to be the fifth one . Low-comote project [11] aims to push advancement in LCDP to a more technical and sophisticated era of "Low-Code Engineering (LCE)" by employing the concrete basic engineering principles in the modeling world. The merge of MDE, Cloud computing, and Machine Learning techniques will do this. Ideally, domain-specific LCDPs have to run on cloud infrastructures, even though in some industrial settings such as IoT, domain-specific modeling environment tends to be local-based [12].

This dissertation presents the current state of research on MDE approaches for IoT by taking a particular account of LCDPs. We present the results from a taxonomy and the findings that have been done by analyzing sixteen IoT development platforms. In addition, we looked at what has been done so far in the IoT domain to support IoT systems' development through cloud-based settings. In particular, we conducted a thorough investigation to see where the IoT community stands concerning the current trend of moving traditional modeling infrastructures to the cloud. After examining 611 articles, we identified 22 different cloud-based IoT system development tools and platforms. Furthermore, we perform an analysis of the various issues that the IoT community is encountering while implementing cloud-based modeling tools. As a result, we take a deeper look at a few options and discuss the research and development opportunities enabled by adopting LCE approaches in the IoT domain [13].

Deciding whether such engineering platforms meet the minimum required software quality standards is complex. Software quality can be defined as the degree to which a software system achieves its intended goal. Various software quality standards have been established to aid in the software quality assessment process; however, due to the nature of engineering IoT platforms, such models may only partially suit the IoT domain. This dissertation presents a model for assessing the software quality of Low-Code and MDE platforms for engineering IoT platforms. The proposed software quality model is based on and extends the ISO/IEC 25010:2011 [14] software product quality model standard. It is intended to assist IoT practitioners in assessing and establishing quality requirements for engineering IoT platforms. We have also presented the methodology used to choose such platforms and perform the quality assessment while subsequently presenting and discussing the results.

MDE tools have proven potential to provide significant benefits in the development of complex systems [15, 16]. However, as the complexity of the systems grows, MDE tools will need to handle large-scale models, which can be computationally expensive and may lead to scalability issues. Considering the heterogeneity in all aspects of IoT systems, MDE tools need to provide a way to integrate these diverse components and ensure interoperability [8]. Additionally, the dynamic nature of IoT systems, where devices can join or leave the network at any time, poses another challenge for MDE tools. The models need to be adaptive to changes and allow for the efficient reconfiguration of the system [9]. Furthermore, deploying IoT systems can be challenging, as they often operate in highly distributed and complex environments. Therefore, MDE tools must provide mechanisms for deploying and managing the system components in a scalable and efficient way [17]. This includes automatic deployment of software updates, efficient management of resources, and monitoring of the system's performance.

1.1 Challenges and motivation

While engineering IoT systems has been subject of intense research, several challenges are still present in the IoT domain. In this section, we highlight different research problems (RP) that is being tackled in this thesis.

1. ***RP1:*** **Low-Code Engineering Platforms and their usage in engineering IoT systems**

   While the complexity of implementing IoT systems is enormous in all aspects, the current technical demand provides significant obstacles to better software development techniques that reduce developer issues. The recent Low-code practices for developing software remain a highly debated topic regarding the degree to which such approaches can be used and the level at which they can satisfy user expectations in developing complex software in fields such as IoT. With the

rapid increase of intelligence on how traditional code-centric software development is done in domains such as IoT, enabling Low-code approaches applied in such domains could contribute toward better software production yield. Understanding what Low-Code Engineering Platforms are and their difference with respect to existing Low-Code Development Platforms remains a work in progress within the software engineering industry. Taking LCDPs from a systems engineering standpoint and industrial automation provides a good picture of what LCEP could be and achieve if made a reality.

2. ***RP2:*** **Evaluating the software quality of Low-Code Engineering Platforms**

   Over the last few years, industry and academia have proposed different LCDPs to ease the development process of IoT systems. However, deciding whether those platforms meet the crucial software quality standards is a complex process as it involves considering and exploring various aspects. In general, in domains such as System engineering, Space, and Automation, practitioners typically rely on well-established standards and practices to improve confidence in whether a system or a product fulfils the required quality requirements. For example, in the past, the ISO/IEC 25010:2011 standard has been adopted to assess not only the product quality of IoT systems [18–20] but also in domains such as Big data [21], Machine Learning [22], Software Product Lines (SPL) [23], Customer Relationship Management (CRM) systems [24] and mobile apps [25], to mention a few. However, when it comes to IoT in general, taking Low-code and MDE tools in particular, there still needs to be a massive gap in what to consider when evaluating their software product quality attributes.

3. ***RP3:*** **Supporting modeling of multi-layered IoT systems**

   There are several challenges in modeling multi-layered IoT systems. One of the main challenges is the system's complexity, which can involve many interconnected components, each with its own set of characteristics and behaviors [26]. This can make it challenging to develop a comprehensive model that captures all of the relevant factors that influence the functional and behavioral aspects of the system. In addition, the dynamic nature of the IoT system, with devices and applications constantly changing and evolving, further complicates the process. Another ongoing challenge is the need to integrate diverse technologies and standards across different system layers [27]. For example, the device layer may use different communication protocols than the fog layer; on the cloud, the application layer may require more data formats than the computing layer. Modeling these different technologies and standards seamlessly and efficiently can be difficult and may require specialized knowledge and expertise in various areas, including networking, software development, and data analytics. Few of the existing model-driven approaches have tackled such issues however, being able to achieve such a modeling task and at the same time offering means to perform other crucial engineering tasks on the model is still an open issue.

4. ***RP4:*** **Providing means for performing safety analysis of IoT systems**

   A significant challenge to be recognized in IoT ecosystems is how to provide a reliable infrastructure for the billions of expected devices and how to deliver their intended services without failing in unexpected and catastrophic ways [28]. Aside from the inherent difficulties in realizing multi-layered IoT applications systems, software developers often make the false assumptions that devices will always succeed [29]. Indeed, IoT systems might fail for a wide range of reasons: device age, data sources, communication protocols, deployment environments, and human errors. In the past, safety engineers relied on different informal design artifacts and documents to measure the safety compliance of the system with less or no involvement of system engineers. Later, several approaches, such as [4, 30–33] (to mention a few), have emerged in the field by providing a tool that adds a degree of automation during the analysis process, bridging the gap between the system and safety engineers. However, these approaches were designed and developed to fit mostly the legacy domains such as aerospace, automotive, and

industrial manufacturing systems. Therefore, such methods might partially reflect IoT. Thus, providing foundational concepts and approaches to support the IoT safety process to cope with the complex nature present in IoT ecosystems can potentially contribute to tackling such gap [34].

5. ***RP5:*** **Supporting the development and deployment of IoT systems across multiple layers**

   The development and deployment of IoT systems require a multi-disciplinary approach that considers the system's hardware, software, and communication aspects. By carefully considering each layer and selecting the appropriate technologies and tools, developers can build robust, scalable, and secure IoT systems that meet the needs of today's businesses and consumers. MDE has shown capabilities to tame some of the complex problems found in software engineering through abstraction. To increase productivity and reduce time to market, models are defined with concepts that are much less bound to their underlying implementation technology and much closer to the problem domain of interest [35]. However, due to the inherent complexity and heterogeneity present in the IoT domain, engineering platforms such as MDE4IoT [36], ThingML [37], IoTML/BRAIN-IoT [38, 39], SimulateIoT [40] and Montithings [17] (to name a few), have demonstrated the potential to be realistic alternatives for developing scalable IoT systems leveraging MDE approaches. While that is the case, finding a platform capable of fully engineering such systems by integrating modeling, software development, system analysis, and deployment becomes challenging.

To tackle the challenges mentioned above, we present the CHESSIoT framework, an environment for engineering IoT systems. CHESSIoT brings a unique possibility to the user to perform the modeling, development, safety analysis, and deployment of multi-layered IoT systems, all from a unique environment. This is achieved through multi-view models, most notably the physical, functional, and deployment architectures. The physical system-level model is annotated with Failure Logic behaviors in which the analysis results are used to perform both qualitative and quantitative safety analysis by employing logical Fault-Trees models (FTs) [41] On the other hand, the software model is equipped with the system's functional and behavioral aspects, and it is employed to generate platform-specific code that can be deployed on low-level IoT device nodes. Furthermore, the framework supports modeling of the system's deployment plan, which is ultimately transformed into deployment configuration artifacts ready to be deployed on remote servers. To facilitate deployment, the tool offers means to design run-time service provisioning modules through deployment agents, which are then used to configure and remotely manage the run-time life-cycle of deployed services.

Throughout this doctoral thesis, we exploit various assessment techniques, such as, but not limited to, addressing potential research questions relative to the specific topic of interest. In addition, we rely on implications from comparative analyses to evaluate the effectiveness of our proposed approach and our contribution relative to the existing approaches. To demonstrate the supporting tool's capabilities in satisfying specific developer requirements, different running demonstrations were implemented.

1.2 Main achieved research and technological results

Employing Low-Code Engineering for engineering IoT systems

Low-Code Engineering (LCE), as a relatively new concept in the MDE industry, has sparked much controversy and misunderstanding about what it is and how it differs from the previous Low-code development methodologies. Furthermore, there is still discussion about what both novel approaches add or change compared to existing MDE techniques. Although in this dissertation we concentrate on the IoT domain, we would therefore discuss their similarities, interdependence, and differences in the procedure for software engineering. As an answer to the first research problem *(RP1)*, this dissertation we highlights the current state of the art regarding how IoT developers are incorporating low-code methodologies into their development process. For instance, as presented in Chapter 4, in [8], we examined 16 different platforms to gain a better understanding of the current state of supporting the

development of IoT systems, with a focus on languages and tools available in the MDE field and emerging LCDPs. In [13], we examined 22 low-code environments by assessing their strengths and weaknesses regarding cloud-based modeling capacity, accessibility, openness, and artifact generation. This work was published both in [13] and [8].

Software product model for evaluating the software product quality of LCEPs in IoT domain
As the IoT LCE tools go to market, their quality to satisfy the user requirements is always questionable since there are not yet established mechanisms for assessing their software product qualities. To answer the second research problem *(RP2)*, this dissertation goes a step further by proposing a software product quality model intended to help people interested in developing or purchasing low-code software products and systems specify and evaluate their quality requirements. In doing so, it presents a quality model based on the ISO/IEC 25010:2011 standard [14] with the enhanced definition of a product quality assessment model that is more suitable for the IoT domain. In the end, the evaluation mechanism of the proposed quality was done by employing it to assess the software product quality of 17 IoT low-code and MDE platforms selected from our previous studies [8, 13]. Finally, we present the methodology we used to choose such platforms, perform the quality assessment, and subsequently present and discuss the obtained results. This work was published in [27].

Multi-view modeling environment for IoT systems
In this dissertation, we introduce the CHESSIoT modeling environment and languages for multi-layered IoT systems as an answer to the third research problem *(RP3)*. To tackle the scalability and complexity of modeling IoT systems, in CHESSIoT different aspects of the system are modeled independently from their respective views and later interlinked to satisfy specific engineering tasks being performed on the model. To achieve that, the designer relies on a series of CHESSIoT DSLs in which the meta-modeling syntax has been specified as an extension to both UML/SysML modeling languages. CHESSIoT DSLs are made up of three primary DSLs (abstract syntaxes), namely *SystemDSL* for IoT system-level modeling, *SoftwareDSL* for functional and behavioral modeling, and the *DeploymentDSL* for deployment-related aspect modeling and runtime service provisioning. To guarantee a fully decoupled extension, CHESSIoT introduced the *"IoT sub-view"* constraint and once applied in all design stages, the user will benefit from a dedicated IoT-specific modeling infrastructure consisting of specific diagrams and palettes. This also enforces correctness and error avoidance during the design phase, as palette elements can be hidden or shown based on the current state of the modeling process (e.g., diagram type or view type).

Model-based safety analysis of safety critical IoT systems
As an answer to the fourth research problem *(RP4)*, this dissertation introduce the CHESSIoT model-driven safety analysis approach targeting IoT systems based on the Fault-Tree Analysis (FTA) approach. The approach runs on top of CHESSIoT, a model-driven development environment for the modeling and the analysis of industrial IoT systems [42, 43]. The presented approach relies on CHESS Failure Logic Analysis (CHESS-FLA) [44], a methodology that enables the user to: *i)* model system's failure behavior of the system through the decoration of the system model components with safety-related information, *ii)* run the Failure Logic Analysis (FLA), *iii)* and propagate the analysis results back onto the original model [45]. The new approach allows the specification of systems failure modes and generates the system's complete Fault-tree based on the failure logic analysis results. In addition, the new approach automatically performs qualitative analyses, which analyze the generated fault trees and eliminates unnecessary paths and redundancies in the FTs' events. Finally, in addition to the qualitative analysis, the proposed approach also calculates the failure probabilities of an entire system from its constituent parts' failure event probabilities. This calculation is automatically performed following the well-known logic Fault-tree probabilities calculations mechanism [46–48].

Model-based development and deployment support for IoT systems

As an answer to the fith research problem *(RP5)*, this dissertation also present the CHESSIoT software design approach that combines both the functional and behavioral modeling aspects targeting the IoT device layer. The functional design involves the systematic definition of the main software components, their sub-components, and their interconnection. Each system's main sub-function is entitled to its state machine, in which aspects such as message payloads, events, actions, and guards are associated with states and their transitions to realize the desired behavioral goal. When the model is complete, the CHESSIoT model is transformed into ThingML models [37], which eventually is used to generate platform-specific code. ThingML is a well-proven software modeling tool aligned with UML (state charts and components) and an imperative platform-independent action language to construct the intended IoT applications [8]. Although the ThingML model can compile and generate code in different languages such as Arduino, C/C++, Java, JavaScript, and Go, the implemented code generator that is supported by our tool only satisfies the CHESSIoT models targeting Arduino-based computing devices. In addition, CHESSIoT provides a deployment environment that aims to support the users in decomposing IoT system deployment plans and managing deployed node services at all layers. The deployment model connects the software to the actual system nodes in which the software program will be executed. Finally, CHESSIoT deployment offers a model-driven approach for runtime service provisioning that allows the definition of runtime software services provisioning and life-cycle management. The provisioning model is defined in the form of deployment rules referred to as agents.

1.3 Structure of the dissertation

The rest of the dissertation is organized as follows: **Chapter 2** provides the general background of the fundamental concepts and principles such as IoT as a multi-layered ecosystem, MDE and low-code around which this dissertation is based on. **Chapter 3** summarizes state of the art on IoT engineering platforms related to several topics discussed in this dissertation. **Chapter 4** presents the limitations and open challenges of existing IoT Engineering platforms by relying on experimental studies on focusing their general features and what such platforms should be supporting. **Chapter 5** presents a model for assessing the software product quality of Low-Code and MDE engineering platforms for IoT development platforms. **Chapter 6** introduces the CHESSIoT engineering approach and a motivating comparative analysis as an evaluation mechanism. The chapter also presents CHESSIoT DSL in great detail and compares its contribution with respect to the existing DSLs. **Chapter 7** presents CHESSIoT's two-fold safety analysis approach for IoT systems based on Fault Trees models. **Chapter 8** presents the supported development and deployment approach by showcasing the supported feature using a running example. Finally, **Chapter 9** concludes the dissertation and highlights the future outlooks.

Chapter 2

Background

This chapter provides a general overview of the essential concepts and principles around which this dissertation is founded. By partially targeting to answer the first research problem (RP1), this chapter discusses the notions of IoT systems in the context of multi-layered architecture, Model-Driven Engineering (MDE) concepts, and how they relate in handling IoT domain complexity. The chapter also covers Low-Code Engineering (LCE) concepts in general, as well as where Low-code and MDE approaches fit in such a context. Finally, it provides an overview of the CHESS environment, on which the proposed engineering platform is based.

The chapter is organized as follows: Section 2.1 provides a high-level overview of the IoT as a multi-layered ecosystem. Section 2.2 provides a broad introduction to MDE, Domain Specific Languages, and IoT reference architectures. Section 2.3 introduces Low-Code Engineering Platforms(LCEPs) in general and how they differ from Low-Code Development Platforms (LCDPs). Finally, Section 2.4 introduces CHESS platforms by emphasizing the supported model-based engineering aspects as well as different supported analysis techniques.

2.1 The Internet of Things

IoT is a term used to refer to the interconnection of devices over the internet. This interconnection enables the collection and exchange of information, as well as the management and automation of devices via a network. IoT offers a wide range of applications, including home automation and security, as well as healthcare, transportation, and agriculture [8]. IoT can enable smart devices such as refrigerators, lights, and appliances to be controlled remotely from a smartphone or other device in the household. IoT in healthcare can enable remote monitoring of patients and provide crucial data to physicians and caregivers [5]. IoT in transportation can improve safety by giving real-time traffic data and enabling autonomous driving systems [36].

2.1.1 IoT system architecture

In the past, IoT was referred to as the emergence of barcodes and Radio Frequency Identification (RFID), which served to automate inventory, tracking, and basic identifying needs. However, nowadays IoT is a keen interest in interconnecting sensors, objects, devices, data, and applications [49]. A fully IoT system is often complex, with numerous players with varying levels of skill and multiple stakeholders with varying roles. According to Costa et al [50], IoT is defined as a collection of automated procedures and data that are integrated with heterogeneous entities (hardware, software, and humans) that interact with one other and with their environment to accomplish a common goal. Advances in technology, such as cloud computing, machine learning, and artificial intelligence, have accelerated the emergence of IoT. The potential for IoT to revolutionize businesses and enhance our

lifestyles grows as even more items become interconnected. Yet, there are worries regarding IoT's security and privacy risks. There is a danger of information being hijacked or exploited because of all the devices monitoring and transmitting information [49].

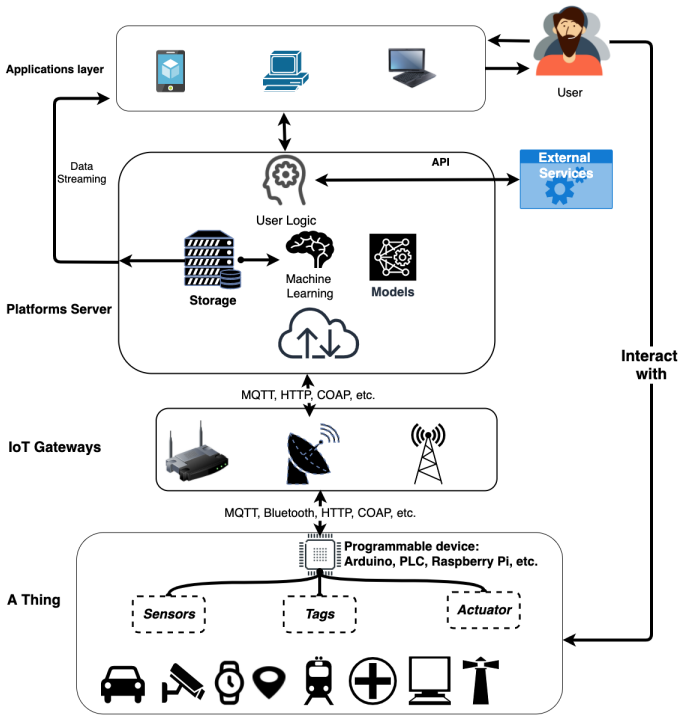


Figure 2.1: IoT system building blocks

Figure 2.1 shows a high-level architecture of a typical IoT system. *A thing* is a combination of on-board devices including *sensors*, *tags*, *actuators*, and *physical entities* like cars, watches, etc. Data is generated from a sensor or a tag attached to the physical entity the user is interested in. A programmable device (such as an Arduino, a Pycom, a Raspberry Pi, etc.) collects data and sends them to the nearby gateway using some well-known protocols such as Z-Wave, MQTT, HTTP, Bluetooth, Wi-Fi, Zigbee, etc. The *Gateway* component acts as a bridge between the physical and digital worlds. Note that in some cases devices and gateways can make some simple computation logic and respond to some events without the need for further processing. The platform server is a combination of processing and storage resources on the cloud. At this stage, data can be streamed, analyzed, or manipulated for meaningful information to be communicated back to actuators, users, or third parties services.

Aside from the inherent difficulties in developing multi-device IoT applications for diverse platforms, software developers often make false assumptions. One of these assumptions is that devices will never fail [29]. Indeed, IoT systems might fail because of a wide range of reasons: device age, data sources, communication protocols, deployment environment, as well as external environment constraints, such as human error. In IoT ecosystems different types of error can occur: local errors, which can be also detected from the device itself, like a failing sensor; or more complex errors that affect multiple devices at the same time, for example, a network failure or a missing communication pattern as a result of a device failure that causes the entire system to fail [17].

An important challenge to be recognized in the IoT ecosystem is how to provide a reliable infrastructure for the billions of expected devices and how to deliver their intended services without failing in unexpected and catastrophic ways [28]. In nature, a system is considered to be fail-safe if it has none or harmless failures, whereas a safety-critical system can have catastrophic failures that can sometimes result in human life loss. In the healthcare domain, for instance, the monitoring of hospitalized patients must be done with extreme caution as a simple failure, such as a false sensor data

reading, can have catastrophic consequences, including the patient's death. Because these systems are at the intersection of information technology and biomedical sciences, it is necessary to have a thorough understanding of how the connected components work as well as the ability to take perfect decisions either manually or through automated software. These systems are among the riskiest in terms of engineering because they interact directly with sick patients.

IoT things, being reactive systems, constantly interact with their surroundings, changing their states. Such entity behavior causes the IoT application extremely dynamic and thus susceptible to unanticipated behavior. Identifying unexpected behavior while also ensuring that essential functionality is in place can be difficult, especially in a dynamic system [51]. This can potentially be achieved by promoting advanced automated software engineering approaches and tools by which can software development could be done in a faster and more secure manner. Tools able to develop, deploy, and analyze the system's reliability to avoid future repair costs could be of huge impact. There are several successful platforms offered by some of the bigger industrial partners to tackle such challenges but looking at the complexity in terms of usability makes it even more challenging.

2.1.2 Safety Critical Systems

The term "safety-critical system" was created in response to growing concern and awareness about the use of computers in situations where human lives could be jeopardized if an error occurs [52]. Safety-critical systems are those systems whose failure could result in loss of life, significant property damage, or damage to the environment [53]. A safety-critical system should be ideally designed to lose no more than one life per billion hours of operation [54]. We can see many examples of such systems in the following domains: aviation, railway, medicine, nuclear engineering, and military. Developing such systems is difficult and must be done with extreme caution, as even the smallest error in the process can have disastrous consequences. More and more modern safety-critical systems are incorporating new technologies, such as machine learning techniques, to reduce the possibility of failure through intelligent responses provided by artificially trained robots [55].

A significant challenge recognized in the IoT ecosystem is how to provide a reliable infrastructure for the billions of expected devices and how to deliver their intended services without failing in unexpected and catastrophic ways [28]. In the IoT context, safety is often considered as the ability to detect and prevent any unintended failure behavior in IoT systems [34]. In the past, IoT systems were considered fail-safe because of their size, as their failures mostly had no or harmless consequences. However, due to the system's rise in size and complexity and the increased demand for IoT systems in the industry, errors and failures for such systems are unavoidable. For instance, IoT systems, such as Intelligent traffic lights, smart homes, smart manufacturing systems as well as patient monitoring systems, can suffer from potential failures generated internally in the system due to several issues such as age or poorly connected or failures caused by external influences such as weather or human error.

As research in this area continues, their developers deem existing proposed concepts and architectures safe. Still, they are frequently found to be impractical for real-life applications because safety-critical systems involve unpredictable behavior of lives, properties, or the environment [56]. In addition, as the technologies evolve in some domains, such as IoT, new failure modes, such as denial-of-service attacks against networked information systems, are emerging. Failures occur through physical effects and service disruption or data loss. The lack of a systematic, disciplined, and quantifiable software engineering methodology, as well as a comprehensive abstraction mechanism for dealing with the increasing complexity of safety-critical systems, results in a wide variety of similar, but not congruent, isolated solutions that cannot be easily reused and combined [34].

The number of computer systems that we consider safety-critical is expected to grow significantly in the future. In addition, the declining cost of hardware, improvements in hardware quality, and other technological advancements ensure that new applications will be sought in a wide range of domains [53]. However, for the analysis of the safety-critical systems, there is no universally accepted rigorous dependability analysis process, which helps in choosing the metrics, techniques, and methodologies for the dependability evaluation of such critical systems [57]. In any case, analysis of software devel-

opment approaches, as well as safety-critical software, is required to determine the most appropriate techniques for use in the production of future software for high-integrity systems [52].

2.2 Model-Driven Engineering

Model-driven engineering (MDE) is a software development methodology that aims at supporting software development and analysis by promoting the adoption of models as first-class citizens [10]. To advance the software development paradigm to a new level, MDE promotes software development through abstraction which significantly reduces the system complexity as well as the development time. To increase productivity and reduce time to market, models are defined with concepts that are much less bound to their underlying implementation technology and which are much closer to the problem domain of interest [35]. In MDE, models are used to specify, test, simulate, verify, modernize, maintain, understand, and generate code for the system, among many other activities [58].

Models in the context of MDE are not sketches or drawings that serve purpose only in design, but they prevail until the end of the development cycle of these systems as machine-readable and processable abstractions [59]. MDE favors the collaboration of engineers and stakeholders, as both work together toward the completion of the conceived products and foster integration of different engineering processes [60]. MDE can also aid to improve the quality of software systems, which is another advantage. Software engineers can reason about the system and make sure it complies with requirements more simply by using models to represent the system. Models are frequently easier to grasp and maintain since they are typically written in a higher-level language than general-purpose programming languages.

In the IoT domain, MDE can well be looked at as the methodology in which the development process focuses only on defining the system's IoT device's behaviors and the data they process, rather than on the platform that runs them. Generally, MDE enforces (i) the system model specification, in which the heterogeneous elements are precisely identified; (ii) promotes the reuse of system elements across teams and other applications (iii) tackles the application's complexity; and (iv) facilitates the communication between the system stakeholders [61]. Different complex engineering activities such as verification and validation as well as different analysis activities can be done on the model to quantify the robustness of the system under development as well as identify and correct any potential future failures that might arise.

The MDE process involves multiple stages, such as model development, model transformation, and code generation. A model of the system being developed is established by the software engineer during the model development step. The software engineer develops a set of rules for transforming the model into another model or into code during the model transformation process. The model transformation engine applies the transformation rules on the model to create a new model or code snippet, automating this step in most circumstances. During the code generation stage, the software engineer generates executable code using the model. This code can be in a traditional programming language, such as Java or C++, or it can be in a specialized language [37].

MDE has had tremendous success stories in many ways in academia as well as in industrial settings. This can be observed in areas such as model-based systems engineering (MBSE), low-code software development, and informal software modeling [62]. Furthermore, the quantitative results from the conceptual analysis of such systems can provide theoretical support for optimizing system architectures and parameters earlier enough [63]. However, this success has led to an even higher demand for better tools, theories, and general awareness about modeling, its scope, and application [64]. This can be looked at from the domain in which such tools are going to be used, technical implementation requirements, and social as well as evolving technologies.

Recently, MDE itself has faced challenges that have shifted the focus of the development of such complex and heterogeneous systems from local environments to the cloud [65]. Modeling-as-a-Service is gaining momentum as the MDE research community is migrating modeling tools and services to the cloud. This migration is encouraged by several out-of-box benefits in cloud computing,

such as easy discovery and reuse of services and artifacts [10]. It has enabled efficient self-healing mechanisms to detect, diagnose, and countermeasure threats and foster collaboration among stakeholders and engineers [66]. Furthermore, migrating modeling artifacts and services on the cloud can facilitate end-users' easy accessibility, hence supporting sustainable management and disaster recovery of model artifacts and tools [67].

Finally, the model can be used to perform even more advanced engineering tasks such as performing different analyses depending on the problem at hand [16]. For instance, it could be better to evaluate the model performance in a real-time state in order to get a sense of how the actual system generated from the model could behave in the long run [42]. Furthermore, models are often injected with external constraints to be able to verify and validate their behavior robustness for instance when code generation is involved [68]. In the IoT domain, there still presents a significant gap in the validation, verification, and analysis of such systems under development [42]. The main challenge is the scope of the analysis; because the number of IoT devices and applications is already large and is only likely to grow in the future, physical replication and testing of IoT systems are complex (due to scale) [69]. This potentially contributes to the long-standing lack of standardized realistic reference models that can perfectly capture the interactions between sensors, apps, and actuators.

2.2.1 Domain Specific Languages in IoT

Domain-specific languages (DSLs) are languages tailored to a specific application domain to define the domain models. These DSLs are used in MDE to pave the way for domain experts to be able to define the system's behavior based on their expertise [70]. A DSL is suited to the specific domain of the software system being developed; it is often used to express models. The DSL provides a way to represent the system at a higher level of abstraction than traditional programming languages and defines the syntax and semantics of the model. Although DSL-based development is hard and requires both domain knowledge and language development expertise, they offer tremendous benefits such as improved productivity for the developers as well as effective communication with the domain experts [71]. Finally, the models defined by these languages are intended to be far more human-oriented than common code artifacts, which are inherently machine-oriented [36].

In the IoT context, DSLs can be particularly useful for simplifying the process of developing software applications for IoT devices. As the core of IoT processes relies on IoT devices that are typically used for specific purposes, such as monitoring environmental conditions or controlling home automation systems. Each of these applications requires a unique set of functions and data structures, and developing software for each of these applications can be time-consuming and complex. DSLs can help to simplify this process by providing a specialized language that is tailored to the specific needs of the IoT application. This can make it easier for developers to write code that is optimized for the particular requirements of the device or application.

Engineering platforms such as MonitorIoT [26], MDE4IoT [36], IoTML [39] and MontiThings [17] (to name a few), have presented potential DSLs able to realistically tackle the high degree of heterogeneity in their hardware devices, data sources, protocols, deployment levels for developing scalable IoT systems. However, developing IoT code generators that are perfectly capable of handling large models and generates full-functional code is still an open issue. Approaches such as ThingML [37] is one of the top code generators in IoT that targets many popular programming languages such as C/C++, Java, and Javascript, and about ten different target platforms (ranging from tiny 8bit microcontrollers to servers) and ten different communication protocols.

Several research and industrial approaches have shown interest in applying ThingML as their modeling or code generation framework. To mention a few, in [72], ThingML has been used to generate code for CAPS, an architecture-driven modeling framework for the development of IoT Systems. In [73] ThingML has been used to specify the behavior of distributed software components, and later it has been extended with mechanisms to monitor and debug the execution flow of a ThingML program. Finally, CyprIoT tool [74] has relied on and extended the ThingML modeling language to model and control network-based IoT applications. Their tool relies on Rule-Based Policy Language, to control

and supervise the behavior of the modeled network and a code Generator that interpret the model and generates deployable network artifacts.

2.2.2 MDE for IoT reference model

IoT reference architectures provide a useful starting point for designing and implementing IoT systems. They offer a standardized approach to IoT system design and help to ensure interoperability and compatibility between different IoT systems. To support the development of complex IoT systems, several standards and tools have been proposed over the last years [61]. Standards like ISO/IEC/IEEE 15288 [1] have been in use to evaluate the quality, efficiency, and life-cycle of different approaches. In [75] an IoT reference model (ITU-TY.2060) [76] is proposed by an International Telecommunication Union (ITU) and presented with respect to other four reference architectures developed in the context of the IoT-A [49], WSO2 [77], Korean RA [76] and Chinese [66] projects.

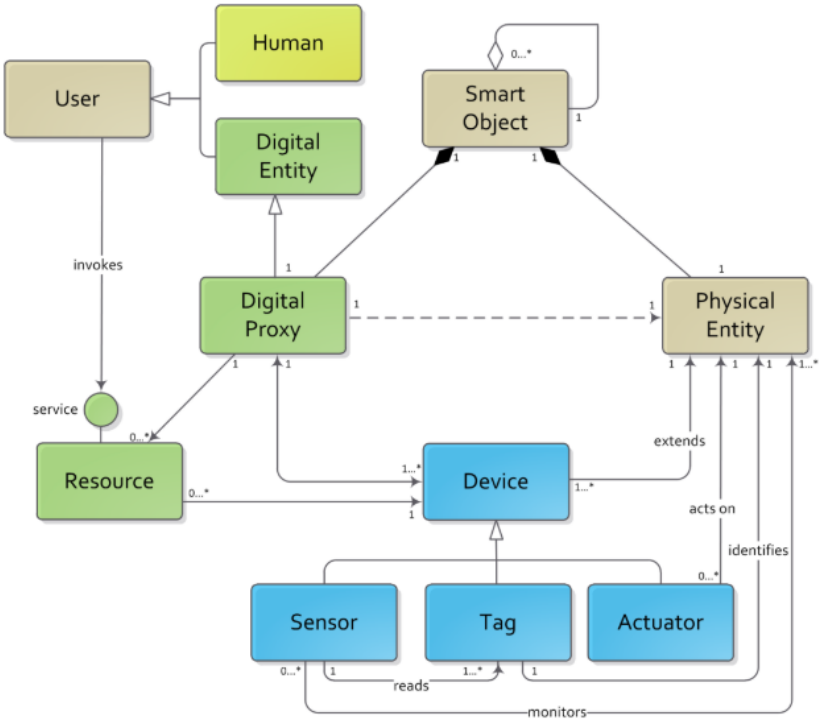


Figure 2.2: IoT conceptual metamodel [1]

Figure 2.2 shows a conceptual representation of the elements shown in Figure 2.1. The physical properties of the associated *Physical entities* are captured through *Sensors*, whereas the modification of physical properties of associated *Physical Entity* is performed through the use of *Actuators*. *Physical Entity* can be represented in the digital world by a *Digital Entity* which is in turn a *Digital Proxy*. *Digital Proxy* has one and only one *ID* that identifies the represented object. The association between the *Digital Proxy* and the *Physical Entity* must be established automatically. A *Smart Object* has the extension of a *Physical Entity* with its associated *Digital Proxy* which then talks to the user by providing or requesting resources. The external services are invoked by the user which can be human or third-party software.

Overall, IoT reference architectures provide a useful starting point for designing and implementing IoT systems. They offer a standardized approach to IoT system design and help to ensure interoperability and compatibility between different IoT systems. However, it has been shown that no single reference model was able to tackle all the aspects that involves in engineering IoT systems [75]. For instance, one of the most used reference architectures in [49] doesn't cover the run-time dynamicity

[1] https://www.iso.org/standard/63711.html

of IoT systems as well as context-awareness concepts. They also lack other aspects such as quantified system reliability, security, and privacy protection. According to different challenges in IoT environments and the existence of some weaknesses in IoT architectures, we believe that more research on IoT reference architectures need to be done.

2.2.3 Model-Based Safety Analysis

Failures that could risk human life, and injuries to the environment, or properties are considered safety hazards. Safety analysis should run concurrently with system design, including interactions between the two, and it should be kept up to date throughout the system life cycle. Risks of this sort are usually managed with the methods and tools of safety engineering. Conducted initially by safety engineers, the safety analysis is one of the dependability analysis techniques that aim to study system response in case of an unwanted failure behavior that can hinder system safety compliance. In safety-critical systems, it is often required to maintain a high level of safety to prevent potentially catastrophic consequences [44]. FTA, as well as FMEA, are already mandatory analysis approaches for performing safety analysis in domains like automotive and aerospace [78, 79], and more domains are going to be subjected to follow that suit [32].

Failure logic approaches map the reliability concepts (produced by reliability engineers) to reflect the underlying fault-to-failure and failure-to-fault propagation within the systems [80]. CHESS Failure Logic Analysis (CHESS-FLA) [44] introduces the possibility of unifying and automatizing existing traditional dependability analysis approaches through the use of the Failure Propagation Transformation Calculus (FPTC) rules [81]. CHESS-FLA enables users (system architects and safety engineers) to decorate component-based architectural models (specified in the CHESS modeling language - CHESSML) with dependability information, perform Failure Logic Analysis (FLA), and have the results back-propagated onto the original model [82]. In practice, a component can act as a source of failure (for example, by causing a failure in output due to the activation of internal faults) or as a sink (a component can avoid failure propagation by detecting and correcting the failure in input). Furthermore, failures in a component can be propagated (i.e., a failure can be passed from input to output) or transformed (by changing the nature of the failure from one type to another from input to output) [3].

The Fault-Tree Analysis (FTA) [41] technique is currently one of the most widely used methodologies when performing safety analysis. The purpose of an FTA is to graphically represent and trace down influence from a system-level hazard to individual failures of distinct system components and sub-components. The graphical representation of the scenarios can aid in explaining these causal chains that can lead to a hazard, followed by an analysis to determine the combination of events that trigger such hazards or compute the chance that such a hazard could occur. During the analysis, the safety engineer starts from the actual hazard, referred to as a "top event," and traces down different event combinations that might contribute to such hazard until the actual cause is reached. This is referred to as a "basic event" in this case. Figure 2.3, shows a typical FT example.

The FT event combination logic relies on logic gates to determine the output of a situation. For example, if two or more events are needed to represent a certain component failure, an "AND" gate is used; while, if one event is enough to trigger the failure, an "OR" gate is used. Other known logic gates can also be used based on the desired system failure behavior.

Failure Mode and Effects Analysis (FMEA) [83] is among the earliest known failure analysis techniques, and it is frequently used as the first stage in a system reliability analysis. Reliability engineers originally developed it to investigate problems that could come from military system failures. It is used to examine as many components, assemblies, and subsystems as feasible to determine failure modes and their causes and effects. The failure modes of each component, as well as their consequences on the rest of the system, are recorded in a separate FMEA worksheet [84]. Unlike the FTA, which follows a top-down deductive approach from the top event to specify its possible causes, the FMEA follows an inductive reasoning approach. Using a forward logic approach, FMEA separates a system into small components, analyses failures that each component may cause, and assesses the effects of

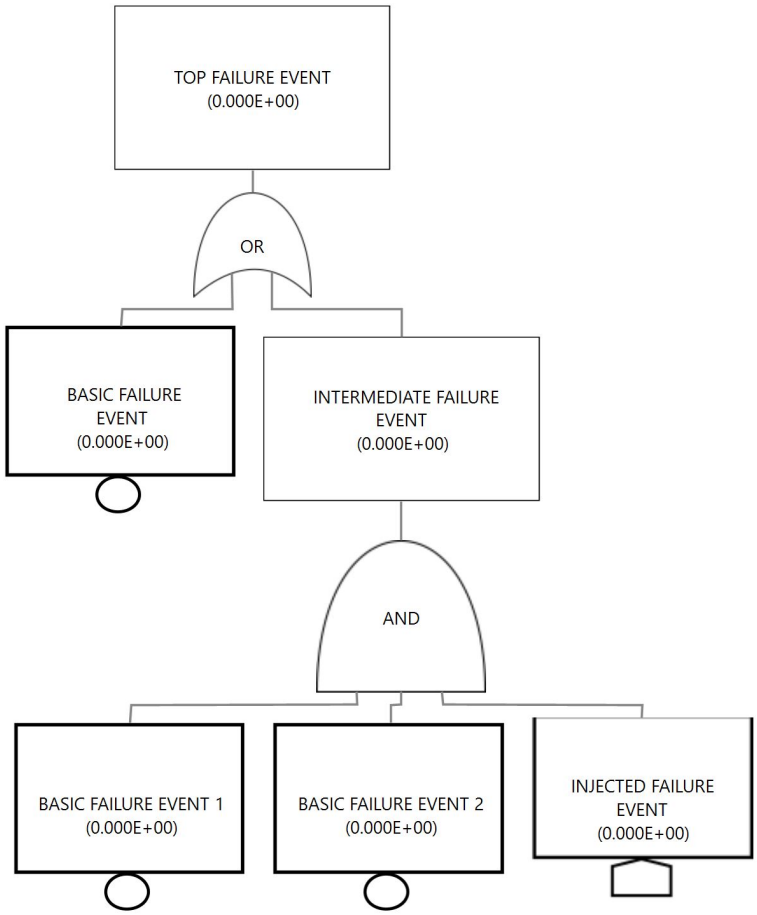


Figure 2.3: An FT example

those failures on the system. As a result, the FMEA performer must properly understand the system safety context and software requirements or design specifics to assure the comprehensiveness of the system decomposition and the validity of each component's usability.

2.3 Low-Code Engineering

The significant advancements in computing power, data storage, and processing are revolutionizing the development and research of complex systems in several domains, including that of the IoT [85]. IoT systems enable the integration of intelligent features into daily human activities through the automation of services. In particular, such systems allow the automation of low-level services that used to be error-prone if done by humans. Moreover, they increase efficacy in current engineering solutions and connect a range of many devices that render our environment smart. Recent reports predict that more than 100 billion devices will be connected by 2025 and 11 trillion dollars of global market capital will be reached [86]. However, to unleash the full potential of these systems, it is necessary that also citizen developers can take part in the development of custom IoT applications [12].

The development and consumption of IoT systems are becoming way more complex, and involving end-users is more challenging due to the heterogeneity of the hardware and required expertise [12]. This complexity originates from various sources. IoT applications are complex systems that use heterogeneous devices and data sources. Besides, IoT systems require enormous efforts and investments both in their implementation and maintenance. Moreover, the systems are implemented using code-centric approaches that make it challenging to foster the inclusion of IoT domain experts and other stakeholders with less IoT programming skills [12].

As a result of the competitive pressures imposed by digitalization [6], more and more industries

across all domains are being pushed to establish software development teams to continue developing and maintaining new products based on customer needs. This allows the organization to keep up to date with customer digital demands while also remaining relevant to market needs. Managing and paying such teams comes at a high budget, which can appear to be a burden to businesses, especially when such teams fail to deliver on time. Reduced time to market as well as the top quality of the developed software are always the main concerns when developing software. Most of the time, software teams reuse existing code to increase the speed at which new features are developed. Although this can always be a better walk around, they are most likely to suffer from code inconsistency, high maintenance as well as incompatibility in case of a very complex software problem. On another hand, businesses are looking for new ways of developing software that can decrease the time to market as well as cut the development cost while maintaining the quality of developed software.

2.3.1 Low-Code Development Platforms

With the increasing interest in advancing the software development process, more and more engineering industries are looking for a way to take advantage of what advanced technologies such as cloud computing and machine learning have to offer. Low-Code Development Platforms (LCDPs) aim at easing the development of fully functional applications by facilitating people with less or no experience in software engineering (eg, business managers) to develop business applications using simple graphical or textual user interface [8]. These platforms drastically reduce the time it takes to build an application, reducing it all from months to days or even hours. Technically, most of such applications are developed through declarative and high-level abstractions languages and take advantage of cloud infrastructures, and automatic code generation to develop entirely functioning applications [87]. LCDPs have shown their strengths in the development of software systems in four main market segments such as database applications, mobile applications, process applications, and request-handling applications. According to Tisi et al. [11], IoT is expected to be the next market segment.

The main goal of MDE is to increase productivity and reduce time to market by enabling the development of systems using models defined with concepts that are much less tied to the underlying implementation technology, and much closer to the problem domain [35]. Same as MDE, low-code software development processes also aim to improve software development processes by raising abstraction and hiding implementation-level details. Both approaches employ model-driven development (MDD) in their development stack; the important distinction seems to be how they enforce factors such as cloud-based deployment, target users, setup, and so on [58]. In addition to that, not all MDE approaches seek to reduce the amount of code required to develop software solutions, and not all low-code approaches are model-driven [58]. Although this is true, their difference in practice is still widely debated on how much work done in MDE is directly transferable to LCDPs [88].

Nowadays, we witness a growing number of successful generic LCDPs on the market (e.g., Google App Maker and Microsoft Power Platform). Despite their success, LCDPs' development capabilities are still limited regarding how complex, intelligent, and sophisticated they may be [11]. Such application often suffers from several issues as follows. On one hand, most of the LCDPs suffer from vendor lock-in problems in which the developed application only being able to be deployed on their own dedicated infrastructure [58]. In addition to that, their scalability in terms of how big and sophisticated such developed applications can be is still questionable. On another hand, domains supported by such platforms are still limited too. For instance, so far LCDPs have been especially successful in the development of domain-specific applications in four market segments such as database applications, mobile applications, process applications, and request-handling applications [11]

Despite their success, LCDPs' development capabilities are still limited regarding how complex, intelligent, and sophisticated they may be [11]. In the IoT domain, only a few LCDPs are available, and they provide limited functionalities given the inherent complexity and heterogeneity of typical IoT systems. Among others, IoT-specific platforms such as Node-RED [89] and Atmosphere IoT [90] have demonstrated a significant push toward the development of fully-fledged multi-layer IoT platforms. However, their capacity to generate code and interact with low-level IoT devices is still

limited. Unlike LCDPs, MDE presents a more significant number of platforms that can still generate low-level platform-specific code. Still, they often suffer from integration and interoperability issues because most of them are deployed and operated locally [8]. Full list of existing LCDPs is presented in Section 3.1

2.3.2 Low-Code Engineering Platforms

With the rapidly rising machine intelligence and how traditional code-centric software development stack has achieved in domains such as IoT, data science, and cloud computing, Low-Code Engineering Platforms (LCEPs) focus on extending the development knowledge present LCDP by injecting it with the theoretical and technical framework defined by recent research in MDE, Cloud Computing, and Machine Learning techniques [11]. These platforms' target span more advanced and complex domains such as IoT, industrial automation, data science, recommender systems, and so on. They aim at overcoming the current limitations present in LCDPs such as those related to scalability (i.e., supporting the development of large-scale applications, and using artifacts coming from a large number of users), open (i.e., based on inter-operable and exchangeable programming models and standards), and heterogeneous (i.e., able to integrate with models coming from different engineering disciplines) [8].

A simple example of differentiating LCEP and LCDPs can be the difference between *Software engineering* and *Software development*. Software engineering is a systematic approach to designing, developing, and maintaining software. It emphasizes on offering software systems of high quality, dependability, and maintainability that satisfy stakeholders. The priority of software engineering is on the use of well-established engineering methods and principles, such as requirement analysis, design, testing, and maintenance [91]. It also takes into account the software's lifecycle, including the phases of development, deployment, and maintenance. Software development, on the other hand, is the process of developing software products through programming, testing, and documentation. It is a component of software engineering that is concerned with developing software solutions that comply with particular business needs. Software development often takes a more realistic and practical approach than software engineering, when implementing software solutions into reality to address particular business or user demands [91]. Even though the ideal LCE target appears to be in its early stages, the current and continuous development under this umbrella shows promising signs of success in the near future.

Although we clearly agree with Di Ruscio et al [58] on the fact that "not all MDE approaches seek to reduce the amount of code required to develop software solutions, and not all low-code approaches are model-driven"; we believe that *"LCEPs should one way or another enforce model-driven principles"*. This is due to the fact that LCEPs combine the MDE, cloud computing and machine learning technologies to tackle most challenging tasks in complex domains. For instance, looking at the system engineering domain, having legacy model-driven services exposed through APIs could potentially boost their usability. The fact that most of the LCDPs are cloud-based and do not require installation significantly lowers the entry barrier for new users [58]. Having such LCDPs consume live services could potentially contribute to their extensibility as well as targeting more complex domains. In the following section, we will go through some of the popular and ongoing potential LCEPs that target complex domains such as Data mining and recommender systems.

LCEP in Data science

- RapidMiner [92] is a data science platform that provides an integrated environment for data preparation, machine learning, deep learning, predictive modeling, and other data analytics tasks. It offers a drag-and-drop interface that allows users to easily build, test, and deploy predictive models without the need for programming. RapidMiner supports a wide range of data sources and formats, including databases, spreadsheets, text files, and cloud-based data storage services. It also offers advanced data preparation and cleansing tools to ensure that

data is accurate and ready for analysis. One of the key features of RapidMiner is its machine learning capabilities, which allow users to build predictive models using a variety of algorithms, such as decision trees, random forests, and neural networks [93]. The platform also provides tools for model validation and optimization, allowing users to fine-tune their models for maximum accuracy. RapidMiner offers both a free, open-source version (RapidMiner Studio) as well as a commercial version with additional features (RapidMiner Server). It is used by businesses and organizations in a wide range of industries, including finance, healthcare, retail, and manufacturing.

- The Konstanz Information Miner (KNIME) [94] is a modular environment, which enables easy visual assembly and interactive execution of a data pipeline. This platform is designed as a teaching, research and collaboration platform, which enables data manipulation or visualization methods in the form of new modules or nodes. KNIME also provides a large selection of pre-built components and tools that can be easily integrated into workflows, as well as the ability to extend its functionality. For instance, KINME integrates with popular data science platforms as R and Python, making it easier for data scientists to incorporate their code into custom applications built on the platform. Finally, the platform is used in a wide range of industries, such as life sciences, finance, and marketing, for tasks such as predictive modeling, text mining, and customer segmentation.

- Kourouklidis et al. [95] developed a low-code technique for identifying and responding to events that can affect the performance of a machine learning model based on MDE concepts. The proposed solution is a cloud-based engine that enables machine learning specialists to design the execution of drift detecting algorithms on a computing cluster and receive email notifications of the results without the need for considerable software engineering knowledge. Their solution is based on DSL, which offers a standardized communication layer between domain experts who declaratively describe the behavior of the ML monitoring system and software engineers who are in charge of developing a concrete implementation that complies to the defined behavior.

LCEP in recommender systems

- Droid [96] is a open source framework for automating the configuration, evaluation, and synthesis of recommender systems for modeling languages. Droid tooling automates all steps of recommender system development, including data preprocessing, system configuration for the modeling language, evaluation and selection of the optimum recommendation algorithm, and deployment of the recommender system into a modeling tool. The Droit tool has been validated on multiple usecases, including recommending UML model attributes in domains such as Literature and Education [97]. Many recommender systems were trained in this approach using a number of collaborative, content-based, and hybrid recommendation methods such as item popularity, item-based collaborative filtering, user-based collaborative filtering, and so on. The recommendation systems are mostly based on the static features of elements and their occurrence in the dataset. When the requested recommendations become more complicated, the tool's performance can worsen. So far, the data's morphological natural language processing feature is regarded as a potential solution for improving recommendation systems. Droid can be accessed at https://droid-dsl.github.io/

- LEV4REC [98] is a low-code environment to foster a recommender systems's design, configuration, and deployment from scratch using such a cutting-edge paradigm. LEV4REC is flexible and extensible as it relies on three core techniques, i.e., feature model, metamodel, and Acceleo templates. Starting from a feature model, RS designers can specify the system's features and then progressively enrich a configuration model automatically generated out of the selected features. With LEV4REC, developers have the flexibility to explore a wide range of algorithms and methodologies for recommendation. They can experiment with different approaches, such as

collaborative filtering, content-based filtering, or hybrid methods, to identify the most effective solution for their specific use case. By fine-tuning the experimental settings, such as adjusting hyperparameters or incorporating additional features, developers can optimize the recommender system's performance and accuracy. Moreover, LEV4REC allows for the selection and evaluation of appropriate metrics to assess the recommender system's performance. Developers can analyze metrics like precision, recall, F1-score, or customized domain-specific metrics to gain insights into the system's effectiveness. This iterative process of experimentation and evaluation enables developers to continuously refine the recommender system until desired performance levels are achieved.

2.4 CHESS environment

The ever-increasing complexity and dependability issues of systems in various domains, such as transportation, space, energy, health, and industrial production, require effective design and development methods. The complexity and heterogeneity of components can be addressed with modeling approaches that span different technical disciplines and prove effective in the end-to-end engineering of the products. This implies taking into account various requirements such as quality, performance, cost, safety, security, and reliability. Model-based design technologies enable the user to perform beforehand different assurance-related activities such as physical architecture exploration, system behavioral analysis, early verification, and validation.

The CHESS toolset [99] offers cross-domain modeling and analysis of high-integrity systems providing an integrated framework that helps the modeler (user) to automate different development phases: from the requirements definition to the architectural modeling of the system's software and hardware, up to its deployment to a hardware platform [99]. CHESS follows a component-based approach where the user decouples different functional parts of the system as components that can be modeled, analyzed, verified, stored, reused individually, and integrated to meet the system's common goals. CHESS supports, among others, schedulability and dependability analysis across the entire project life cycle. The results of the analysis are back-propagated to the model itself so that the modeler can review and fine-tune the model to satisfy real-time and dependability requirements.

CHESS tool is a full-fledged open-source project, hosted by The Eclipse Foundation (https://www.eclipse.org/chess/). The code has been developed by various contributors following an open-source approach with public projects for issue tracking, code repository branches, and continuous integration. CHESS was developed, used, and extended in many research projects, by both industrial and academic partners. To list a few, CHESS was involved in international projects such as the homonymous CHESS project[2], CONCERTO[3], and SESAMO[4], under the ARTEMIS Joint Undertaking initiative, AMASS[5], AQUAS[6], and MegaM@Rt[7], under the ECSEL Joint Undertaking initiative. CHESS has been applied in different domains such as Avionics [100], Automotive [101], Space [102], Telecommunication [103], and Petroleum [104] [105].

Because CHESS is the foundation for our CHESSIoT extension, we go over the basics of CHESS in this section. We discuses its supporting engineering methodologies, such as multi-view modeling, component-based, and correct-by-construction approaches. Through the contract-based design analysis and model checking, we provide a quick introduction to the support verification and validation processes. In addition, we summarizes the existing analysis support, such as dependability, timing, safety, and quantitative reliability analyses. Lastly, we look at a number of successful stories and the impact CHESS has had on system engineering both in industry and academic settings.

[2] http://www.chess-project.org/
[3] http://www.concerto-project.org/
[4] http://sesamo-project.eu/
[5] https://www.amass-ecsel.eu/
[6] https://aquas-project.eu/
[7] https://megamart2-ecsel.eu/

2.4.1 CHESS in nutshell

The CHESS modeling tool was released under the Eclipse PolarSys project[8] and recently it was moved from the incubation status to the first major release. The CHESSML is an integrated modeling language profiled from OMG standard languages: UML, SysML, and MARTE under the Papyrus modeling environment [9]. Not all the features from all three languages were profiled to CHESS but only specific subsets that suit CHESS's perspective. In particular, sub-profiles supporting contract-based and dependability concerns have been defined, while MARTE has been adopted (with minor deviations) for what concern the timing perspective. There are different tools, plugins, and languages that were integrated into CHESS to support model validation, model checking, real-time, and dependability analysis. In this section, we are going to briefly describe the core aspects of CHESS methodology.

CHESS editor tooling

With reference to Figure 2.4, CHESS editor is an extension of the Papyrus UML editor and is activated when a CHESS model is created or opened. It provides additional functionality and constraints specific to the CHESS modeling approach. A *CHESS model* is essentially a UML model with the CHESS profile applied to it. To create a CHESS model, you can use a dedicated wizard that guides you through the process of applying the CHESS profile to your UML model.

When working with the CHESS editor, you can use the *CHESS design views*. Each design view imposes specific constraints on the UML diagrams and entities that can be created, viewed, or edited within that view. These constraints help enforce the rules and guidelines of the CHESS modeling methodology.

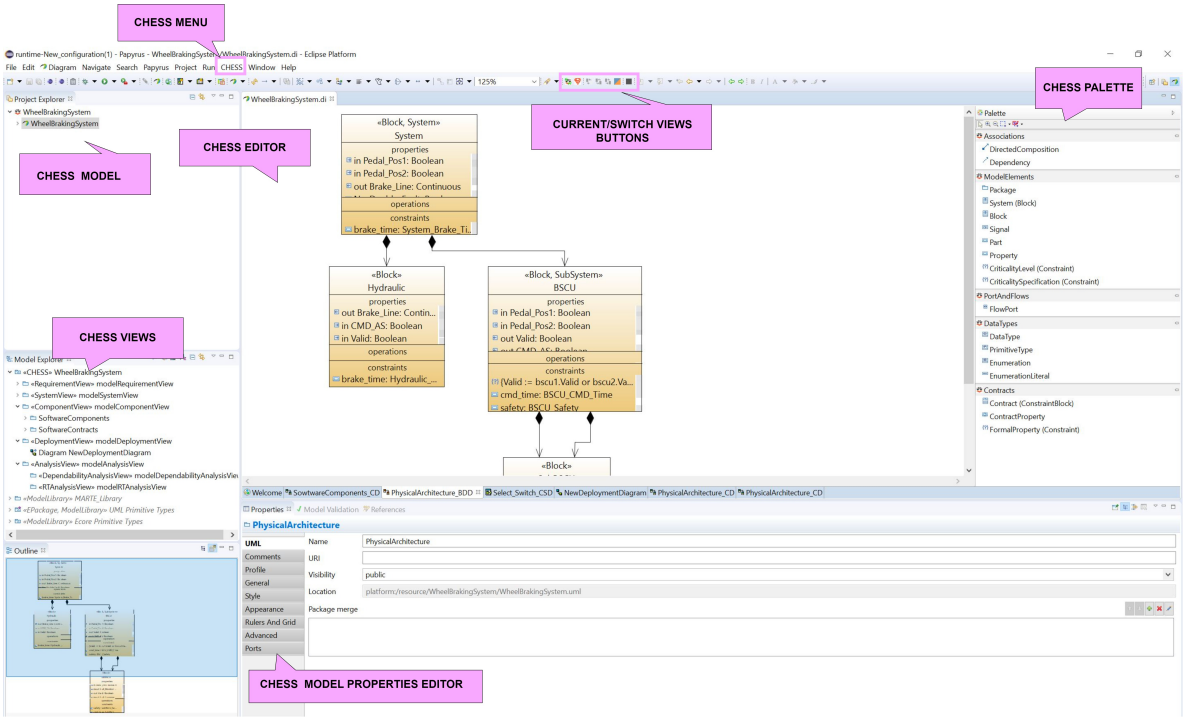


Figure 2.4: CHESS editor overview

The CHESS editor allows you to *switch between different views* based on your modeling needs. As you switch between views, the editor keeps track of the current view's status. It ensures that you do not violate the constraints defined for the specific diagram-view pair you are working with. This

[8] https://projects.eclipse.org/projects/polarsys.chess
[9] https://www.eclipse.org/papyrus/

prevents you from unintentionally creating or modifying UML elements that are not allowed in the current view.

In order to support the constraints of the current diagram view, the native *Papyrus palettes* have been customized. The customized palettes only display the entities that are allowed to be created within the current diagram view. This customization helps streamline the modeling process by presenting you with the appropriate set of options for the specific view you are working in.

Overall, the CHESS editor enhances the Papyrus UML editor by providing specialized features, design views, and constraints that align with the CHESS modeling approach. It facilitates adherence to the methodology and helps maintain consistency and correctness throughout the modeling activity.

Component-based methodology

Component-based design is an approach where software systems are decomposed into modular, reusable, and independent components. These components encapsulate specific functionality and can be composed and connected to build larger systems. The goal is to promote modularity, reusability, and maintainability of the software. The CHESSML language supports a component-based development methodology enabling property-preserving component assembly for real-time and dependable embedded systems. Emphasis is given to *separation of concerns* between the functional and the non-functional dimensions, such as safety, security, reliability, performance, and robustness [106].

The tool provides mechanisms to compose components and establish connections between them. You can define relationships such as aggregation, composition, and association to represent how components interact and collaborate. At the design level, components encompass functional concerns only (i.e., they are devoid of any constructs pertaining to tasking and specific computational models). Components interact with each other through well-defined interfaces. The CHESS tool enables you to define interfaces and specify the operations and properties that components expose to other components. The specification of non-functional attributes is then used for the automated generation of the container, enforcing the realization of the non-functional attributes declared for the component to be wrapped.

The CHESS methodology follows the “Correctness by Construction” practice which enforces *(1)* the use of formal and precise tools and notations for the development and the verification of all product items; *(2)* say things only once to avoid contradictions and repetitions; *(3)* the design of software components that are easy to verify, by e.g., using safer language subsets, and to implement, by using appropriate coding styles and design patterns [107]. The CHESS tool supports the deployment of components onto target platforms or execution environments. You can model the deployment architecture and specify the mapping of components to hardware or software resources. Finally, CHESS tool provides analysis capabilities to validate and verify component-based designs. You can perform checks and simulations to ensure that the components and their interactions adhere to design constraints and requirements.

Multi-view modeling approach

The CHESS tool provides a set of design views to uphold the "separation of concern", the "correctness by construction" and the other methodological principles introduced before. Six main views (requirement, component, system, deployment, analysis, and instance views) are defined to support The CHESS modeling approach. Throughout the development process, each view has its own underlined constraints that enforce its specific privileges on model entities and properties that can be manipulated. Depending on the current stage of the design process, CHESS sub-views are adopted to enhance certain design properties or stages of the process. Figure 2.5 shows the high-level architecture of CHESS views and their inter-relations.

- ***Requirement view:*** Originally adopted from the SysML requirement diagram, the requirement view is used to define system requirements and track their verification. In CHESS, requirements

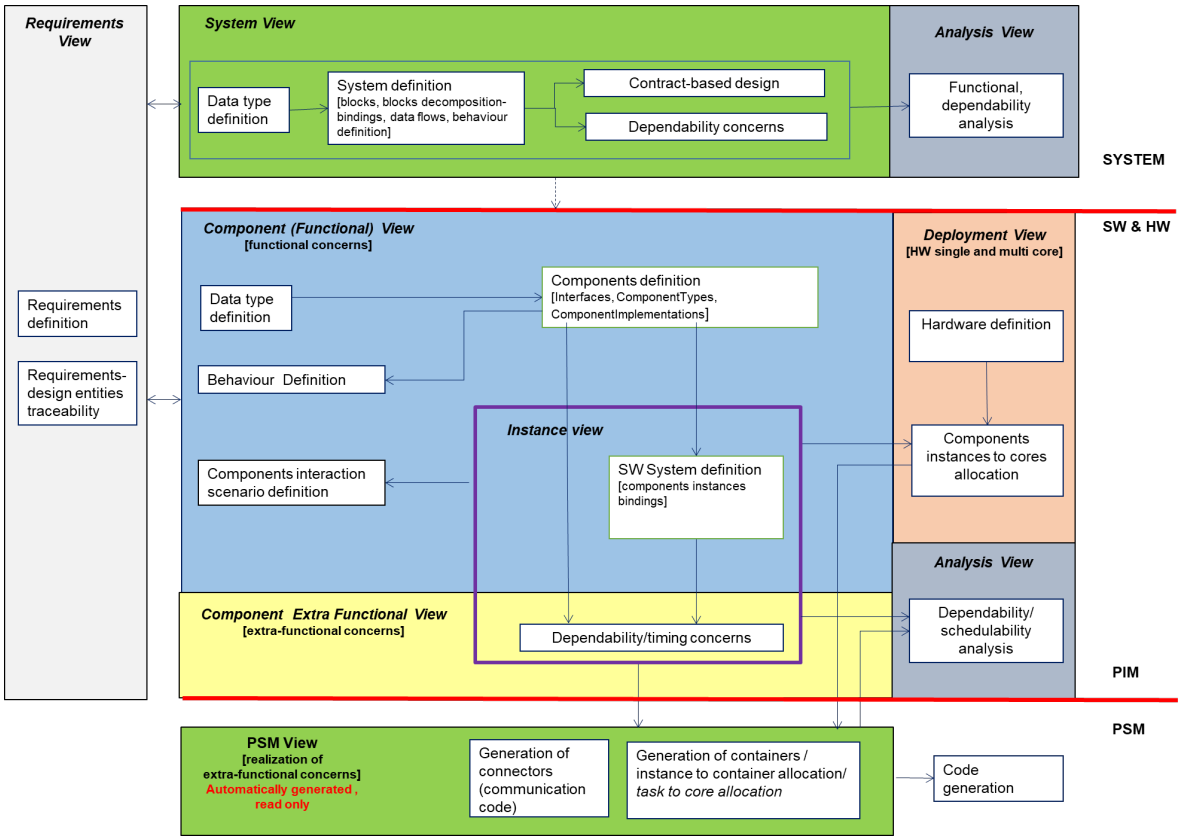


Figure 2.5: CHESS views architecture [2]

are part of the model and play a central role in the system development life cycle. The system elements are associated with the technical requirements they satisfy, which are, in turn, traced to higher-level requirements, up to system-level requirements [106]. This association technique enhances traceability while evaluating the correctness and consistency of the modeled system. In this way, the change's impact can be better evaluated and faithful model verification evidence can be provided according to the requirements. An example of a requirement created on a *WheelBrakingSystem (WBS)* example is shown in Figure 2.6

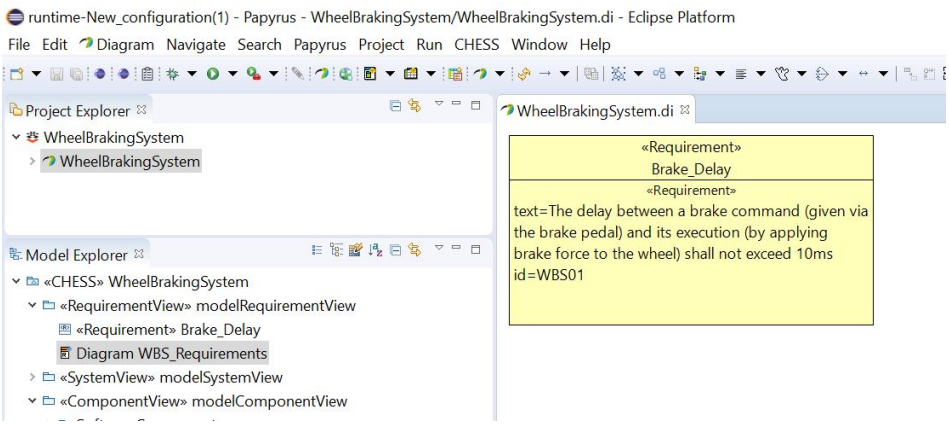


Figure 2.6: Wheel Braking System requirement example

- ***System view:*** It provides a suitable frame for system-level design activities. In the System view, the system entities are initially designed into blocks and then hierarchically decomposed (see *CHESS editor section* from Figure 2.4). CHESSML inherits from SysML the specification of the block hierarchies and their internal decomposition, i.e. a block definition diagram can describe

a system structure by means of a set of blocks and each block may have its own dedicated internal block diagram describing its sub-blocks decomposition and interfaces. An example of the internal block decomposition architecture of the WBS system example is shown in Figure 2.7. Furthermore, in addition to system-level design, in system view it possible to perform contract-based design as well as several functional and dependability analyses from system models.

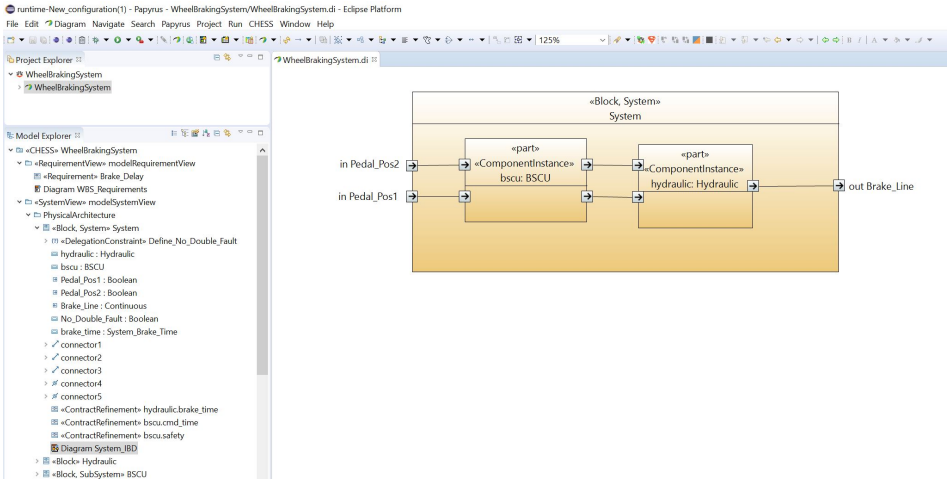


Figure 2.7: Wheel Braking System internal block diagram

- ***Component view:*** This view is used for software design work and logic of the intended model. The component view is composed of two sub-views, *Functional View* which is enabled by default, and the *Extra-Functional View* which is enabled manually in the tool. The *Functional View* is used to model system functional specifications using diagrams such as class, composite structure, state machine, activity, and sequence diagrams. Under the component view, the system's software construct can be designed through the modeling of component interfaces, component types as well as their components implementation. Figure 2.8 shows an example modeling of component implementations for a producer-consumer components implementation example.

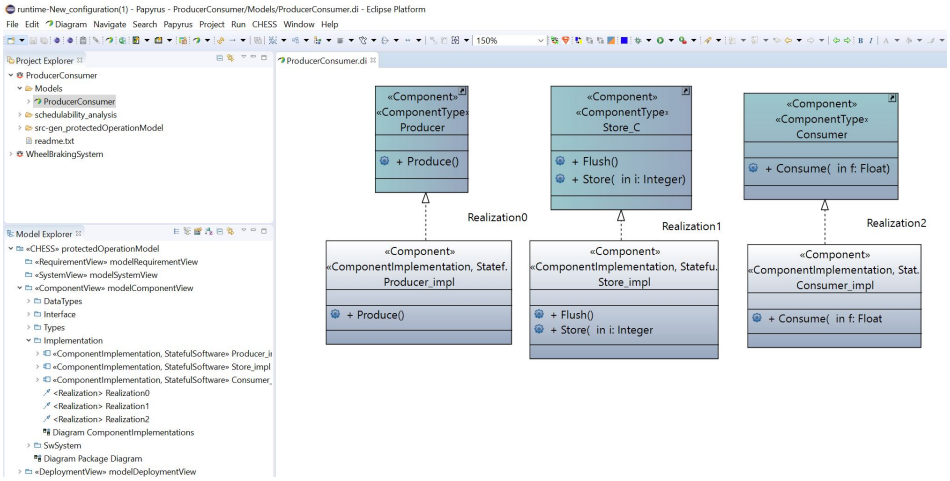


Figure 2.8: Producer-consumer: components implementation modeling

In addition to that, it is possible to model the system functional behavior using state machines which can be used for code generation purposes. On the other hand, when switched to it (see Figure 2.9), the *Extra-Functional View* can be used to compose the system's extra-functional specifications such as the real-time and dependability attributes. Recall that all views have a

dedicated palette depending on their requirements, for instance, the extra-functional view has no access to the activity diagram and has a palette with entries exclusively related to extra-functional concerns (See Figure 2.9).

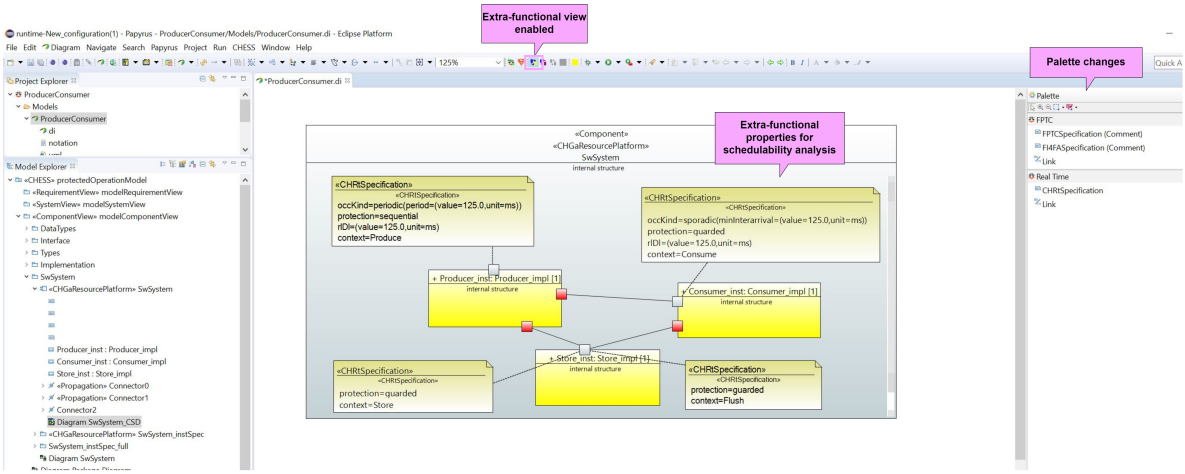


Figure 2.9: Producer-consumer: extra-functional properties modeling

- ***Deployment view:*** This view is used to model the hardware structure of the system and permits the allocation of their corresponding software component instances modeled from the before (from component-view). Through the use of class and composite structure diagrams, the user can model the type of deployment on either a single or multiple-core processor. In this view, each hardware resource is allocated to a specific memory partition and can only access and change its own memory space. Regarding the software-to-hardware resource allocation, all software components are allocated to cores. Figure 2.10 shows producer-consumer: software-to-hardware allocation modeling example in which the producer and consumer software component instances are allocated to cores on *CPU_inst* hardware instances.

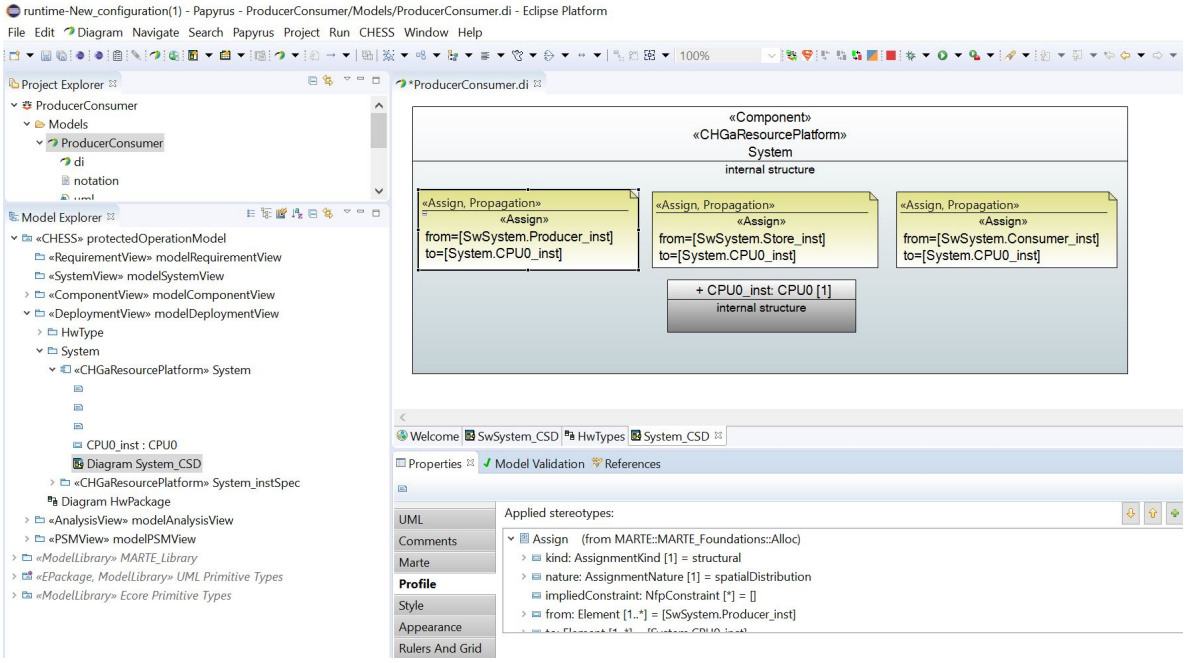


Figure 2.10: Producer-consumer: software-to-hardware allocation

- ***Analysis view:*** This is used to capture all the activities and diagrams related to analysis in CHESS. It consists of the two sub-views as *Dependability Analysis view* and *Real-Time Anal-*

ysis view. The analyses performed in CHESS are real-time analysis, quantitative dependability analysis, failure propagation analysis, and so on. We will discuss further analysis in Section 2.4.2.

- ***Instance view:*** CHESS offers a dedicated view to visualize and model the PSM. This model comprises a combination of hardware and software instances derived from the deployment and component views, respectively. Composite structure diagrams serve as the foundation for this representation, allowing for a detailed depiction of the system's structure and its dynamic behavior during runtime. One of the key strengths of CHESS is its ability to facilitate runtime analysis between model instances. By representing the PSM as a combination of hardware and software instances, it becomes easier to understand how the system components interact and how their behavior is affected by the underlying hardware. This comprehensive view enables analysts and developers to identify potential performance issues, resource constraints, and architectural issues that may impact the system's runtime behavior.

  The generation of model instances is automatic by invoking the **"BuildInstance"** command, the tool automatically creates the necessary hardware and software instances based on the deployment and component views. This automation saves time and effort for system designers and allows them to focus on the analysis and validation of the runtime behavior rather than manual instance creation. In the generated instance model, each component's properties and connectors are mapped onto dedicated *InstanceSpecifications*. This mapping provides a detailed representation of the characteristics and connections of the system's components within the PSM. By associating properties and connectors with specific instances, CHESS enables a deeper understanding of the runtime behavior and facilitates analysis of component interactions. More on this is presented our published work in [42].

2.4.2 Supported Model-based Analysis

In this section, we present different major model-based system analyses supported by CHESS.

Model-based system analysis and verification

CHESS provides the capability to perform several kinds of analysis depending on the specific requirements (functional, timing, dependability).

Timing Analysis is built on top of the MAST[10] analysis tool. It is invoked to perform analyses such as schedulability and end-to-end response time analysis. Schedulability analysis is performed by taking input from the annotated PSM model and the computed partition schedule on each available processing unit. Then, the response-time analysis calculates the worst-case response time of each task [100] assessing the schedulable tasks complying with the given timing constraints. Sample schedulability analysis results for the producer-consumer example is shown in Figure 2.11. The end-to-end response time analysis, on another hand, is done by utilizing the component sequence diagram. Applying MARTE timing stereotypes, the tool evaluates the hardware component's responsiveness. This analysis facilitates "early end-to-end response time verification", giving a sense of any possible refinement of the model before deployment [106] (See Figure 2.12).

Dependability analysis include Failure Logic Analysis and State-Based Quantitative Dependability Analysis. CHESSML dependability profile which normally supports different techniques for safety and dependability analysis has been extended to model fault injection and threats. Other new features include contract validations, parameter-based architectures, and document generation. Failure Logic Analysis [44] builds on top of Failure Propagation and Transformation Calculus [81] and enables deductive as well as inductive hazards analysis towards a semi-automatic generation of artifacts, necessary for arguing about HARA (Hazards Analysis and Risk Assessment). Supported by the DEEM tool [108], State-Based Quantitative Dependability Analysis [109] supports safety engineers in the

[10] https://mast.unican.es/

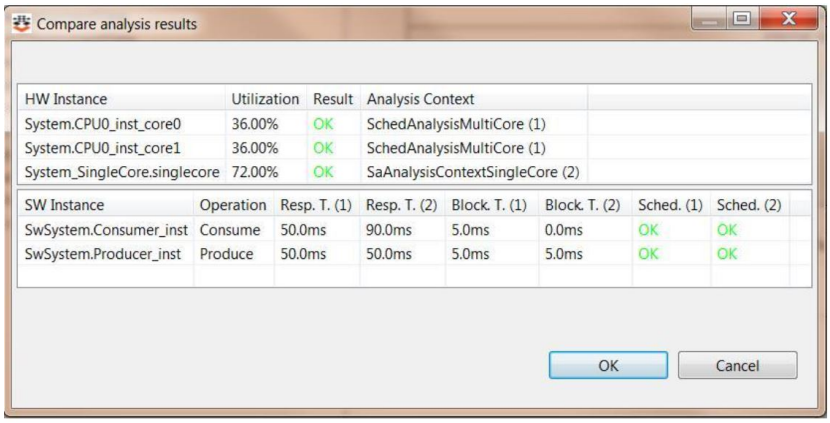


Figure 2.11: Producer-consumer: Schedulability analysis results

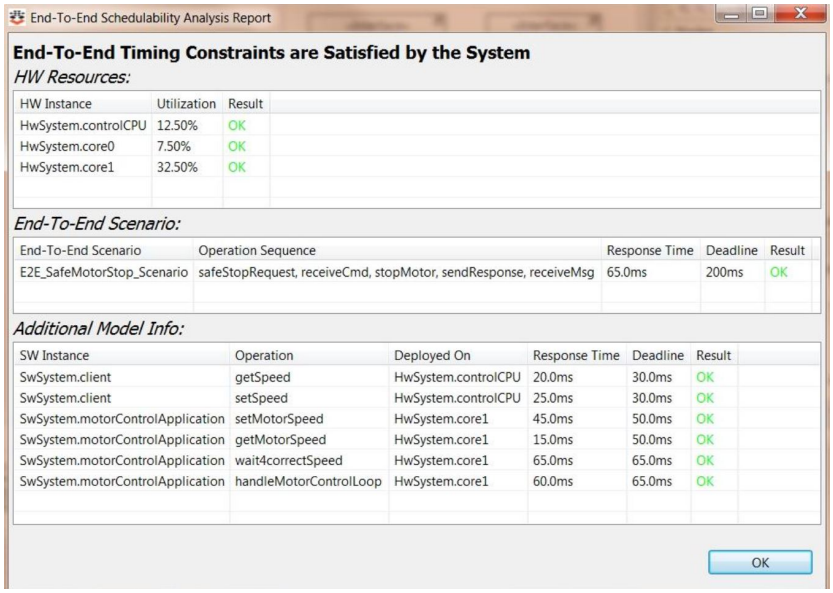


Figure 2.12: Producer-consumer: End-2-end response time analysis results

management of Reliability, Availability, Maintainability, and Safety (RAMS) properties, and in the assessment of hazard rate threshold associated with safety integrity levels. Properties modeled using the CHESSML dependability profile are analyzed to obtain the probability of occurrence of specific failure modes and other quantitative metrics involving reliability, availability, and safety. More on this is presented in Chapter 7.

Functional Verification by means of model checking is supported by the integrated *nuXmv* model checker [110]. System and component properties, derived from requirements, can be formalized into linear temporal logic properties, then they can be verified on top of the system's or component's behavioral models developed using state machines.

Model validation enforces several types of model constraints, depending on the specific analysis to be exploited, and the related application domain and criticality. For example, we can mention: (1) Model core constraints validation is performed to enforce the CHESS model constraints including specific preconditions as required by the schedulability analysis (2) Validate model for criticality specification and (3) Validate model for Automotive 26262 compliance (only specific for automotive domain) checks the system correctness of Automotive Safety Integrity Level(ASIL) inheritance and decomposition according to the ISO 26262 standard.

Contract-based design analysis and model checking

CHESS supports the specification of component contracts specified in the OCRA contract language. Requirements are formalized into Formal Properties, which contain OCRA assertions, i.e., textual specifications of temporal logic formulas (see Figure 2.14 for a contract while Figure 2.13 shows formal property specifications example).

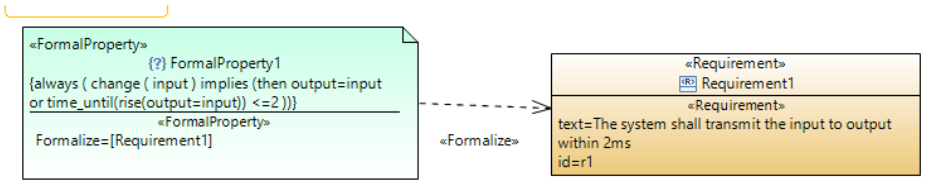


Figure 2.13: Example of a FormalProperty formalizing a requirement

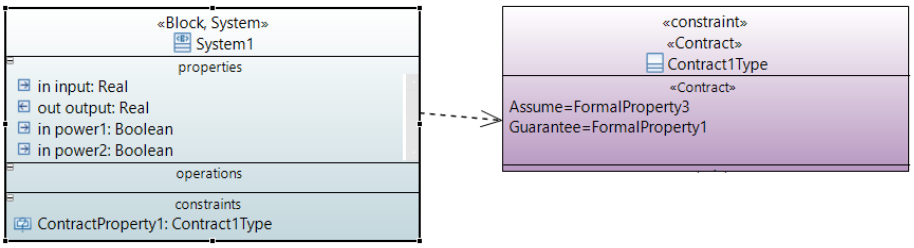


Figure 2.14: Example of a component contract

In the contract-based paradigm, these properties are restricted to the related component interface. A contract is a pair of properties representing an assumption and a guarantee of the component. In addition, the CHESS tool supports the contract refinement analysis for composite components. The contract of a composite component is defined by the assumption of the composite component itself and the guarantee ensured by the contracts of its sub-components, considering their interconnection as described by the architecture and that the assumption of each sub-component is ensured by the contracts of the other sibling sub-components.

CHESS supports the improved contract-based analysis aspects by integrating CHESS with verification and validation (V&V) tools such as OCRA [111], nuXmv [68], and xSAP [112]. In this regard, the contract-based analysis includes:

(1) *Model checking*, i.e. the behavioral models, that describe how the internal state of a component and the output ports are updated, can be verified against some formal properties in different temporal logics. The formal properties can represent some requirements (e.g., functional or safety-related requirements) or some validation queries such as the reachability of states.

(2) *Contract-based compositional verification of state machines* is performed on composite components. The local state machine of each sub-component is verified separately against its local contract. The correctness of the composite component is implicitly derived from the correctness of the contract refinement and the successful verification of local state machines.

(3) *Contract-based safety analysis*, i.e. identify the component failures as the failure of its implementation in satisfying the contract. When the component is composite, its failures can be caused by a failure of one or more sub-components and/or a failure of the environment in satisfying the assumption. As result, the analysis produces a fault tree in which each intermediate event represents the failure of a component or its environment, linked to a Boolean combination of other nodes. The top-level event is the failure of the system component. The basic events are the failures of the leaf sub-components, in addition to the failure of the environment (see [113] for more details).

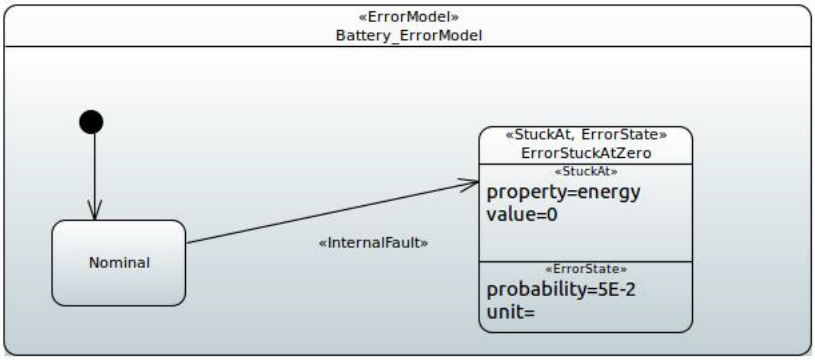


Figure 2.15: State machine modeling faulty behavior

Fault Injection and Safety Analysis

CHESS supports safety analysis based on fault injection thanks to the integration of xSAP [114]. The behavioral models of components are extended with faults and the tool automatically generates Fault Trees, showing the combinations of events leading to a failure or an undesired state, and Fault Mode Effect Analysis (FMEA) tables, listing all potential failure modes and their effects on the system [115].

More specifically, once the system model is defined in CHESS, through components definition and their nominal behavioral model, the faulty behavior is expressed through a specific state machine called *"Error Model"*. The Error Model extends the nominal state machine with information about the effect upon a property of the component, and consequently on its nominal behavior. Figure 2.15 represents an example of an error model that, in case of an internal fault, moves the related component in an error state where the property "energy" is stuck at 0 value. The optional probability assigned to that transition is $5 \cdot 10^{-2}$.

Once the error model is defined, the Fault Tree Analysis (FTA) or FMEA can be done by invoking xSAP through the CHESS environment. The xSAP approach is based on the library-based fault injection (i.e., an extension of a behavioral model with the definition of faults taken from a library of faults) and the use of model-based routines to generate safety artifacts. The result of the FTA is the fault tree that is automatically shown in a dedicated panel in the front end. If fault probabilities have been specified during the configuration of the error model, the fault tree will report their combination. The fault tree shows all minimal cut sets, identifying the basic fault conditions which can lead to top-level failure. This is complementary to other existing analysis techniques supported in CHESS such as Failure Logic Analysis and State-Based Quantitative Dependability Analysis, which do not consider the nominal behaviors and fault injection, but explicitly model the faulty behavior and fault propagation.

Quantitative Reliability Analysis

The CHESS profile for dependability is used to enrich functional models of the system with information regarding the behavior with respect to faults and failures, thus allowing properties like reliability, availability, and safety to be documented and analyzed.

CHESS supports the modeling of security concerns which helps in threat identification at the early stages of the development and facilitates the exploiting of the Mobius capabilities for analysis of reliability. **Möbius**[11] is a software tool for modeling the behavior of complex systems, by allowing the study of the reliability, availability, security, and performance for large-scale discrete-event systems [116]. Many reliability analysis results can be obtained with probabilistic models built with Mobius

[11] https://www.mobius.illinois.edu/

using the stochastic activity networks (SAN) formalism, solved via Monte-Carlo simulation[12]. Specific extensions of the dependability profile are related to the modeling of Cyber-Attacks aspects and model transformations from CHESS to the Mobius tool to run the analysis of SANs.

As result, the implemented methodology allows the modeling of a system security threat and data corruption which may result in service misfortune. An example of a system security threat can be a cyber-security attack, i.e. unauthorized access to the system, halting services. Figure 2.16 depicts the

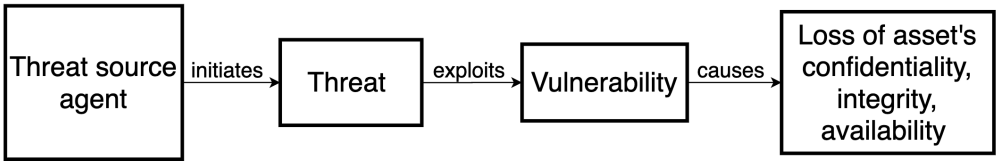


Figure 2.16: Process of Security breach

process of a security breach that leads to the violation of security-related properties. *A threat* event, initiated by a threat source agent, able to exploit a *vulnerability* of an asset (e.g. a component/system) may result in a loss to the confidentiality, integrity, and/or availability of the asset. Vulnerabilities could be represented as a pre-defined enumeration collected through different sources (e.g. personal competence, standards, results of previous threat analysis, etc.). Finally, the consequences could be modeled using pre-defined effects, which refers to the loss of Confidentiality, Integrity, and Availability (CIA).

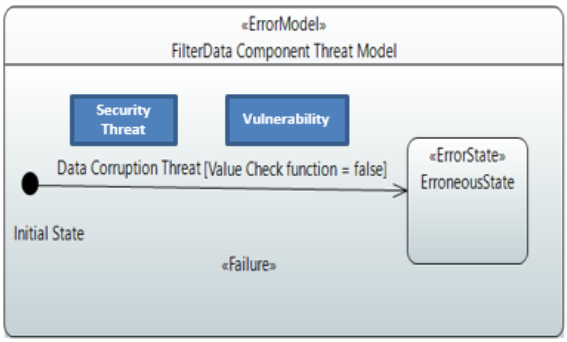


Figure 2.17: Erroneous state transition due to security threat event and vulnerability

An ***«ErrorModel»***-tagged state machine is used when modeling the security breach. The failure, internal fault, and effect are extended to include security threats, vulnerability, and consequences respectively. Figure 2.17 illustrates an example of an error model, where a cyber-security attack initiates a data corruption threat. The vulnerability was modeled by exploiting the value check function which is set to false. In this case, the system transits to an erroneous state leading to component failure. Note that a component could have multiple instances of ***«ErrorModel»***-tagged state machines, attached to it. Each instance would provide the elaboration of input/output failure behavior addressing a specific concern.

The generation of the Mobius SAN model process is done by performing an automatic model-to-model transformation from a model instance to the SAN model recognized by Mobius for reliability analysis. The reliability analysis is additional to the existing State Based Quantitative Dependability Analysis. It exploits Mobius's powerful and well-supported analysis capabilities as an engine for

[12] https://www.investopedia.com/terms/m/montecarlosimulation.asp

safety and security co-engineering, according to the scenario addressed in [117]. Editing Mobius models can be nontrivial, CHESS modeling language fully supports the modeling of system architectures taking into consideration safety and security co-engineering for reliability analysis with MOBIUS.

The generation of the Mobius SAN model process is done by performing an automatic model-to-model transformation from a CHESS model instance. The transformation engine was implemented using eclipse-based tools such as **QVTo** [118] and **Acceleo** [119]. The traceability information about the CHESS entities and the generated Ecore SAN model are saved in the SAN folder as **.qvtoTrace** file. The file comes with a dedicated editor to be used when checking the mappings. This extension has been developed in the context of the AQUAS project, as a result of a collaborative effort among Intecs and the City University of London. This approach was applied and evaluated across different use cases namely the Automated Teller Machine (ATM) and Industrial Drive. This approach provides a smooth integration, guarantees consistency among SysML and SAN models, and largely reduces the effort required to construct an appropriate SAN analysis model.

Support for parameterized architecture and trade-off analysis

In a parameterized architecture the number of components, the number of ports, the connections, and the static attributes of components depend on a set of parameters. Parameters are defined along the architectural hierarchy and, thus, the number of parameters themselves can depend on other parameters. CHESS supports the modeling of the parameterized architecture as well as its instantiation. In particular, the user can set the values of the parameters, defining the configuration of the architecture, and the tool automatically generates a concrete architecture corresponding to that configuration. Figure 2.18 shows an example of parameterized architecture.

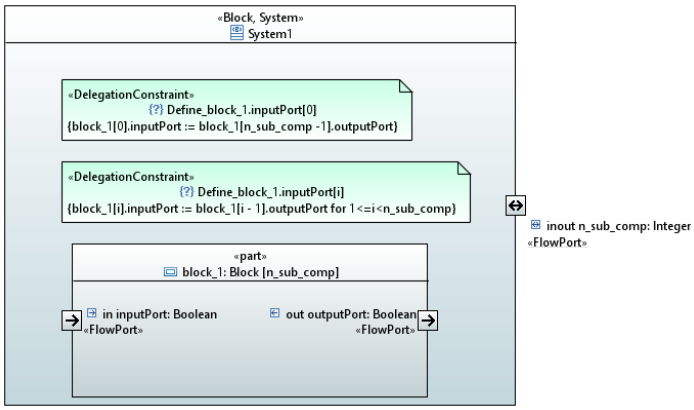
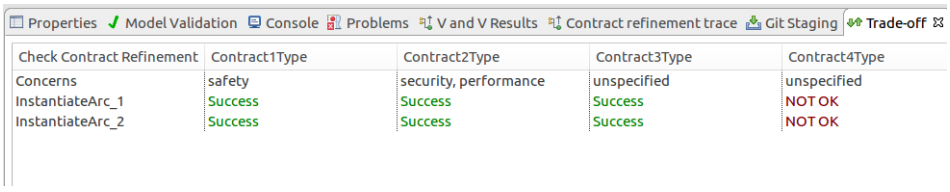


Figure 2.18: Example of parameterized architecture

The parameterized architecture is also exploited for trade-off analysis, by performing various analyses on the different instantiations and comparing the results. This makes it easy to visually get an idea of how the intended model instances perform with respect to the selected configurations. Figure 2.19 shows the sample result of a trade-off analysis made on two instances by looking at different concerns specified by the assumption/guarantee formal properties of each contract.



Check Contract Refinement	Contract1Type	Contract2Type	Contract3Type	Contract4Type
Concerns	safety	security, performance	unspecified	unspecified
InstantiateArc_1	Success	Success	Success	NOT OK
InstantiateArc_2	Success	Success	Success	NOT OK

Figure 2.19: Trade-off Analysis results sample

Automatic generation of diagrams and documentation

The traditional way of editing a model is by adding an element to a diagram but changes made in the model are not reflected in the diagrams. CHESS offers the possibility of generating a diagram from the model which reflects the data in the model on the fly. The supported diagrams are Block Definition Diagram (BDD) and Internal Block Diagram (IBD). The generated diagram elements will be automatically aligned but the user can rearrange by moving elements manually or by invoking *"layout selection command"*.

CHESS also supports the generation of the model architecture and the report on various analyses executed on the model in an HTML document or a LaTeX source code. The report is divided into two sections. The first describes the structure of the model which includes diagrams and the associated components while the second includes the report lists of the results of the validation and verification (V&V) analyses results. An example of generated model architecture report is shown in Figure 2.20.

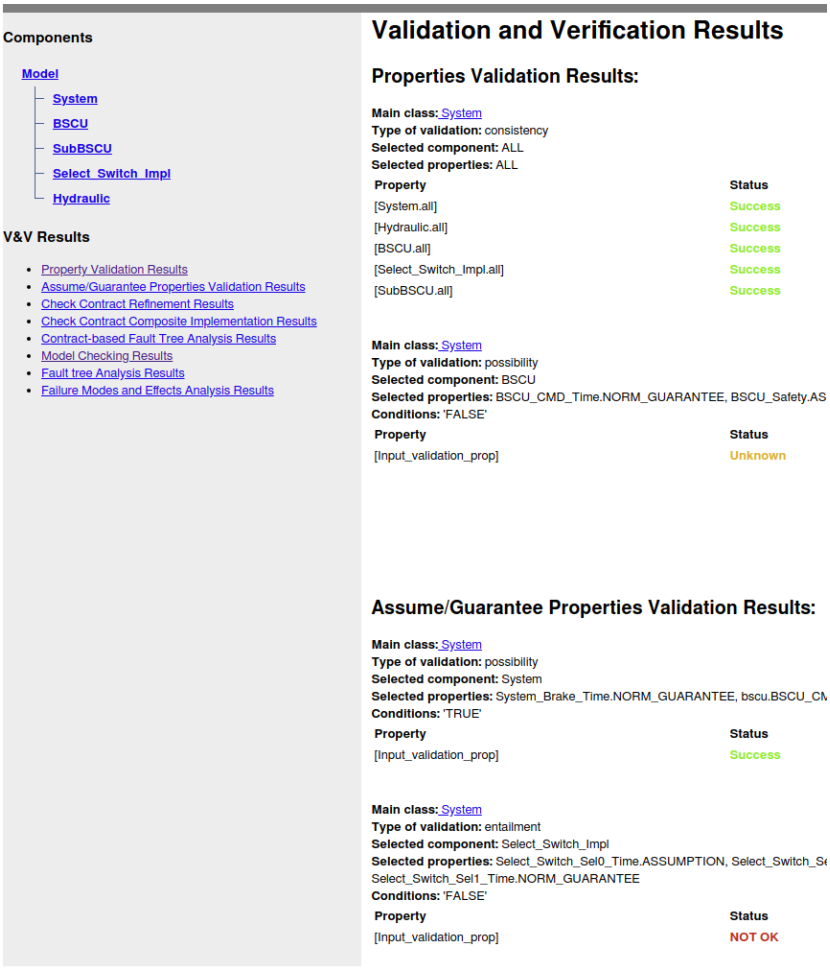


Figure 2.20: Generated report sample

2.4.3 Related tools

Several commercial tools provide similar functionalities to CHESS. One of the most popular is **Matlab/Simulink**[13]. Although Simulink facilitates the modeling and analysis of complex systems, its simulation efficiency might be an important disadvantage. Being based on a single Model of Computation and Communication (MoCC) is another limitation. **CoFluent**[14] is another commercial tool

[13] https://www.mathworks.com/products/simulink.html
[14] https://www.intel.com/content/www/us/en/cofluent/overview.html

extended to model IoT systems. Although supporting more interaction models than Matlab/Simulink, it is also limited in the way components may interact among them.

Another tendency is to overcome the UML lack of semantic content, required in some application domains, towards a proliferation of DSLs [60]. Among the available DSLs, UML/MARTE is the standard language for real-time and embedded systems design, while SysML is the standard language for system modeling. Several modeling environments like **Papyrus**[15] support UML/MARTE. Nevertheless, its flexibility and semantic richness require the definition of efficient modeling methodologies.

Capella[16] is an open-source comprehensive and extensible Eclipse system modeling tool. It is inspired by the SysML principles and it supports the ***ARCADIA*** methodology that is successfully deployed in a wide variety of industrial contexts [120]. ARCADIA provides architectural descriptions for functional analysis, structural analysis, interfaces, and behavior modeling, structured in five perspectives according to major system engineering activities and concerns.

COMPASS [121] supports model checking, model-based safety, reliability, and performance analysis and shares with CHESS some of the tools used as a backend for such analyses. Different from CHESS, it targets a variant of AADL and does not support traceability and code generation.

MapleSim[17] is a modeling tool for multi-domain engineering systems built on top of ***Modelica*** modeling language [122]. MapleSim features an integrated environment in which the system equations can be automatically generated and analyzed [123].

Although we see some approaches able to tackle modeling challenges, no tool or approach has been able to fit into our methodology with such analysis and verification functionalities. This makes CHESS a novel approach for implementing component-based modeling methodology for real-time and dependable systems by taking care of non-functional properties and enforcing correctness at all the stages of the development process.

2.4.4 Conclusions

Dependable complex system design and development present several challenges; the well-known canonical approach is to divide complex systems into smaller chunks (or subsystems), build them separately, and later integrate them. In this section, we presented the current state of the CHESS tool to tackle the design and verification of real-time dependable complex systems. First, we briefly presented the CHESS tool engineering methodologies, such as component-based and multi-view modeling environments. Next, we have summarized the existing analysis support, such as dependability, timing, safety, and quantitative reliability analyses.

[15] https://www.eclipse.org/papyrus/
[16] https://www.eclipse.org/capella/
[17] https://www.maplesoft.com/products/maplesim/

Chapter 3

IoT Engineering Platforms: a state of the art

This chapter presents the state of the art by summarizing all of the existing approaches related to several topics presented in this dissertation. This chapter covers related approaches and tools available in different area of interest in which this thesis covers. For instance, we covers related works in domains such as Low-code development tools, model-driven development, deployment, safety analysis and the software quality models of LCEPs in the IoT domain.

This chapter is organized as follows: Section 3.1 reviews the current low-code-based methodologies for engineering IoT systems, highlighting general-purpose approaches as well as IoT-specific approaches that focus on general design and development as well as service-oriented development approaches. Section 3.2 presents the MDE approach for modeling and developing IoT systems. Then, in section 3.3, we present the MDE approaches that focus on deploying IoT systems. In Section 3.4, the related system analysis approach is described with an emphasis on safety-related approaches. Lastly, in Section 3.5, we present the existing conceptual methods for evaluating the quality of IoT engineering systems.

3.1 Low-Code Development Platforms for IoT

In this section, we make an overview of existing LCDPs for IoT that take into account MDE concepts in their core implementations.

3.1.1 General-purpose LCDPs for supporting IoT applications development

Mendix [124] is one of the popular LCDPs that offer significant enterprise characteristics especially attractive to large businesses [125]. Its platform is equipped for multi-cloud and hybrid computing, due to its support for on-premises, virtual private multi-cloud, and multi-tenant public cloud deployment options. With Mendix, you can easily connect to different devices and sensors, and use data from these devices to create applications that can automate processes, provide insights, and enable real-time monitoring and control. In addition, Mendix provides a range of pre-built connectors and integrations for popular IoT platforms and protocols such as MQTT, AWS IoT, and Azure IoT. These integrations make it easy to connect your Mendix application to your existing IoT infrastructure and start collecting and analyzing data right away.

Salesforce [126] is a popular CRM LCDP that has been adopted through many different new technologies like AI, Machine Learning, and Cloud computing. Salesforce supports the rapid prototyping of IoT applications through the connection with the underlying Salesforce IoT cloud engines [126]. This platform provides data visualization and event management through a visual set of rules and triggers on different data source components. Furthermore, Salesforce IoT Cloud allows businesses to

create rules that trigger automated actions based on IoT data, such as sending alerts, updating records, or creating new cases in Salesforce. Additionally, it provides real-time insights into IoT data, helping businesses to optimize their operations and make data-driven decisions.

ThingWorx [127] offers a set of tools and services that assist companies to develop, implement, and manage IoT solutions. ThingWorx is designed to make the process of developing and deploying IoT applications easier by allowing developers to focus on designing their applications instead of infrastructure and platform difficulties. It includes a visual drag-and-drop interface that enables developers to design complicated IoT apps without writing much code. ThingWorx also includes an array of pre-built connectors and templates for fast integrating IoT devices and data channels. Finally includes a powerful analytics engine that allows developers to analyze and visualize data from IoT devices in real time as well as ensure data protection from unauthorized access.

Microsoft Power Platform [128] offered by Microsoft, is an LCDP for business software development. While it's not specifically designed for developing IoT applications, it can certainly be used to develop IoT applications with some additional tools and services. PowerApps, which is part of the platform, provides a visual interface for building custom mobile and web applications that can interact with IoT devices. Power Apps include connectors that can be used to communicate with IoT devices, such as Azure IoT Hub or IoT Central. Power Automate, another tool in the platform, can be used to create automated workflows that connect IoT devices to other systems and services such as Azure Event Grid or Azure Stream Analytics. Finally, Microsoft offers a variety of other services that can be used to develop IoT applications, such as Azure IoT Edge and Azure IoT Hub. By using these services in conjunction with the Power Platform, developers can quickly and easily create custom IoT applications with minimal coding.

AWSIoTCore [129] is a fully-managed service provided by Amazon Web Services (AWS) that allows developers to connect, manage, and securely communicate with IoT devices and applications. It provides a platform for developing IoT applications by facilitating device connectivity, message routing, and data management. It includes an LCDP called AWS IoT Things Graph, which allows developers to visually model the structure and behavior of their IoT systems. It offers different development features such as device management which register and manage devices; Rules Engine which allows you to define rules to filter, transform, and route data between devices and AWS services Finally, it integrates the developed application with other AWS services, including Amazon Kinesis, Amazon DynamoDB, and AWS Lambda.

IBM Watson IoT Platform [130] provided by IBM, provides a set of tools and services to help developers build IoT applications. It offers a range of capabilities such as device connectivity, data management, analytics, and cognitive computing, all of which can be used to develop and deploy IoT applications. The platform is designed to handle large volumes of data generated by IoT devices and sensors and provides secure and scalable communication between devices and applications. It supports a variety of protocols such as MQTT, HTTP, and CoAP, making it easy to connect different types of devices and sensors to the platform. One of the key features of the IBM Watson IoT Platform is its ability to apply analytics and machine learning to the data generated by IoT devices. The platform includes a range of built-in analytics tools, such as dashboards and predictive analytics, that can be used to gain insights into the data and make data-driven decisions.

Simplifier [131] integrates business and IoT applications that enable users to create, manage, deploy, and maintain enterprise-grade SAPUI5[1] apps for web, mobile, and wearables. Simplifier uses a pre-built interface for bidirectional integration of existing SAP and non-SAP systems and leverages shop-floor integration with native IoT interfaces (OPC-UA, MQTT) [131]. It is provided as a web-based environment available on-premise or in the cloud. Simplifier permits to the deployment of applications on the SAP Cloud Platform, SAP NetWeaver, as stand-alone, or on a dedicated Simplifier cloud.

GoogleCloudIoT [132] offered by Google, offers end-to-end solutions for device connectivity, data ingestion, processing, storage, and visualization. The platform also supports a wide range of

[1] https://www.guru99.com/sapui5-tutorial.html

IoT devices and protocols, including MQTT, HTTP, and CoAP. It also integrates with popular IoT development platforms such as Arduino and Raspberry Pi. With Google Cloud IoT, developers can easily create and manage device registries, configure device settings, and monitor device health and status. The platform also provides secure and reliable data transport and storage, using encryption and authentication protocols to ensure data privacy and integrity. In addition, Google Cloud IoT offers a range of analytics and machine learning tools, such as BigQuery and Cloud ML Engine, to help developers gain insights from IoT data and build intelligent applications.

3.1.2 IoT specific LCDPs for system modeling and development

Modeling IoT systems need to take into account heterogeneous parts of the system which run at different levels. There are several LCDPs available for IoT structural modeling and development. In this section, we analyze approaches that propose the modeling of the entire structural aspects of the system.

Node-RED [89] is a programming tool specifically conceived in the IoT context, with the aim of wiring and connecting together hardware devices, APIs, and online services [89]. It provides a cloud-based editor that makes it easy to connect together flows using the wide range of nodes in the palette that can be deployed to its runtime easily. Node-RED provides a rich text editor built on top of Node.js taking full advantage of its event-driven, non-blocking model. Node-RED can be run locally or on the cloud. Node-RED is platform agnostic and compatible with several devices such as Raspberry Pi, BeagleBone Black, Arduino, Android-based devices. Node-RED also supports its integration with cloud-based resources such as IBM Cloud, SenseTecnic FRED, Amazon Web Services, and Microsoft Azure. Finally, in Node-RED, the user can also create and deploy real-time dashboards.

AtmosphereIoT [90] provides IoT solution builders with languages and tools to build, connect, and manage embedded-to-cloud systems. Atmosphere IoT Studio offers a free drag-and-drop online IDE, to build all device firmware, mobile apps, and cloud dashboards. AtmosphereIoT connects devices from a range of wireless options including Wi-Fi, Bluetooth and BLE, Sigfox, LoRa, ZigBee, NFC, satellite, and cellular. This platform is entirely cloud-based but it offers downloadable artifacts. Finally, AtmosphereIoT offers a range of analytics and visualization tools that allow developers to monitor and analyze data from their IoT devices in real time. This helps developers to identify issues, optimize performance, and improve the overall functionality of their applications.

DSL-4-IoT [133] is a cloud-based modeling tool for the IoT domain, which comprises a JavaScript-based graphical frontend programming language and a runtime "OpenHAB" execution engine. DSL-4-IoT provides a multistage model-driven approach for the design of IoT applications that supports all stages of the life cycle of these systems. Automatic model transformations are provided to refine abstract model elements into concrete ones. Those transformation results formatted as JSON-Arrays are passed to the OpenHAB runtime engine for execution. In addition to that, it can help to improve the reliability and performance of the system, DSL-4-IoT can make it easier to test and debug the system, since the rules and automation logic can be evaluated independently of the underlying device hardware and software.

BIoTA [134] offers a cloud-based modeling approach for IoT architectures. The specification and implementation of the BIoTA language involve a grammar and a compiler, responsible for syntax and semantic analysis, as well as code generation. A graphical DSL and supporting tools allow users to perform syntax and semantic analysis. BIoTA renders it possible to computationally formalize a software architecture suggested by a user according to formal automata techniques. The component & connectors are created following specific rules to meet IoT-specific scenarios while exporting the resulting software architecture to a software distribution package pattern based on containers (Docker Compose) for future deployment.

Kiljander et al [135] proposed a cloud-based textual language and tool for Event-based Configuration of Smart Environments (ECSE) had been proposed. The tool enables the end-user, expert or not, to configure a smart environment by employing an ontology-based model. In their approach, the authors used the Resource Description Frameworks (RDF) to define the event-action rules.

Bezerra et al. [136] propose a cloud-based approach for creating responsive and configurable Web of Things user interfaces. Models@Runtime are used to produce runtime interfaces based on a formal model named Thing Description (TD). TD's goal is to expose Web Things (WT) attributes, actions, and events to the outside ecosystem. The modeling language has been developed in JavaScript, using the VueJS framework, and it is publicly available[2].

FloWare Core [137] is a model-driven open-source toolchain for building and managing IoT systems. FloWare supports the Software Product Line and Flow-Based Programming paradigms to manage the complexity in the numerous stages of the IoT application development process. The system configures the IoT application following the IoT system model supplied by the IoT developer. A Node-RED engine [89] is integrated into FloWare.

Vitruvius [138] is an MDD platform that allows users with no programming experience to create and deploy complex IoT web applications based on real-time data from connected vehicles and sensors. Users can design their ViWapplications straight from the web using a custom Vitruvius XML domain-specific language. Furthermore, Vitruvius provides a variety of recommendation and auto-completion features that aid in creating applications by reducing the amount of XML code to be written.

3.1.3 IoT specific LCDPs for service-oriented applications development

This category includes approaches providing users with cloud-based modeling environments targeting service-oriented architectures. Thus, different services are connected to build the final IoT systems.

MIDGAR [139] is an IoT platform specifically developed to address the service generation of applications that interconnect heterogeneous objects. This is achieved by using a graphical DSL in which the user can interconnect and specify the execution flow of different things. Once the desired model is ready, it gets processed through the service generation layer, generating a tree-based representation model. The model is then used to generate a Java application that can be compiled and run on the server.

IADev [140] is a model-driven development framework that orchestrates IoT services and generates software implementation artifacts for heterogeneous IoT systems while supporting multi-level modeling and transformation. This is accomplished by converting requirements into a solution architecture using attribute-driven design. In addition, the components of the produced application communicate using RESTful APIs.

LogicIoT [141] offer a textual web-based DSL to ease data access and processing semantics in IoT and Smart Cities settings. LogicIoT is implemented as a set of custom Jakarta Server Pages (JSP)[3] in which different custom JSP tags have been implemented to define the modeling semantics. The language consists of seven constructs: relations, triggers, endpoints, timers, facts, rules, and modules. Using the custom tags, the user can define the application's operations required to enable the communication between process instances and sensors without being concerned with low-level programming details.

glue.things [142] offers a cloud-based mashup platform for wiring data of Web-enabled IoT devices and Web services. glue.things take care of both the delivery and maintenance of device data streams, apps, and their integration. In this regard, glue.things rely on well-established real-time communication networks to facilitate device integration and data stream management. The glue.things modeling tool combines device and real-time communication, allowing users to describe elements' triggers and actions and deploy them in a distributed manner.

Taherkordi et al. [143] proposed a cloud-based framework for scalable and real-time modeling of cloud-based IoT services in large-scale applications, such as smart cities. IoT services are modeled and organized in a hierarchical manner by relying on references to services and their real-time data. In

[2] https://github.com/smar2t/td_interface_builder

[3] https://en.wikipedia.org/wiki/Jakarta_Server_Pages

order to guarantee real-time and fresh service delivery to interested parties, the service tree supports notification-based access to service data and changes.

Valsamakis et al. [144] presented a portable web-based graphical end-user programming environment for personal apps is proposed. This tool allows users to discover smart things in their environment and create personalized applications that represent their own needs. Each of the defined smart objects can provide various features that can be published via a well-defined API. The graphical representation of the system is then generated from the constructed JavaScript objects in which the user can interact with the system on the fly.

E-SODA [145] is a cloud-based DSL under the Cloud-Edge-Beneath (CEB) architecture ecosystem. In E-SODA, a cloud sensor comprises a set of Event/Condition/Action (ECA) rules that define the sensor service life-cycle. It allows the user to be abstract and simulate sensor behavior in an events-based fashion. This is achieved by having the ECA rules listen for the occurrence of a predetermined "event" and respond by performing the "action" if the rule's "condition" is met. Finally, the generated cloud sensor application can be used in any cloud-based application which needs sensor data.

Mayer et al. [146] introduced an integrated graphical programming tool based on a goal-driven approach, in which end users are only required to specify their purpose in a machine-understandable manner, rather than designing a service architecture that fulfills their goal. This allows a smart environment's ultimate purpose to be graphically represented, but the complexities of the underlying semantics are hidden. A reasoning component uses the provided goal statement and analyses whether the goal can be achieved given the set of available services and infers whatever user actions (i.e., requests involving REST resources) are required to achieve it.

InteroEvery [147] is a cloud-based development tool that promotes a micro-service-based architecture to deal with interoperability issues of the IoT domain. First, an IoT system is configured through a web-based graphical interface showing each micro-service functionalities. A universal broker connects a dedicated interoperability micro-service with various adaption micro-services depending on employed choreography patterns.

3.2 Model-driven design and development of IoT systems

In this section, we present the related work with more emphasis on software modeling and development that may further generate code ready for deployment on IoT devices.

Ciccozzi, F. et al. [36] presented MDE4IoT, an MDE platform that combines different UML DSLs as profiles used through different viewpoints to enforce the separation of concerns. Its main goal is to combine the support for the design, development, and runtime management of IoT systems. This is achieved by providing means for supporting the modeling and self-adaptation of Emergent Configurations (ECs) of connected systems. MDE4IoT performs a series of model-to-model as well as model-to-text transformations to satisfy the generated platform-specific code from state machines. The run-time monitoring and self-adaptations are supported through the re-allocations s well as re-generation mechanisms according to the system's runtime feedback.

Costa B. et al. [50] presented SysML4IoT a Model-Based Systems Engineering tool for IoT application development, focusing on the design phase. SysML4IoT is strongly based on IoT-A domain reference model [49] established by European research body as well as ISO/IEC/IEEE 15288 standard [4], aiming to enhance system models with Systems Engineering concepts [49]. To address different stakeholders involved in the process, the tool adopts a multi-disciplined IoT application design by using views and viewpoints. In [148], SysML4IoT was been extended to assist IoT application engineers in precisely modeling IoT applications and verifying their quality of service (QoS) properties. Through a model-to-text translator that converts the model and QoS properties specified on it to be executed by NuSMV [68], a mature model checker that allows entering a system model comprising a number of communicating Finite State Machines (FSM) and automatically checks its properties specified as Computational Tree Logic (CTL) or Linear Temporal Logic (LTL) formulas. The tool has been

[4] https://standards.ieee.org/ieee/15288/5673/

adopted in [149] for developing IoT self-adaptive systems endorsing the public/subscribe paradigm to model communication with other systems.

Thramboulidis K. et al. [150] introduced UML4IoT, an MDE platform for industrial automation systems. It was designed to support the full automation of the generation process of the IoT-compliant layer required for the cyber-physical component to be effectively integrated into the modern IoT manufacturing environment. A model-to-model transformation has been implemented to automatically transform the mechatronic components into Industrial Automation Things (IAT). The tool adopted the Open Mobile Alliance (OMA) LWM2M application protocol running on top of the CoAP communication protocol as the mean for exposing the IoT interface as simple smart objects [151]. The approach also enables the usage of high-level languages such as Java to specify the system's behavior in case a higher-level design specification such as the UML one is not available.

Harrand N. et al. [37] presented ThingML, an engineering IoT platform that combines well-proven textual software-modeling constructs aligned with UML (statecharts and components) and an imperative platform-independent action language for developing IoT applications. In ThingML, a thing can be defined by a set of properties, functions, messages, ports, as well as state machines. These behaviors are local to a thing and can be accessed only through interfaces inside the state machines or functions. The interaction between things is enabled through required or provided ports by means of message exchanges. The tools include an advanced multi-platform code generation framework that supports multiple target programming languages such as C/C++, Java, Arduino, JavaScript, Python, and Go. In [152], ThingML was extended to assist IoT/cyber-physical modeling with machine learning needs. The new approach targets the issue of IoT communications and behavioral modeling which was normally done using state machines.

Nicholson R. et al. [39] presented IoTML, a tool developed in the context of the BRAIN-IoT project [39] as an integrated modeling tool to ease rapid prototyping of intelligent cooperative IoT systems based on shared models. The BRAIN-IoT modeling environment i.e IoTML is implemented as a Papyrus profile. The BRAIN-IoT architecture mainly consists of three macro-blocks: the BRAIN-IoT Modeling Framework, the Marketplace, and the Federation of BRAIN-IoT Fabrics. The constructed models are transformed into XML format before being uploaded to the BRAIN-IoT marketplace for run-time system deployment and dynamic remote edge/cloud reconfiguration.

M. de Farias et al. [153] presented a Cloud and Model-based IDE for the IoT tool (COMFIT) to target wireless sensor networks (WSN) applications for IoT. The COMFIT modeling environment is built on top of Papyrus, and it presents a simple multi-view environment to model the system's requirement, structural and behavioral aspects. The wireless nodes of the system and their communication links are created in the structural view, while the model activities and behaviors are modeled as functional units, which later get linked according to the desired execution sequence. Finally, the tool provides the model-checking infrastructure respecting the OCL rules specified in the meta-model.

Muccini H. et al. [72] introduced CAPS an architecture-driven modeling framework for the development of Situational Aware Cyber-Physical Systems. CAPS is based on a multi-view architectural approach that combines the design for the IoT system's software components and their interactions, the hardware specification of situational awareness, as well as the physical environment where hardware equipment is deployed. To link together the modeled views, the authors introduced two auxiliary languages, denoted Mapping Modeling Language (MAPML) and Deployment Modeling Language (DEPML). The authors used the Atlas Model Weaver (AMW)[5] to define relations among models and to create semantic links among model elements. In [154], CAPSml extension to CAPS was introduced to support the platform-specific code generation through the usage of the ThingML [37].

Dhouib S. et al. [155] introduced Papyrus4IoT, a modeling tool developed under the Smart, Safe, and Security Software Development and Execution Platform (S3P) project. Papyrus4IoT environment that enables the design and deployment of complex IoT systems following an IoT-A reference architecture [7]. the designer can define process specification definition, functional, and operational platform, and the deployment which is done by allocating the system's functional blocks to the device

[5] https://projects.eclipse.org/projects/modeling.gmt.amw

processing units. The authors proposed the use of development-time models to supervise a running IoT system to reflect the Models@Runtime monitoring approach. This typically helps in detecting overall system critical states and helps make decisions on the adaptation of the running system. Authors suggest the extension of Papyrus Moka [156] to perform model simulations. Concerning the deployment of the modeled systems, authors make use of Prismtech's Vortex as a dynamic platform to discover and deploy microservices, and MicroEJ as the target operating system.

Salihbegovic A. et al. [133] presented DSL-4-IoT, a tool based on a high-level visual programming language established to tackle the complexity and heterogeneity of IoT systems. The Editor enables the application designer to configure the system structure and select devices, sensors, and actuators either from built-in library modules available. When the design is finished, the user can export the data into one JSON array configuration file. This file keeps the information about the position of all the items within the configuration, relationships between items and groups, the value of all configured fields associated with items, and of data types. After all configuration files are generated, they can be transferred to the respective OpenHAB runtime directory manually or automatically downloaded, using a simple web service for execution.

Erazo-Garzón L. et al. [26] presented Monitor-IoT, a graphical designer (high-level visual language) built in the Obeo Designer Community and Eclipse Sirius tools to support developers in modeling IoT multi-layer monitoring architectures with a high level of abstraction, expressiveness, and flexibility. Monitor-IoT supports the definition of computing nodes and their resources that support the monitoring processes (data collection, transport, processing, and storage) at the edge, fog, and cloud layers. It is also possible to specify the properties to be monitored for each entity as well as the definition of dataflows between digital entities, based on synchronous or asynchronous communication. Although Monitor-IoT supports a variety of different interesting concepts, it doe no support the generation of any kind of code, monitoring scripts, or data flows which can be executed on an actual IoT system.

Pramudianto F. et al. [157] presented IoTLink, a development toolkit based on a model-driven approach to allow inexperienced developers to compose mashup applications through a graphical domain-specific language. Modeled applications can be easily configured and wired together to create an IoT application. Through visual components, IoT Link encapsulates the complexity of communicating with devices and services on the internet and abstracts them as virtual objects that are accessible through different communication technologies. To support interoperability with other services, authors implemented custom components like ArduinoSerial for Arduino connectivity, SOAPInupt, RESTInput, MQTTInput, etc. The tool is able to generate a complete Java project including an extendable Java code. At runtime, the tool generates connections by using the Drools[6] engine to poll the rules from a database repository, which allows developers to deploy and change deployment rules at runtime. In a controlled experiment, IoT Link was 42% faster than using a Java library and able to outperform the Java library's user satisfaction.

Corradini et al. [158] presented X-IoT, a model-driven approach supporting the development of cross-platform IoT applications. X-IoT is based on a DSML and its related notation, whose development has been guided by a deep analysis of IoT application characteristics. The proposal covers the modeling, development, and deployment phases, supporting different actors in the development process and the derivation of specific artifacts, resulting in a model-to-code transformation approach. X-IoT tackles the IoT platform portability issue by promoting "a single development/multiple deployments" strategy. Tool support is provided through the ADOxx platform [7], which allows using the DSML to model platform-independent IoT applications. The resulting application can be successively refined by introducing platform-specific information and then deployed on the selected IoT platform.

Thang Nguyen et a. [159] presented a FRAmework for Sensor Application Development (FRASAD), a model-driven framework to develop IoT applications. The tool was developed with aim of tackling the reusability, flexibility, and maintainability of sensor software. FRASAD relies on a node-centric

[6] https://www.drools.org/
[7] https://www.adoxx.org

software architecture model in which a rule-based programming model is enabled by a DSL that uncouples the programming language and the execution model used by the underlying operating system. FRASAD has been developed on top of Eclipse EMF/GMF and consists of a graphical modeling language, a code generator, and other supporting tools to help developers design, implement, optimize, and test the developed IoT applications. Two case studies are provided to show the usability and portability of our framework. The evaluation results demonstrate that our framework FRASAD can be considered a promising solution to reduce the complexity of IoT software development.

Nepomuceno et al. [160] presented AutoIoT, a framework that allows users to model their IoT systems using a simple JSON file. The process starts by modeling the system using the graphical interface generated from GMF. When the modeling phase is completed, AutoIoT loads the content of the model as a JSON file to be validated and transformed into Python objects using the Pydantic[8] library. After that, the framework finally delivers these objects to an appropriate *Builder* that performs model-to-text transformations to generate a ready-to-use IoT server-side application. The Prototype Builder generates a Flask application written in Python, HTML, CSS, and Javascript. The generated server-side application communicates with IoT devices and third-party systems through MQTT, Rest API, and WebSockets.

Soukaras et al. [161] presented IoTSuite, a suite of tools for IoT applications development, for reducing development effort. The tool consists of the following components: *i)* an *editor* to support the application design phase by allowing stakeholders to specify high-level descriptions of the system under development; *ii)* an ANTLR[9] based *compiler* that parses the high-level specification and supports the application development phase by producing programming framework that reduces development effort in specifying the details of components of an IoT application; *iii)* a *deployment module*, which is supported by the mapper and linker modules; *iv)* a *runtime system*, which leverages existing middleware platforms and it is responsible for the distributed execution of the modeled IoT application. The current implementation of IoTSuite targets both Android and JavaSE-enabled devices and makes use of an MQTT-based middleware.

Xuan Thang et al. [162] presented a DSL designed for specifying all aspects of a sensor node application, especially for data processing tasks such as sampling, aggregation, and forwarding. The proposed DSL offers a set of declarative sentences to express the behavior of sensor nodes application such as *sampling*, *aggregating*, and *forwarding* which is necessary for developing data-centric Wireless Sensor Networks (WSN) applications. The tool is based on Eclipse GMF for specifying PIMs. The transformation from the PIM to nesC[10] PSM models has been implemented by using the ATL transformation language[11]. Acceleo-based model-to-text transformations have been developed to generate the final nesC source code of the modeled system.

3.3 MDE for deployment of IoT systems

In the context of IoT, MDE can help to simplify the deployment process by automating many of the tasks involved, such as configuring devices, setting up communication protocols, and managing data flows. By creating models that capture the key aspects of an IoT system, such as the devices, sensors, and data streams involved, MDE can provide a high-level view of the system and make it easier to manage and maintain. In this section, we go over the MDE approach that focuses mainly on the deployment modeling and automation of IoT systems.

Kirchhof et al. [17] introduced MontiThings a C&C language offering automatic error handling capabilities and a clear separation between business logic and implementation details. Built on top of MontiArc [163], MontiThing is an integrated modeling language for architectures of IoT applications, their deployment, and error handling that lifts the level of abstraction in the IoT system engineering

[8] https://pydantic-docs.helpmanual.io/
[9] https://www.antlr.org/
[10] http://nescc.sourceforge.net/
[11] https://www.eclipse.org/atl/

process. The error-handling approach makes C&C-based IoT applications more reliable without cluttering the business logic with error-handling code that is time-consuming to develop and makes the models hard to understand, especially for non-experts. Targeting mainly the edge layer, Montithings provides a model-driven toolchain for the automated synthesis of executable IoT containers, and automated deployment planning, featuring deployment suggestions, for the generated containers. Finally, the tool monitors the generated container and in case of need, the tool is able to suggest the deployment goals changes based on deployment planning feedback.

Duran et al. [164] proposed a new technique for supporting the reconfiguration of running IoT applications, represented as a set of coordinated rules acting on devices. These techniques compare two versions of an application (before and after reconfiguration) to check if several functional and quantitative properties are satisfied. This information can be used by the user to decide whether the actual deployment of the new application should be triggered or not. The approach uses advanced Event-Condition-Action (ECA) rules by providing means for the composition of rules, such as the sequential execution of rules, the choice between several rules, the concurrent execution of several rules, or the repetition of rules. Finally, the property property-based verification was implemented to analyze whether the proposed reconfiguration preserves the consistency of the application.

Ivan A. et al. [165] proposed a model-based approach for the specification and execution of self-adaptive multi-layered IoT systems. A domain-specific language (DSL) for the specification of such architectures, and a runtime framework to support the system behavior and its self-adaptation at runtime were presented. The proposed DSL covers modeling primitives covering the four layers of an IoT system that includes IoT devices (sensors or actuators), edge, fog, and cloud nodes. The modeling of the deployment and grouping of container-based applications on that node. In addition to that, the tool supports a specific language to express adaptation rules to guarantee QoS at runtime. A proof of concept of a generator for deploying and executing the runtime state of modeled IoT system on a K3S-based infrastructure (Kubernetes distribution built for IoT and edge computing) is also provided.

Negash et al. [166] introduced DoS-IL, a textual domain scripting language for resource-constrained IoT devices. It allows a flexible and scalable approach that enhances modifiability and programmability through client-server-server-client architecture. DoS-IL allows changing the system's behavior after deployment through a lightweight script written with the DoS-IL language and stored in a gateway at the fog layer. This mainly is to support easy maintenance and modification after deployment, without the need to physically access the end node. The gateway hosts an interpreter to execute DoS-IL scripts accessible by devices in the perception layer. The interpreter splits the script into tokens first, identifies the function of each token, and structures it in a convenient way for execution. On the target node, the Device Object Model (DOM) exposes the available resources for the DoS-IL script to manipulate.

Fei Li et al. [167] proposed Topology and Orchestration Specification for Cloud Applications (TOSCA), a structured (XML-based) language that defines different components of an application and relations between them using an application topology while capturing all management tasks in management plans. TOSCA aims at automating the deployment and management of composite applications by providing a generic way to describe the application topology of composite cloud applications and leverages portable workflow languages to ensure the portability of deployment and management plans. Moreover, it aims at improving the reusability of service management processes and automating IoT application deployment in heterogeneous environments. In TOSCA, common IoT components such as gateways and drivers can be modeled. In addition, the gateway-specific artifacts necessary for application deployment can also be specified to ease the deployment tasks.

Ferry et al. [168] proposed GENESIS, a textual cloud-based domain-specific modeling language that supports continuous orchestration and deployment of Smart IoT systems on edge, and cloud infrastructures. GENESIS (Generation and Deployment of Smart IoT Systems) uses component-based approaches to facilitate the separation of concerns and reusability; therefore, deployment models can be regarded as an assembly of components. The GENESIS execution engines support three types of deployable artifacts, namely ThingML model [37], Node-RED container [89], and any black-box deployable artifact (e.g., an executable jar). The created deployment model is subsequently passed to

the GENESIS deployment execution engine, which is in charge of deploying the software components, ensuring communication between them, supplying the required cloud resources, and monitoring the deployment's status.

Erazo-Garzón et al. [26] introduced Monitor-IoT, a graphical designer (high-level visual language) built in the Obeo Designer Community and Eclipse Sirius tools to support developers in modeling IoT multi-layer monitoring architectures with a high level of abstraction, expressiveness, and flexibility. Monitor-IoT supports the definition of computing nodes and their resources that support the monitoring processes (data collection, transport, processing, and storage) at the edge, fog, and cloud layers. It is also possible to specify the properties to be monitored for each entity as well as the definition of dataflows between digital entities, based on synchronous or asynchronous communication. Although Monitor-IoT supports a variety of different interesting concepts, it doe no support the generation of any kind of code, monitoring scripts, or data flows which can be executed on an actual IoT system.

3.4 MDE for safety analysis of IoT systems

MDE can help with safety analysis by allowing designers and safety engineers to create models of the system that capture its behavior and interactions with the physical world. These models can then be used to perform various types of safety analysis, such as hazard identification, Fault-Tree, and Failure Mode and Effects Analysis (FMEA) [79]. MDE-based safety analysis can be particularly effective for complex IoT systems, where the interactions between devices, sensors, and data streams can be difficult to understand and analyze manually. By creating models that capture the system's behavior and interactions, MDE can provide a more complete and accurate view of the system and its potential risks.

Fault tree analysis is one of the hugely used and suggested methods when performing different dependability analysis studies, including safety analysis [34]. We have also mentioned that FTs are among the mandatory artifacts that should be provided for performing Safety Analysis in different domains and IoT is yet to follow [32]. However, most of the approaches presented in the literature still rely on the manual construction of the FTs, which still makes the process time-consuming.

One of the widely used tools in the industry as well as in the academia to perform the FTA is the ISOGRAPH tool [169]. The ISOGRAPH Reliability workbench is a powerful integral visual modeling and analysis environment in which all the aspects of the reliability analysis such as failure rate and maintainability prediction, Failure Mode Effects & Criticality Analysis (FMECA), Reliability Allocation, Reliability Block Diagram as well as Fault Tree, Event Tree, and Markov analysis are combined. Although this tool is seemingly powerful in terms of what can be covered, different from our approach presented in Chapter 7 in which the system FTs are automatically generated from the analysis, the FTs are still manually constructed from the system failure requirements provided by the safety experts.

Haider et al. [3] employs the CHESS Failure Logic Analysis results to build European Cooperation for Space Standardization (ECSS)[12] compatible FTs. Although their approach is linked with ours, it differs significantly in various points. To name a few, their approach only supports system-level component composition while creating FT, while our approach supports any level of composition. For instance, with our approach, FTs of a single composite component can be generated and analyzed individually. Furthermore, in their approach, only basic events from the system-level input ports can be generated, whereas in our case, any component can initiate a basic failure event. Finally, their approach only supports FT generation, however, unlike our approach, they neither support qualitative nor quantitative FT analysis.

Parri et al. [170] presented JARVIS (Just-in-time ARtificial intelligence for the eValuation of Industrial Signals), a model-driven tool that facilitates the development and verification of the integration of physical IoT devices, enterprise-scale software agents, data analytics, and human operators.

[12] https://ecss.nl/standards/

JARVIS promotes semi-formal specification of structural elements, functional requirements, and behavioral characteristics of subsystems from a System of a Systems perspective. JARVIS employs agents to facilitate the development and integration of intelligent data agents capable of detecting failure events that occur in accordance with a set of failure modes. Eventually, a *FaultTreeAnalyzer* agent is used to perform Fault Tree Analysis on detected failure events. Although their approach performs a qualitative analysis, the quantitative one is not supported. Finally, their FT generation approach relies on the practical data model constructed by the deployed agent, while our approach relies on FLA for the FT generation.

Several approaches have been proposed for the automatic generation of FTs from SysML models. For example, Mhenni et al. [171] present an approach for generating FTs from SysML models, relying on a combination of information provided in activity and IBD diagrams as well as information in the FMEA table. Although the current tool generates a single FT picture representing the system failure paths, no FT models are generated. Another critical difference with respect to our proposed approach regards the use of *directed graph traversal* and *block design patterns* to generate FTs which is nothing but using the component-directed edge relationships to determine how the next component has to be represented in an FT. Even though the presented block design patterns are useful to derive the component failure propagation behaviors, they do not cover certain topics such as "internal failure of the components", since this information is probably picked from the FMEA table, as well as they do not provide any support for any kind of automated qualitative or quantitative FT analysis.

Alshboul et al. [4] presented an MDE environment for performing preliminary safety analysis from SysML models. The approaches use UML state machines to model the component functional behavior and annotate them with failure behaviors; later this information is used to generate the system FTs. Although the proposed approach generates the FTs, certain aspects of the safety analysis are not covered such as injected or external failures, as well as the qualitative or quantitative analysis of the generated FTs. On another hand, Yakymets et al. [172] presented a framework that integrates the formal method approach for facilitating the automatic FT generation within an MDE workflow. The approach annotates to the SysML model elements the formal analytical expressions showing how deviations in the block outputs can be caused by internal failures of the block and/or possible deviations in the block inputs. Later this information is transformed into an AltaRica model [173] representation which is used to perform qualitative and quantitative analysis using the XFTA tool provided by the framework. Although this approach seems very interesting, the process of annotating the model with formal analytical expression can be very complex to grasp whereas, in our proposed approach, failure logic behavior rules following FPTC notation are used and we retain they are simpler and straightforward to be used.

In terms of safety-critical systems, Han et al. [84] presented an approach for performing a combination of FMEA and FTA analysis on safety-critical systems starting from the Preliminary Hazard Analysis (PHA) method, initially conducted by the safety experts. However, no supporting tool is provided. Same as Hame at al. [174] an approach for manually deriving FT diagrams from the Reliability Block Diagram (RBD) was also presented, however, the qualitative and quantitative analysis are manually performed, differently from our approach where the analysis is performed automatically. Furthermore, unlike our approach which models the system architecture, annotates the model with safety-related information, and later generates and analyzes FTs, several approaches, such as [175–177], propose SysML profiles which are used to create FT models and later translate them into FT graphs without any support for system modeling itself. On another hand, Chaari et al. [178] proposes a Meta-modeling-based Failure Propagation Analysis (MetaFPA) framework to support the synthesizing of the system failure propagation models in order to help the creation of the system FTs. Although the presented framework presents an alternative to FPTC on how system failure propagation rules can be modeled, unlike our proposed approach, the framework does not generate the system FTs but relies on the ISOGRAPH tool [169] to perform the FTA. The same goes with the approaches proposed in [174, 179] which rely on the ISOGRAPH software to manually construct and analyze the FTs.

In the IoT domain, very few approaches specifically target the execution of safety analysis on IoT systems. This might be mainly caused by the lack of systematic, disciplined, and quantifiable

software engineering standards, as well as comprehensive abstraction models for dealing with the increasing complexity and safety requirement heterogeneity present in the IoT domain. Silva et al. [180] presented a dependability evaluation tool for IoT applications, when hardware and permanent link faults are considered. The tool supports the modeling of system network architecture and, later, the so-called network failure condition events (*nfc*) are defined to help in generating the FT. The *nfc* formalism somehow follows the logical association rules for addition and multiplication in order to reflect the "OR" and the "AND" gates respectively. Finally, the tool supports the qualitative analysis, by generating minimal cut-sets, as well as the quantitative analysis. Although this tool supports the automatic generation and the analysis of the FTs, it differs from our approach presented in Chapter 7 both in terms of system failure behavior formalism and because it does not support any mechanism related to failure transformation, propagation, and injection.

Chen et al. [181] presented an intelligent method for fault diagnosis based on a combination of FTA and fuzzy neural networks in the aquaculture IoT systems. In their approach, the FT is manually constructed for each component of the system and later the “IF-THEN” rules are extracted from the FT to be fed into the fuzzy neural network to train the relationship model between fault symptoms (failures) and faults. Although this method uses the FTA for the safety analysis process, it differs from our approach since the generation of the FT is done manually, while in our case it is performed automatically. Furthermore, our approach conducts a quantitative analysis by calculating the system failure probability, while their approach does not. Finally, Xing et al. [182] presented an FT modeling infrastructure in which different reliability analyses for mesh topology IoT networks are performed taking into account the quantitative analyses. However, same as the ISOGRAPH tool, the aspect of the FT construction is still manually done from the system failure requirements provided by the safety expert.

3.5 Software product quality model for IoT LCEPs

Practitioners typically rely on well-established standards and practices to improve confidence in whether a system or a product fits the wanted quality requirements. ISO/IEC 25010:2011 standard is one of the model standards that have been heavily used to assess the quality of complex software and systems. In this section, we present the existing approaches that was been used to evaluate the quality of IoT systems based on ISO/IEC 25010:2011 standard.

To name a few, Azham ate al. [18] relied on it to assess the quality of online health awareness systems, Martinez et al. [19] relied on the model for assessing the quality of IoT brokers. Schipore et al. [183] presented Euphoria, a software architecture design and implementation that enables easy prototyping, deployment, and evaluation of adaptable and flexible interactions across heterogeneous devices in smart environments. The tool was designed and developed following the requirement set by the ISO/IEC 25010:2011 standard. Johan et al. [20] relied on the standard to evaluate the quality of IoT implementations. Although this quality model’s scope is intended general for software and computer systems, it can also be applied to assess larger systems and services [14]. For instance, Janine et al. [25] adopted the standard to assess the quality of mobile applications, as well as in [21, 22] for Machine learning and Big data systems.

There have been several approaches in the modeling domain for quality measurements. For example, Gökhan at al. [184] introduced a Framework for Qualitative Assessment of DSLs (FQAD) was presented. The FQAD framework is based on ISO/IEC 25010:2011 when determining the evaluation’s perspective, the assessment’s goal, and selecting relevant quality characteristics to guide the assessment process. In addition to that, in [185] Christian et el. relied on the standard to assess the quality assurance of DSLs. In contrast, Mohamed et al. [186] employed the standard to evaluate the quality of design architectures. In addition, Miguel et al. [187] relied on the standard to evaluate MDE quality studies in attaining maximum quality evaluation level by considering that study that touched more of the quality characteristics as specified by the standards would be ranked as optimal.

Luana et al. [23] used the standard to perform a mapping study on the quality assessment of

software product lines. The standard was used by Jara et al. [188] to assess the security quality of mobile cloud computing-based technologies. Finally, Bernardes et al [24] used it to evaluate the quality aspects of customer relationship management (CRM) systems. A large number of extensions and suggestions on the standard have been proposed. To name a few, Estdale et al. [189] proposed the extension to meet the lifetime service-oriented quality aspect of software systems. Rahman et al [21] conducted a study to assess the product quality of Big Data systems concerning non-functional requirements, while Gonzalez et al. [190] extended the model to Semantic Web exploration tools.

Finally, Farshidi et al. [191] presents a highly comprehensive effort to assist the quality evaluation of MDD platforms, as well as a mapping from the proposed model to the ISO/IEC 25010:2011 standard. In their approach, a multi-criteria decision-making (MCDM) model for MDD platforms is used to help in choosing an optimal quality sufficient platform for their requirements. External decision models can be uploaded to the existing decision support system (DSS) knowledge base to support long-term software-producing organizations make decisions based on their needs and preferences. Nevertheless, the approach as well as the results presented are too generic, whereas our approach focuses primarily on IoT-specific platforms. Furthermore, the approach, in our opinion, focuses mostly on LCDPs, while other conventional MDE platforms, such as those based on the Eclipse Development Environment, are not taken into account at all.

Chapter 4

Limitations and open challenges of existing IoT Engineering Platforms

IoT Engineering platforms have to cope with several challenges mainly because of the heterogeneity of the involved sub-systems and components. With the aim of conceiving languages and tools supporting the development of IoT systems, this chapter presents the results of different studies conducted to understand the current state of the art of LCEPs in the IoT domain. By partially targeting to answer the first research problem (RP1), this chapter initially focuses on MDE and Low-Code approaches to satisfy the engineering support of IoT systems. First, we present a general overview of what an IoT Engineering platform should support through a well-conceived taxonomy of features. By doing so, we selected sixteen platforms, and such features were used to evaluate the functionalities and services supported by each analyzed platform. Furthermore, we identified some weaknesses of the analyzed platforms to pave the way toward a Low-Code platform for developing IoT systems. As a last step, we have also identified some limitations of existing approaches and discussed possible ways to improve and address them in the future.
The current evolution of cloud-based computing opens up many possibilities for software development. In the near future, the engineering of complex systems in various domains, such as Space, Automotive, IoT, and Smart Cities, will be done from cloud-based environments, lowering production and maintenance costs. In particular, parts of the IoT domain have to run on Edge, Fog, or Cloud, posing significant difficulties in determining what, where, and when to develop. Therefore, as a contribution toward answering the first research problem (RP1), this chapter also present the results of the state-of-the-art review we conducted to investigate where the IoT domain community stands concerning the current trend of moving traditional modeling infrastructures to the cloud. After examining 611 articles, we focus on 22 different cloud-based IoT system development approaches. The considered approaches have been analyzed to assess their strengths and weaknesses concerning many characteristics, including their modeling focus, accessibility, openness, and artifact generation. Throughout the chapter, we have discussed many challenges IoT developers encounter while adopting such tools. We also discuss various generic technologies and tools which can be adopted in the IoT domain.
The chapter is organized as follows: Firstly, Section 4.1 demonstrates the current state of the art of IoT engineering platforms in general, including both Low-code and MDE approaches. In fact, section 4.1.1 highlights the IoT engineering platform features, and 4.1.3 evaluates the outcomes of the study while sub-section 4.1.4 concludes the section. Secondly, Section 4.2 covers the current state of the art in cloud-based modeling methodologies in IoT. Section 4.2.1, for instance, presents related studies on cloud-based modeling in IoT, Section 4.2.2 presents the study design methodology while Section 4.2.3 describes the study's results. Section 4.2.4 presents the topic's challenges, and Section 4.2.5 highlights related opportunities.

4.1 Engineering IoT platforms

According to European Union CONNECT Advisory Forum report [192], IoT promises to be one of the most disruptive technological revolutions since the advent of the Internet. It is projected that by the end of 2030, fifty to hundred billion IoT devices will be connected to the Internet using a variety of information technologies [193]. As we experience in daily life, now we see more and more intelligent traffic lights, advanced parking technologies, smart homes, and intelligent cargo movement. This is due to the rising adoption of artificial intelligence (AI), and 5G infrastructure is helping the global IoT market register increased growth.

IoT engineering platform aims to simplify and streamline the process of developing and deploying IoT applications, helping developers to focus on the higher-level functionality of their applications instead of worrying about the underlying infrastructure and connectivity [8]. IoT engineering platforms especially commercial ones may also include a set of pre-built components, such as sensors, actuators, and communication protocols, that can be used to quickly develop IoT applications without having to build everything from scratch [90]. Additionally, these platforms often support various programming languages, making it easier for developers to build applications using the language they are most comfortable with [89].

In this section, we present the current state of research model-based approaches for engineering IoT systems by taking into account LCDPs in particular. We present a general overview of what we think a typical IoT Engineering platform should support through a well-conceived taxonomy of features. We present the results and the findings that have been done by analyzing sixteen IoT development platforms. They are divided into two categories considering their basic implementation mechanisms. In particular, the first category consists of tools based on the Eclipse technologies such as Eclipse Modeling Framework(EMF), Graphical Modeling Framework(GMF), and Papyrus environment. The second category is a collection of tailor-made platforms LCDPs.

4.1.1 Engineering IoT platforms features

In this section, we introduce a taxonomy of terms, which can support the description and the comparison of different approaches for the development of IoT systems. By analyzing different languages and tools drafted from Chapter 3, we identified and formalized their corresponding variabilities and commonalities in terms of a feature diagram. These features were selected mainly based on our expertise as well as the studies from different research papers. Figure 4.1 shows the top-level feature diagram, where each sub-node represents a major point of variation.

Requirement modeling support: This group of features emphasizes the first stages of any MDE-based development process. This evaluates whether a tool has an inbuilt requirement specification environment. Supporting this feature is very important because it helps keep track of whether the specified requirements are correctly implemented throughout the whole development. This also helps in requirements traceability and verification.

Domain Modeling support: It refers to the kind of modeling tools the user is provided with e.g., if it is graphical or not if it gives the possibility to model the static structure of the system's blocks or components. Some of the systems provide modelers with behavior modeling capabilities to specify semantic concepts relating to how the system behaves and interacts with other entities (users or other systems). For instance, OMG-based implementations of UML/SysML inherit all the modeling functionalities which include structural and behavioral diagrams. Additionally, this also includes the tool supports for design through multiple views which in turn is referred to as multi-view modeling support should also be considered.

Testing and verification support: It refers to whether a tool has inbuilt mechanisms to evaluate artifacts before deployment which can be done by conducting different verification checks. As IoT applications are present in our daily life, developing systems that will cause no harm to users in case of more and more sophisticated scenarios should be a priority. To be more specific this feature evaluates

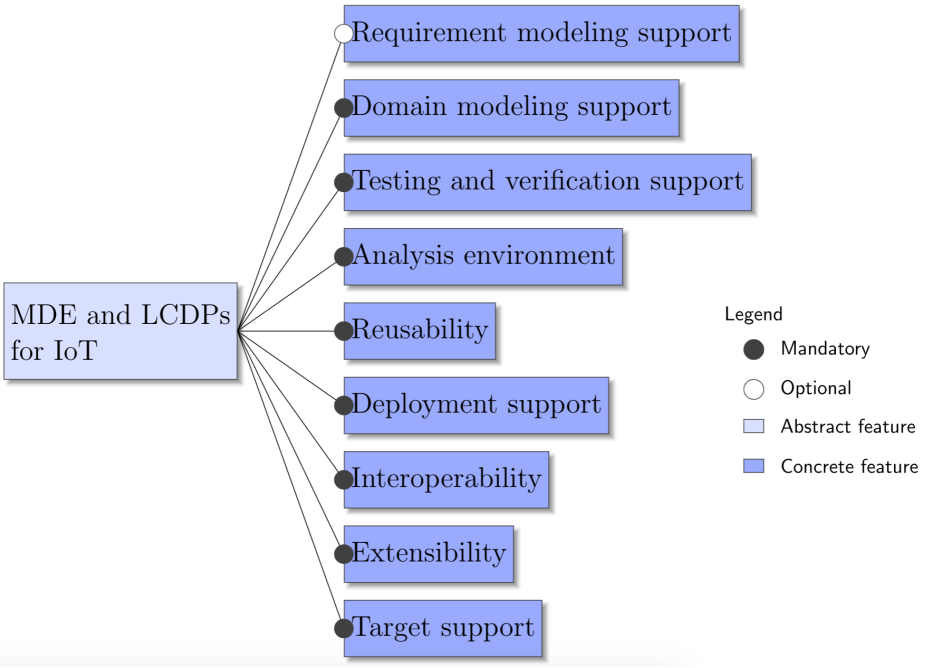


Figure 4.1: Feature diagram representing the top-level variation areas

if the tool support model-based testing, inbuilt model checking, and validation mechanism. This is very important as it ensures the system's correctness and robustness which make the system safe and secure.

Analysis environment: Such a group of features is related to the capability of the considered environment to support different analysis checks for the intended system before its deployment. This can be done on different blocks or components of the system by checking on their responsiveness in case of failure, network loss, security breach, and so on. In this regard, we can feature dependability analysis, real-time analysis, and system quality of service in general.

Reusability: This category illustrates whether the tool under analysis allows the export of artifacts for future reuse. This can be done on developed models or on generated artifacts. Reusability features are also related to the way artifacts are managed e.g., locally or by means of some cloud infrastructure.

Deployment support: It is related to the ways developed systems are deployed and if the generated artifacts are ready for deployment or not. To the best of our knowledge, this should be one of the important features to focus on when implementing an IoT engineering tool. In addition to that, the development tool can be installed locally or on the cloud depending on the client's interest. Finally, factors such as run-time adaptation mechanisms (on modeled or generated artifacts) to respond to the contextual changes are also considered in this category.

Interoperability: This feature examines the ability of a tool to exchange information either internally between components, expose or consume functionalities or information from external services e.g., by means of dedicated APIs.

Extensibility: The tool should provide the means for refining or extending the provided functionalities. In the case of modeling tools, such a feature is related to the possibility of adding new modeling features and notations.

Additionally to the above features, we have added the ***"additional characteristics"*** row to elicit other orthogonal characteristics to the previously discussed elements. In particular, some tools target early phases of development like system design, data acquisition, and system analysis by focusing on the *thing* behavior. Some other tools target the application generation without taking much care of the data acquisition phases which can be done by integrating the developed system with already implemented data source engines, etc. Another peculiar aspect is if the considered approach is available as

	MDE Approaches											*Lowcode development platforms*				
	BRAIN-IoT/ IoTML	*MDE4IoT*	*Papyrus4IoT*	*SysML4IoT*	*UML4IoT*	*ThingML*	*IoTLink*	*IoTSuite*	*FRASAD*	*AutoIoT*	*CAPS*	*NodeRed*	*Atmosphere IoT*	*Simplifier*	*Mendix*	*Salesforce*
Requirement modeling support																
Requirement specification				✓					✓					✓		
Requirement verification				✓					✓							
Domain modeling support																
Graphical user interface	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Structure model	✓	✓	✓	✓	✓		✓		✓	✓	✓	✓	✓	✓		✓
Behaviour model	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓
Multi-view modelling	✓	✓	✓	✓	✓		✓			✓	✓	✓			✓	
Testing and verification support																
Testing environment									✓			✓		✓	✓	✓
Model checking	✓		✓	✓		✓				✓	✓					
Model validation	✓		✓			✓				✓	✓					
Analysis environment																
Realtime analysis																
Dependability				✓												
System QoS				✓							✓			✓		
Reusability																
Allow exporting artifacts	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Allow importing external artifacts												✓				✓
Allow artifact storage and future reuse	✓	✓				✓		✓			✓	✓	✓	✓	✓	✓
Deployment support																
Can be deployed locally	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓
Can be deployed on cloud	✓					✓						✓	✓	✓	✓	✓
Can generate code	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓		✓
Run-time adaptation	✓	✓	✓	✓			✓	✓				✓				
Interoperability																
Support API integration					✓		✓					✓	✓	✓	✓	✓
Support connect to extern data sources	✓				✓		✓					✓	✓	✓	✓	✓
Extensibility																
Easy to add modeling features	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓
Easy to modify the existing features	✓			✓		✓	✓	✓	✓	✓		✓	✓		✓	✓
Additional characteristics																
Focus on "The thing" layer	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓		
Focus on Application layer						✓						✓	✓	✓	✓	✓
Open source to developers	✓		✓			✓						✓				
Target support																
Underlying infrastructure	Papyrus OSGI	Papyrus	Papyrus	Papyrus, Moka	Papyrus	EMFCore Maven	EMF, GMF, EEF, Accelelo, Esper	xText, ANTLR, J2SE	EMFCore GMF	EMF, GMF	EMF,GMF, AMW, ThingML	NodeJS	Java	Node.js, Docker	Java, JS, AWS	Java, Salesforce IoT Cloud
Target platform	IoTboards, cloud servers	IoT boards	MicorEJ, Prismtech h's Vortex	MicroEJ, Edison board	Contiki, Rasp. Pi	IoT Boards, OS, Cloud	Arduino	Android, J2SE platforms	TinyOS, Contiki	cloud	Same target as ThingML	platform agnostic	Mobile, Many IoTBoard	OS, Android	Cloud	Cloud
Code generation language	Java OSGI artifacts	Java, C++	Java	Java	C	C/C++, JavaScript, Java	Java	Java	C, nesC	Python, HTML, CSS, JavaScript	thingml	None	Java,c, arduaino	HTML, CSS, JavaScript	Java, JavaScript	Java

Table 4.1: Taxonomy table

open source or not has an important impact on the possibility for the community to contribute to its development.

4.1.2 Findings

The elicited features, which have been discussed in the previous section, have been considered to study and analyze 16 platforms selected following the above design process. Table 4.1 presents the results from the selection and the assessment process. As it can be seen from such table, we have added the ***Target Support*** to represent to the characteristics related to the infrastructure in which such platform targets. Under such row, we have conceived the *underlying infrastructure* to exhibit the core technologies a tool relies on, *target platform*, which presents different devices and platforms supported by the generated code, and *code generation language* to refer the programming languages supported by the considered system.

According to Table 4.1 most of the analyzed approaches rely on Eclipse and OSGi. We have realized a huge lack of focus on requirement specification except for SysML4IoT (as it extends SysML which enforces requirement specification) and FRASAD, which enforces the requirement specification at the PIM level using rules that can be tracked throughout. The huge lack of analysis support for almost all the tools selected is alarming. We think that it is highly important to analyze and verify the intended system's behavior before deployment as it gives developer indications of what may happen before deployment and help make any adjustment earlier enough. Moreover, we can see that most of the analyzed tools can be deployed locally especially concerning EMF-based tools but mostly all LCDPs are cloud-based with some of them being able to be run also locally.

4.1.3 Limitations

From the above results we have identified the following main weaknesses:

1. **Lack of standards**: we noticed a lack of a standards to support the model-based development of IoT systems. We noticed that each tool proposes its own way of development by hampering interoperability possibilities among different platforms. This is due to the presence of many industrial players, which make the IoT meta-modeling convoluted. On the other hand, different research attempts proposed IoT reference models, which cover different development phases and perspectives. The IoT reference model presented in [49] has been adopted by different tools [61, 155, 157] as a fundamental meta-model. This shows the potentials and benefits of having the availability of standards in such a complex domain. We believe that it as a good starting point, which needs to be further explored to better cover the interoperability dimension (e.g., to enable the possibility of interacting with third-party data resources in general).

2. **Limited support of multi-view modeling:** we noticed that most of the approaches focus on single view modeling. In particular, except for CAPS [72], MDE4IoT [36], AtmosphereIoT [90], and Mendix [124], the analyzed approaches use one specific view to model everything, which is not a good practice in general. Using multi-view modeling presents enormous benefits as it enforces separation of concerns: the system component is designed using a single model with dedicated consistent views, which are specialized projections of the system in specific dimensions of interest [106]. Multi-view modeling is regarded as a complicated matter to address for tailor-made LCDP as they mostly focus on connecting dots aiming at having an application up and running.

3. **Limited support for cloud based MDE:** Moving model management operations to the cloud and supporting modeling activities via cloud infrastructures in general is still an open subject. From our study, we noticed that mostly Low-Code development approaches provide the option to run tools on cloud or on-premise. This is not yet the case of tools based on Eclipse EMF, which still requires local deployments. The research presented in [194] proposed a DSL as a Service (DSMaaS) as a solution to address the reusability of so many created DSL over the cloud. Other attempts like MDEforge [195] aim at realizing cloud based model manipulations [195].

4. **Limited support for testing and analysis:** According to the performed study, very few tools care about the testing and analysis phases of the IoT system development process. There is still a big challenge regarding how to analyze IoT systems responsiveness before deployment. The complexity of the problem relies on the fact that IoT system involve human interaction, environment constraints and we have also to recognize the heterogeneity of the target platforms that makes it hard to depict the kind of analysis properties to address.

4.1.4 Conclusion

In this section, we discussed state of the art on existing approaches supporting the development of IoT systems. In particular, we focused on languages and tools available in the MDE field and the emergent LCDPs covering the IoT domain. The study has been performed in three main steps: first, we conceived a taxonomy consisting of features characterizing the studied IoT development platforms. Then, such features are used to evaluate the functionalities and the services supported by each analyzed platform. As a last step, we identified some weaknesses of the analyzed platforms to pave the way toward an LCDP for developing IoT systems. We have also identified some limitations of already existing approaches and discussed possible ways to improve and address them in the future. In future work, we want to continue the investigation of MDE-based IoT platforms by considering both the quantitative and qualitative aspects of the solutions developed.

4.2 Cloud-based modeling in IoT domain

Cloud-based modeling is one of the relevant topics in the MDE community due to the induced possibilities of designing, developing, analyzing, and deploying applications seemingly with reduced efforts. This has also been recently favored by the increasing adoption of LCDPs. Ideally, domain-specific LCDPs have to run on cloud infrastructures. However, in some industrial settings such as IoT, domain-specific modeling environment tends to be local-based [12]. Nowadays, industries and companies are trying to migrate their modeling infrastructures to the cloud. However, especially in industrial contexts, the existing modeling infrastructures are implemented in complex environments in which the migration cost can be far more expensive and very complicated.

The future of modeling will forcefully be cloud-based [62]. Several initiatives, including Visual Studio Code[1], Eclipse Che[2], Theia[3], and others have shown a lot of potential in shifting modeling environments from local-based and monolithic installations to cloud-based platforms in order to eliminate accidental complexity and expand the variety of available functionalities. Adopting cloud-based modeling will not only attract more citizen developers, but it will unravel a lot of modeling opportunities on different devices such as tablets, and mobile devices [144, 196].

In the IoT domain, modeling and development infrastructures need to consider several heterogeneous aspects of the system's data, communication, and implementation layers. The Web of Things (WoT) paradigm has brought the IoT a step closer to people's perception because it allows treating a networked thing as a Web resource [197]. We think that adopting the concept "Thing-as-a-service" [198] could provide tremendous help in addressing the interoperability issue that exists in the IoT domain [147]. In this case, all the system's sub-components will be modeled as black boxes (services), and they only communicate with well-defined mechanisms (e.g., employing REST APIs). For instance, approaches such as [139, 141, 142, 147] had taken a step toward this modeling paradigm. This section looks at what has been done so far in the IoT domain to support IoT systems' development through cloud-based modeling approaches. In particular, we conducted a thorough investigation to see where the IoT community stands concerning the current trend of moving traditional modeling infrastructures to the cloud. Following an examination of 611 articles, we identified 22 different cloud-based IoT system development tools and platforms. We perform an analysis of the various issues that the IoT community is encountering while implementing cloud-based modeling tools. As a result, we take a deeper look at a few options and discuss the research and development opportunities enabled by adopting cloud-based modeling approaches in the IoT domain.

4.2.1 Related cloud-based modeling studies

We identified very few studies that focus explicitly on cloud-based MDE approaches ([12, 199–201] to mention a few). In this section, we are interested in examining the possible approaches helping in migrating the classical local-based MDE in IoT technologies to the cloud and its adoption.

Our previous study [8] looked at the current state of LCE adoption in the IoT domain. LCE combines LCDPs, MDE, machine learning, and cloud computing to facilitate the application development life-cycle, namely from the design, development, deployment, and monitoring stages for IoT applications. A comparable set of features has been identified by examining sixteen platforms to represent the functionalities and services that each of the investigated platforms could support. We discovered that just 7 of the 16 could be deployed on the cloud, with the majority of them being LCDPs, whereas classical MDE approaches rely on a local-based design paradigm.

In [202], the authors conducted a comprehensive assessment of model-based visual programming languages in general before narrowing their focus to 13 IoT-specific visual programming languages. The research was carried out based on their characteristics, such as programming environment, licensing, project repository, and platform support. According to a comparison of such features, 72%

[1] https://code.visualstudio.com
[2] https://www.eclipse.org/che/
[3] https://theia-ide.org

of open-source projects are cloud-based, whereas only 17% percent of closed-source platforms are cloud-based, which confirms a strong uptrend of cloud-based systems in open-source IoT projects.

In [203], the authors discussed tools and methods for creating Web of Things services, in particular, mashup tools as well as MDE approaches. The techniques regarding expressiveness, suitability for the IoT domain, ease of use, and scalability have been analyzed. Although this study is related to this section, it solely focuses on mashup tools and only includes a few approaches. We can observe from the preceding discussion that only a few techniques attempted to explore cloud-based MDE approaches implicitly. According to this, and to the best of our knowledge, this is the first study analyzing the status of cloud-based modeling in the IoT domain.

4.2.2 Study design

This section aims to analyze how the IoT domain is coping with the trend of moving existing modeling and development infrastructures to the cloud. To this end, we followed the process shown in Fig. 4.2 according to the methodology presented in [204]. In particular, the search and selection process was mainly conducted in four main phases. In the first phase, we formally and explicitly represented the problem to get a head start on the search. Second, we defined a search string and selected well-known academic search databases. Third, we performed a search to gather approaches to answer properly defined research questions. Fourth, we narrowed down the potential approaches and mapped them based on their similarity and variability. Finally, we analyzed the collected approaches and elaborated some recommendations on the identified difficulties.

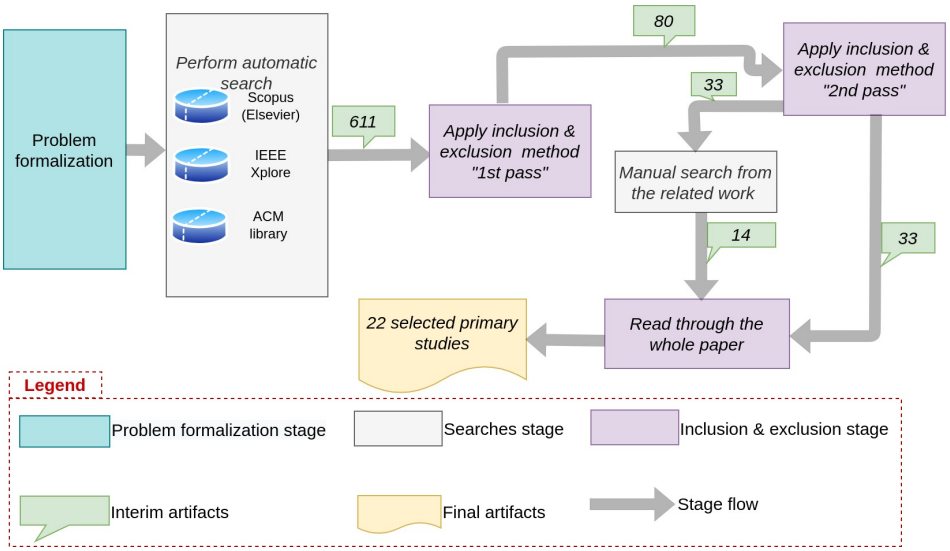


Figure 4.2: Search and selection process

Phase 1: Problem formalization: This phase mainly focused on formalizing the problem we wanted to solve by looking at the current MDE tendency. One of the sources of inspiration for this study was the work in [62] which addresses the topic of *"What is the future of modeling?"*. Thus, we came up with the formulation of the following research questions:

- ***RQ1:*** *How is the IoT community adopting cloud-based modeling approaches?*
- ***RQ2:*** *What challenges do researchers face when developing cloud-based IoT modeling and development infrastructures?*
- ***RQ3:*** *What are the main potential opportunities laying ahead for future researchers and developers in the IoT domain?*

Phase 2: Automatic search: In this phase, we applied a search string to different academic databases, i.e., Scopus (Elsevier)[4], IEEE Xplore[5] and ACM library[6] by limiting the search on the last 10 years. The query string we used for the automatic search was: `("MDE" OR "Model Driven Engineering") AND ("IoT" OR "Internet of Things") AND ("Cloud" OR "Web").`
Table 4.2 shows the number of approaches we managed to collect in this phase.

Table 4.2: Results table

Database	Results
Scopus (Elsevier)	**233**
IEEE Xplore	**263**
ACM library	**115**
Total	**611**

Phase 3: Inclusion & exclusion, 1st pass: Table 4.2 shows that 611 publications were initially discovered from different sources. At this point, we have reviewed the approach's title, keyword, and abstract and exclude approaches that were not satisfying the following criteria:

- Studies published in a peer-reviewed journal, conference, or workshop.
- Studies written in English.
- Approaches that focus explicitly on the IoT topic.
- Studies that propose a cloud-based modeling approach, either explicitly or implicitly.

At the end of this point only 80 approaches were deemed to be satisfying the above set criteria and were considered for the next phases.

Phase 4: Inclusion & exclusion, 2nd pass: In this phase, we read the introduction and the conclusion of the approaches previously collected. We also removed some duplicates. Various documents were rejected during this phase for a variety of reasons, for instance, because the presented approach is not explicitly offering an IoT-based cloud-based development environment. At the end of this phase, we ended up with 33 documents. Furthermore, via the 33 papers, we conducted a more in-depth manual search of potentially related work referenced by them, in which 14 approaches were selected and manually added, bringing the total to 47.

Phase 5: Reading of the whole approach text: We've gone over the entire articles in this phase, focusing on the proposed approaches and their evaluation sections. Several documents were discarded because of different reasons. For instance, approaches that presented hybrid solutions (e.g., enabling local modeling with the possibility of storing models on remote repositories) were discarded. In addition, the approaches that claim to build web-based IoT data-wrangling platforms by reusing existing IoT data storage platforms were also discarded. Finally, we selected 22 documents that leverage a cloud-based modeling environment to design, develop, or deploy IoT applications.

Figure 4.3 shows the distribution of the selected approaches with respect to their corresponding sources. As you might notice from Fig. 4.3, a portion of the selected approach (4 out of 22) was found from manual snowballing process. In the next section, the research questions presented in Sec. 4.2.2 are answered singularly by analyzing the research approaches that have been collected as previously described.

[4] https://www.elsevier.com/
[5] https://ieeexplore.ieee.org
[6] https://dl.acm.org/

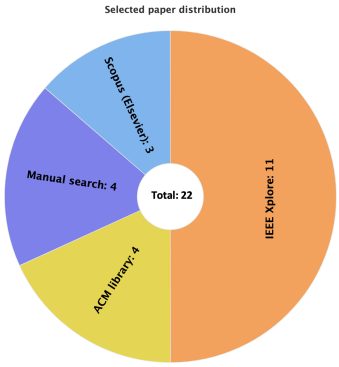


Figure 4.3: Selected approach distribution

4.2.3 Findings

In general, cloud-based modeling is not a new topic in terms of demand and market viability, but when it comes to the IoT domain, there are few approaches in research. In contrast to LCDPs, which have largely adopted cloud-based methods even in the IoT domain, the move from traditional local-based MDE practices to the cloud is still in its infancy. This section goes over different cloud-based modeling approaches that target the IoT domain. We organized the analyzed approaches into three categories according to their main focus of interest i.e., modeling IoT structural aspects, service-oriented approaches, and deployment orchestrations. The aim is to answer the research question ***RQ1:*** *How is the IoT community adopting cloud-based modeling approaches?*

As previously presented, several approaches are available to support cloud-based modeling in the IoT domain. Table 4.3 shows an overview of the analyzed approaches; half of them are concerned with structural issues, whereas only a few deal with deployment concerns. The current state of the art suggests that there is no predominant common language, although the graphical syntax is preferred. In terms of technical needs, textual cloud-based modeling environments might be simpler to adopt as opposed to graphical ones.

To assess the tool's source-code accessibility, the open-source status factor was chosen. This is a key factor that contributes to the tool's scalability because more individuals can access the source code and potentially extend it. This goes hand in hand with the criterion for determining whether the tool is still accessible. The relationship between the two parameters is depicted in Figure 4.4. We can observe that practically all of the tools that are not open-source are also not currently accessible. When looking at industrial settings, this is especially true when it comes to internal proprietary tools. The same can be said for open-source tools, with the majority of them being freely available. Most of the analyzed approaches are supported by tools, which are not open source. This goes hand in hand with the public availability of the methodologies. We can observe that all the tools that are not open-source are also not publicly accessible. When looking at industrial settings, this is especially true when it comes to internal proprietary tools. The same can be said for open-source tools, with the majority of them being freely available.

While analyzing each approach, we also looked at the supporting infrastructures and their ability to generate deployable artifacts. In this regard, we have discovered that JavaScript-based environments like Node.js and Angural.js are widely used for tool development. This might be due to the fact they are among the modern languages for front-end technology implementation. On the other hand, it appears that the majority of techniques generate artifacts, even though few of them are standalone deployable components. It is also worth noting that the generated deployable artifacts can only be deployed within the same original environment in most of cases. To ensure interoperability, scalability, and reusability of the tools, the generated artifacts should generally be deployed anywhere.

In traditional local-based modeling environments, the aforementioned evaluation factors also apply. We chose not to include more complex evaluation criteria such as tool extensibility, scalability, analysis, model verification and validation, and so on. This is due to the fact that the cloud-based

Table 4.3: Analyzed approaches

Tool name	Category	Language syntax	Open-source	Tool availability	Underlying infrastructure	Generated artifact
DSL-4-IoT	Structure	Graphical	no	no	js, OpenHAB	JSON config
BIoTA	Structure	Graphical	no	no	Apache Tech. GraphQL	YAML file
IADev	Service	Textual	no	no	ASR,REST,ATL	REST app
Node-RED	Structure	Graphical	yes	yes	Node.js	Node-RED app
AutoIoT	Structure	Graphical & textual	no	no	Python, js	Flask app
[135]	Structure	Textual	no	no	Smart-M3	–
AtmosphereIoT	Structure	Graphical	no	yes	Multi-platform	Multi-platform apps
[136]	Structure	Textual	yes	yes	js,VueJS	UI code
[205]	Structure	Graphical	no	no	WebRatio, IFML	UI code
FloWare Core	Structure	Graphical	yes	yes	JavaScript	Node-RED Config file
Vitruvius	Structure	Textual	yes	no	XML,HTML,js	HTML5 with JavaScrit app
MIDGAR	Service	Graphical	no	no	Ruby, js, HTML Java	Java app
LogicIoT	Service	Textual	no	no	JSP	-
glue.things	Service	Graphical	yes	no	AngularJS,Meshblu PubNub	NodeRED service
[143]	Service	Textual	no	no	Firebase&Node.js	–
TOSCA	Deployment	Textual	yes	yes	Multi-platform	Config files
[144]	Service	Graphical	no	no	-	-
E-SODA	Service	Textual	yes	no	OSGI cloud	OSGI java bundles
[146]	Service	Textual	yes	yes	ClickScript,AJAX	REST services
InteroEvery	Service	Graphical	no	no	Spring Boot,Rest RabbitMQ,Angular	–
DoS-IL	Deployment	Textual	no	no	js,HTML,DOM	Config files
GENESIS	Deployment	Textual	no	no	multi-platform	Genesis dep. agents

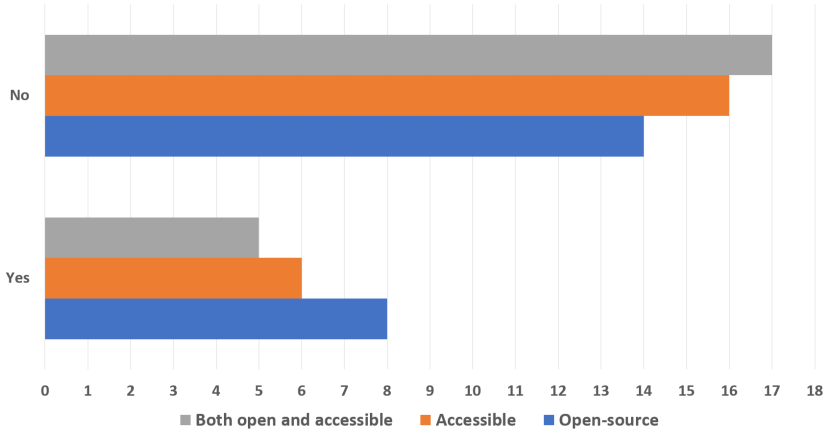


Figure 4.4: Accessibility vs Open

modeling topic is still in its early phases. Nonetheless, we've discovered that with the exception of NodeRED [89], which partially implements some of the above functionalities, none of the other tools clearly express their support for it. This illustrates the magnitude of the task that needs to be addressed. In this regard, we recognize that developing the modeling environment (whether graphical or textual) is critical and that once this is done, other services can be migrated to the cloud and used via a consumable API.

4.2.4 Open challenges

Multiple issues have arisen as a result of the expansion of connected smart and sensor devices, as well as the increased usage of cloud-based models [66]. As a typical IoT system consists of multiple complex sub-systems, having an all-in cloud-based environment can become even more complicated. On the other hand, overcoming these barriers is worth the effort because it opens up more opportunities. This section elaborates on the current challenges IoT systems face while developing and integrating such tools in a cloud-based environment. Essentially, we are answering the research question ***RQ2****: What challenges do researchers face when developing cloud-based IoT modeling and development infrastructures?*

Extensibility mechanisms: Extensible platforms allow the addition of new capabilities without having to restructure the entire ecosystem. Because IoT systems are distributed, a typically recommended architecture would be to use the micro-service architecture throughout the development process [85]. Aside from that, IoT systems may require additional interactions with third-party technologies. As a result of the previous scenario, developing tools to design and develop such distributed applications on the cloud need efficient tools that traditional domain specialists may not have. Accessibility mechanisms are presented through tools like [89, 137], but there is still a lot to be done. Currently, domain experts must provide cloud-based automation mechanisms and tools to allow citizen developers to add new features without requiring sophisticated knowledge or changing existing architectures.

Heterogeneity: It is an important challenge of the IoT domain, which involves different players developing various applications running at different layers, namely the edge, fog, and cloud [85]. In addition, deployments and data consumption methods are very diverse, increasing the complexity of traditional code-centric approaches [206]. Cloud-based modeling in IoT brings even more sophistication regarding the environment in which the system should be designed and developed. The typical cloud-based modeling platform should foster the integration of heterogeneous technological implementations, promoting reusability and developing solutions close to the problem domain. Approaches such as [139, 140, 167] have presented different strategies to tackle such issues, but much more have to be investigated.

Scalability: IoT systems are expected to handle a wide range of users, perform demanding computations, and share enormous amounts of data among nodes. Therefore, supporting cloud-based modeling approaches must be implemented in such a way that scalability concerns are mitigated. One of the approaches to tackle such challenges is to adopt container-based orchestration tools such as Kubernetes. The use of such tools can offer out-of-box features such as self-healing, fault-tolerance, and elasticity of containerized resources [86]. This will also help automate cognitive processes that can detect scalability needs and adjust autonomously without human intervention.

Interoperability: The interoperability of various tools, services, and resources is critical in the IoT domain. The interoperability of cloud-based modeling platforms, particularly in the IoT area, is currently limited since different tools run in different environments and have different natures. A tool like [147] promotes the micro-service architecture by allowing all parts of the system to communicate with each other. Several regulations, such as standardization, will need to be implemented to achieve interoperability among different cloud-based modeling environments. To address interoperability concerns, technologies like [89, 133, 137, 160] promote a common format based on JSON to encode models. It is worth noting that adopting Model-as-a-Service (MaaS) architectures could also promote the interoperability of services and artifacts.

Learning curve: It is not easy to find professionals who can master and combine the different sophisticated technologies involved in developing and managing IoT systems. IoT domain experts may lack modern programming expertise, whereas experienced software programmers may lack modeling domain expertise. For instance, conceiving a cloud-based code generator requires understanding different model transformation techniques and particular programming abilities; Implementing a visual mashup tool will necessitate knowledge of modern languages such as JavaScript, HTML, and CSS.

Security concerns: Current IoT systems suffer from security concerns as data are collected from a

wide distribution of private and public nodes. Furthermore, the data is transferred using remote IoT gateways, which might get exposed in the process. This heterogeneity of secured and unsecured data might favor attackers to target devices and compromise the integrity of data and operations [207]. Therefore, proper abstractions and automation techniques are needed to help target users that might not necessarily have the required knowledge of the security practices to be employed.

4.2.5 Opportunities

In this section, we examine several opportunities that we think researchers and developers can leverage to improve the cloud-based development and management of IoT systems. Therefore, we aim at answering the research question ***RQ3:*** *What are the main potential opportunities laying ahead for future researchers and developers in the IoT domain?*.

Tools and platforms

Numerous tools and platforms are being built to tackle cloud-based modeling concerns. Thus, now is the right moment to suggest powerful and extensible tools that the IoT community may harness to solve their domain-specific issues. In this section, we look at various open-source and highly extensible platforms that are popular among the modeling community and that we would recommend for the IoT domain.

Cloud-based development tools based on Eclipse: We believe that a significant part of the MDE community, or at least for research purposes, uses Eclipse-based technologies. This is because most Eclipse projects and technologies are open-source, making them more accessible and encouraging individuals to participate. As of March 2021, the Eclipse Foundation hosts over 400 open source projects, 1,675 committers, and over 260 million lines of code have been contributed to Eclipse project repositories [208]. Through the Eclipse Cloud Development (ECD)[7] effort, the Eclipse community has demonstrated its willingness to transit a part of its ecosystem to the web. Eclipse's ECD Tools working group strives to define and construct a community of best-in-class, vendor-neutral open-source cloud-based development tools and promote and accelerate their adoption. Some of the best cloud-based technologies that the IoT community can benefit from are the following:

- *EMF.cloud, GLSP, Theia* - Independently from the Eclipse modeling framework (EMF), the EMF.cloud community recently expressed a strong desire to migrate the Eclipse-based modeling infrastructure to the cloud. This project aims to develop a web-based environment for creating modeling tools that can support the editing mechanisms of EMF-based models. EMF.cloud allows users to interact with models through the EMF.cloud model server, which coordinates the use of GLSP for graphical modeling, and LSP for textual modeling. Code generation infrastructures based on Eclipse Xtend are also included, while Eclipse Theia provides a web-based code editing and debugging infrastructure. Several resources are available in the community for extending those tools, and we believe that IoT developers may use such technologies to construct cloud-based IoT DSLs.

- *Sirius Web*[8] - It is an Eclipse Sirius-based modeling tool that provides a powerful and extensible graphical modeling platform for users to design and deliver modeling tools on the web. In Sirius Web, the ability to create your modeling workbench in a configuration file is supported. In this case, no code generation is required because everything is interpreted at run-time [209]. Furthermore, being open-source, Sirus Web provides greater accessibility and customizability than the desktop version, making it easier for the IoT community to get started with their cloud-based solutions.

Another alternative, such as Eclipse Che[9] makes Kubernetes development accessible for developer

[7] https://ecdtools.eclipse.org/
[8] https://www.eclipse.org/sirius/sirius-web.html
[9] https://www.eclipse.org/che/

teams. Che is an in-browser IDE that allows you to develop, build, test, and deploy applications from any machine. Finally, Epsilon playground[10] has been recently launched to offer cloud-based tools for run-time modeling, meta-modeling, and automated model management.

Low-Code Development Platforms: Looking at the LCDPs, the only powerful cloud-based open-source platform we would recommend is Node-RED [89]. Due to its high extensibility and accessibility, Node-RED offers an excellent IoT system mashup environment in which IoT systems can be designed, developed, and deployed on the fly. The Node-RED platform is open, and IoT system developers can build their custom nodes, compile, test, and deploy them in the Node-RED ecosystem. Several extensions have been made, such as [210] tackling the reusability issues in cloud-based modeled components, [211] to tackle the heterogeneity and complexity challenges found in the Fog based development. Finally, in [212], the authors presented SHEN to enable self-healing capabilities of applications based on Node-RED. In terms of interoperability, Node-RED models are represented as JSON objects, which any third-party tools can easily consume. Some of the tools in this domain, such as FloWare [137] and GENESIS [168] already support the Node-RED models, which shows a great sign of its high impact. Table 4.4 outlines the essential characteristics of the recommended platforms.

Table 4.4: Recommended technologies

	EMF.cloud	**GLSP**	**Theia**	**Che**	**Node-RED**
Open-source	✓	✓	✓	✓	✓
Extensible	✓	✓	✓	✓	✓
Scalable	✓	✓	✓	✓	✓
IoT-specific	—	—	—	—	✓
Application	Web-based EMF modeling tools	Graphical language-server-editor	Web-based code editor	Kubernetes-native IDE for DSL deployment	Flow-based programming

Benefits of cloud-based modeling

Although there are difficulties in adopting a cloud-based modeling approach in the IoT domain, various opportunities will emerge, making the investment worth it. This section outlines various opportunities that will emerge once cloud-based modeling is widely adopted in the IoT domain.

1. **User communities:** Adopting cloud-based modeling in the IoT domain will have the potential of attracting more citizen developers, and it will unravel a lot of modeling opportunities on different devices such as tablets and mobile devices [144, 196, 213].

2. **Collaborative modeling:** Once IoT modeling infrastructures are moved to the cloud, it can be necessary to introduce collaborative modeling features to simplify the interaction of both developers and stakeholders. Unfortunately, none of the discussed approaches provide collaborative modeling functionalities. However, collaborative modeling can harness the power of real-time information, artifacts, and service exchange.

3. **Productivity:** Empowering users to develop their applications by embracing cloud-based modeling is a head-start toward high production and reducing time to market [12]. Users can create applications that cater to their problems, and engineers focus on developing features that facilitate the user for a smooth development at an appropriate abstraction level. In addition, the participants will focus on problem-solving in their particular domain and avoid wasting time and resources on solving problems that are outside their competencies.

[10] https://www.eclipse.org/epsilon/live/

4. **Maintenance:** Traditional code-centric methodologies necessitate a significant investment in the ongoing maintenance of developed systems. In addition, systems require regular upgrades and installations, which can be error-prone and time-consuming. During upgrades or troubleshoots times of the system, sometimes system downtime is necessary, impacting production. Furthermore, the growing need for software systems in our daily lives and constantly changing user requirements required an agile approach for addressing these issues quickly without compromising system availability or user access. In many cases, such challenges are handled by cloud providers, leaving developers and engineers to focus on developing applications that directly impact customer demands [214].

5. **Monitoring and debugging:** Cloud-based modeling enables monitoring of activities and their archive through its cloud providers. This is a head-start when debugging distributed applications because developers can track down the microservices, which are the root of the detected problems. Without appropriate cloud infrastructures, it would be challenging to solve these issues, even with features such as self-healing and repair strategies. Current cloud-based solutions come bundled with monitoring tools that assist in problem diagnosis and monitor the usage of the applications.

4.2.6 Conclusion

To develop IoT applications, developers must overcome various challenges, including heterogeneity, complexity, and scalability. Moving development infrastructure to the cloud will open up plenty of new opportunities regarding accessibility, productivity, maintenance, and monitoring. In this section, we conducted a systematic study to assess the current state of the art on cloud-based modeling approaches in the IoT domain. We looked at 22 approaches proposing cloud-based modeling environments in the IoT domain. The considered approaches have been analyzed to assess their strengths and weaknesses concerning many characteristics, including their modeling focus, accessibility, openness, and artifact generation. Throughout the section, we have discussed many challenges that IoT developers encounter while adopting such tools. We also discussed various generic technologies and tools, which can be adopted in the IoT domain.

Chapter 5

Assessing the quality of IoT Engineering Platforms

Over the last few years, industry and academia have proposed several Low-Code and MDE platforms to ease the engineering process of IoT systems. However, deciding whether such engineering platforms meet the minimum required software quality standards is not straightforward. Software quality can be defined as the degree to which a software system achieves its intended goal. Various software quality standards have been established to aid in the software quality assessment process; however, due to the nature of engineering IoT platforms, such models may not entirely suit the IoT domain. This chapter presents a model for assessing the software quality of Low-Code and MDE platforms for engineering IoT platforms. Tackling the second research problem (RP2), the proposed software quality model was based on and extends the ISO/IEC 25010:2011 software product quality model standard. It is intended to assist IoT practitioners in assessing and establishing quality requirements for engineering IoT platforms. To determine the effectiveness of the proposed model, we used it to evaluate the quality of 17 IoT engineering platforms, and the results obtained are promising.
This chapter is structured as follows: The background of the study is presented in Sect. 5.2. The proposed product quality model is presented in Section 5.3. In Section 5.4 we go through the selected primary studies and present the followed evaluation process. In Section 5.5, we present the results from the conducted assessment by reflecting on specific research questions. Section 5.6 discusses the results as well as its limitations. Section 5.7 concludes the chapter and discusses perspective future work.

5.1 Introduction

IoT systems offer enormous benefits in our daily lives by enabling seamless communication with our surroundings. Such systems demand many development skills, from handling tiny microcontrollers to more extensive and complex cloud-based systems. In the IoT domain, systems often include safety-critical tasks that, if mishandled, might have disastrous consequences and even cost human lives [34]. To guarantee robustness and safety during operation, it is critical to investigate the correctness and the quality of the platforms and the process by which systems are developed.

Over the last few years, both academia and industry have proposed novel languages and tools to support the engineering of IoT systems. To this end, MDE and Low-Code paradigms are employed to conceive engineering platforms specific to the IoT domain. In the software engineering process, MDE promotes the use of models as first-class citizens in software development [42]. Its goal is to improve productivity and reduce time to market by allowing the development of systems using models defined with concepts that are much less linked to the underlying implementation technology and much closer to the problem domain [35]. LCDPs are generally cloud-based software development platforms that leverage a Platform-as-a-Service paradigm to enable users with little or no programming skills to create fully functional apps with dynamic graphical user interfaces [11, 87, 215]. For the sake of

readability, we use *IoT Engineering Platforms* when we need to refer Low-Code and MDE platforms indistinguishably.

Deciding whether an IoT engineering platform meets the required standards for adoption in terms of quality is not a straightforward process as it involves considering and exploring various sources of information. MDE and low-code development technologies frequently rely on rigorous verification and validation processes conducted under predefined arbitrary constraints to guarantee the quality of generated systems. However, in most cases, the quality of such employed engineering tools is not taken into account seriously or simply overlooked [184, 185].

Practitioners typically rely on well-established standards and practices to improve confidence in whether a system or a product fits the wanted quality requirements. The ISO/IEC 25000 to ISO/IEC 25099 series of International Standards, titled *"Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQUARE)"*, aims to address issues concerning software quality requirements specification and evaluation [189]. The ISO/IEC 25010:2011 standard, in particular, [14], is intended to help people who are interested in developing or purchasing systems and software products to specify and evaluate their quality requirements.

This chapter presents a quality model for assessing the quality of IoT engineering platforms based on the ISO/IEC 25010:2011 standard. In doing so, we enhanced the standard's product quality model by defining a product quality assessment model that is more suitable for the IoT domain. Initially, the ISO/IEC-25010:2011 standard, which replaced ISO/IEC-9126-1:2001 standard, defines two main sets of quality models, namely ***"quality in use model"***, and ***"product quality model"***.

The former is related to the outcome of interaction when a product is used in a specific context. In contrast, the latter is related to software platforms' static and dynamic properties. These models are defined in terms of characteristics, with some of them further subdivided into sub-characteristics [14]. Both models are equally crucial for thoroughly assessing the quality of a given platform. However, the proposed model focuses on the *product quality model* since it emphasizes on platform's technical quality rather than using quality which can be challenging to measure for platforms in their early development stages.

Even though all of the approaches presented in 3.5 relied on the ISO/IEC-9126-1:2001 standard like ours, none specifically target IoT engineering platforms. While there are many quality assessments of DSLs available, only a small number of them refer to an established standard in the evaluation [184], and we could not find any that addresses the IoT modeling domain in particular. As a result, we are confident that our study is the premier to use a well-established quality standard to evaluate the product quality of IoT engineering platforms.

To evaluate the effectiveness of the proposed model, we employ it to assess the quality of 17 IoT platforms selected from our previous studies [8, 13]. We present the methodology we used to choose such platforms, perform the quality assessment, and subsequently present and discuss the obtained results. We summarize this chapter's contribution as follows:

- We propose a quality model based on the ISO/IEC 25010:2011 standard for evaluating the quality of IoT engineering platforms;
- We assess the quality of 17 IoT engineering platforms by relying on the proposed model;
- We present and discuss the findings as well as the limitation of the study in order to validate the effectiveness of the proposed model.

5.2 Overview on Software Quality Models

The discipline of Software Quality Engineering [216] is concerned with improving the approach to software quality. However, the various perspectives present throughout the software life cycle show what constitutes software quality is frequently contested. In this context, relying on software quality models to support the quality management of software systems is widely accepted [217]. A

quality model is typically defined as a set of sub-characteristics and their interrelationships that serve as the foundation for specifying quality standards and evaluating quality [218]. Various standards have been defined in the literature to assist in the evaluation process. The ISO/IEC 9126:1991 standard aimed at defining a software quality paradigm and a set of guidelines for assessing the characteristics associated with it [219]. This standard was later revised by ISO/IEC 25010:2011, which included a model for software system quality with well-formed specifications for quality characteristics of software products [184].

The ISO/IEC 25010:2011 standards help design, develop and acquire systems and software products with the specification for evaluating their quality requirements [189]. The standards are made up of two quality models, each with its own set of characteristics, some of which are further subdivided into sub-characteristics. First, the *"quality in use model"* focuses on the outcome of interaction when a product is used under particular contexts [14]. This model is primarily intended to provide targets for promoting the development and verification efficiency, as well as to anticipate quality in use before delivery [189]. This model includes five characteristics, namely *effectiveness, efficiency, satisfaction, risk freedom, and context coverage* where some of them are further subdivided into nine sub-characteristics. The *"product quality model"* refers to the static and dynamic qualities of a software platform. This model primarily focuses on providing assessment ground for people supplying software products and those acquirers who wish to get more involved in the process technically [189]. The model is divided into eight characteristics, namely *functional appropriateness, performance efficiency, compatibility, reliability, usability, security, maintainability, and portability* which are further elaborated into 31 sub-characteristics.

In the past, the ISO/IEC 25010:2011 standard has been adopted to assess not only the product quality of IoT systems [18–20] but also in domains such as Big data [21], Machine Learning [22], Software Product Lines (SPL) [23], Customer Relationship Management (CRM) systems [24] and mobile apps [25], just to mention a few. In the MDE world, approaches such as [187] relied on it for assessing the performance of MDE quality studies, while [184, 185] adopted it in order to assess the quality of Domain Specific Languages (DSLs) while and design architectures quality [186] adopted it to assess the quality of design architectures. Although ISO/IEC 25010:2011 cannot be considered a one-size-fits-all solution, it can be a beneficial approach for evaluating the quality of IoT engineering platforms. A detailed description of the mentioned quality characteristics is given in the next section.

5.3 The product quality model

In this section, we discuss the *product quality model* of the ISO/IEC 25010:2011 standard to enable the quality evaluation of IoT engineering platforms. To this end, by referring to Figure 5.1, in the following, we discuss the common model characteristics concerning the peculiarities of the IoT domain.

5.3.1 Functional suitability

This factor evaluates how well a platform meets specified and implied objectives when used in specific contexts. In the IoT engineering context, its corresponding sub-characteristics can be defined as follows.

Functional completeness is the extent to which a given platform supports the design of all the layers that comprise an IoT ecosystem. For instance, the platform's capacity to represent the complete IoT ecosystem from the edge, fog, to the cloud layers while seemingly handling all of the communication heterogeneity that might occur. Concerning *Functional Correctness*, this can be defined as the extent to which the platform applies specific correctness methodologies during the development phases, including but not limited to correct by construction, model-checking, model validation, and rule-based modeling. Finally, the *Functional appropriateness* refers to the amount to which the given functionalities facilitate the completion of defined activities and objectives. IoT engineering platforms

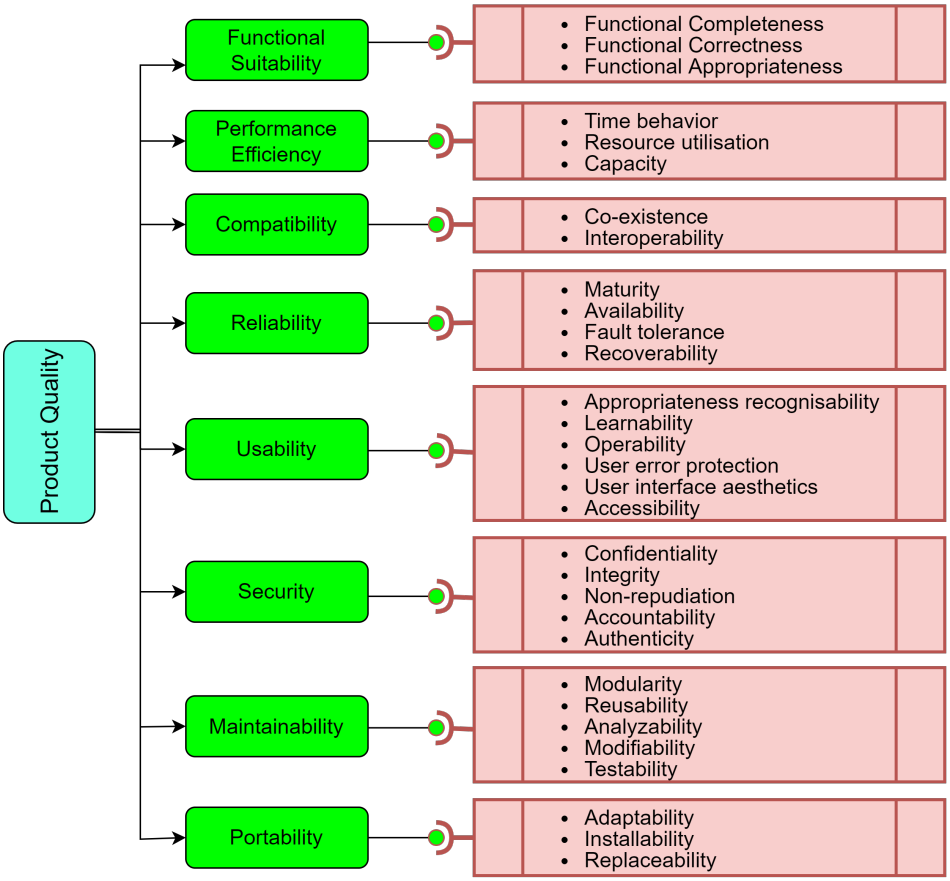


Figure 5.1: Software Product Quality Model

have diverse functionalities; in this case, we can propose to analyze how the specified correctness methodology permits achieving the goal.

5.3.2 Performance efficiency

This characteristic determines how the platforms consume resources under specific conditions. This characteristic is mainly related to run-time attributes of the underlying system infrastructure, which in some situations cannot be predicted accurately. In the context of IoT engineering platforms, the corresponding sub-characteristics can be defined as follows: *Time behavior* refers to how well a platform function meets its criteria in terms of response and processing times, as well as throughput rates. The platform's responsiveness can primarily indicate this, and it is worth noting that it can only be evaluated on platforms that are accessible.

Resource utilization measures how much and what kind of resources are used to suit the platform's needs. Although this can somewhat be predicted when a platform runs on well-known underlying infrastructures, the actual utilization can only be evaluated accurately during the platform's usage. Finally, the *Capacity* is defined to assess the platform's ability to model and coordinate a large and complex system that spans multiple IoT sub-domains.

5.3.3 Compatibility

This characteristic implies the platform's capacity to interchange data with other platforms. The two sub-characteristics are defined as follows: *Co-existence* refers to the degree to which a platform continues to operate efficiently while sharing a shared environment and resources with other platforms without hammering the other platforms. *Interoperability* is referred to the degree to which a platform

can exchange information with other platforms while in use. In our context, this can be assessed in terms of the platform's capability to support the data exchange during the development process by consuming or sending information to/from external services via dedicated means such as REST APIs.

5.3.4 Reliability

This characteristic reflects how a platform can complete a set of tasks in a given amount of time. Its sub-characteristics can be defined as follows: *Maturity* reflects how well the platform supports all of the basic functionalities of a typical engineering platform, such as design, code generation, and deployment. *Availability* is concerned with the extent to which a platform is operational when needed. *Fault tolerance* is the degree to which a platform performs as expected despite the presence of hardware or software faults. In our case, this can be evaluated as the platform's capability to support advanced mechanisms, including but not limited to self-adaption and self-healing. Finally, the *Recoverability* reflects the extent to which a platform, in case of interruptions or failures, can recover data directly affected and re-establish the desired state of the system. To that end, the platform should be able to handle the mechanisms including but not limited to self-recovery or self-redeployment, etc.

5.3.5 Usability

This characteristic reflects how well specific users can use a platform to achieve particular goals with efficiency, effectiveness, and satisfaction. The sub-characteristics are as follows: *Appropriateness recognizability* is defined as the degree to which users can determine whether a platform is appropriate for their needs. This can be promoted in the IoT engineering world by the degree to which the platform is customized concerning the underlying environment. *Learnability* can be defined as the extent to which the platform aids developers in learning how to use it, e.g. with context-based modeling support, on-the-fly suggestions, and so on.

Operability represents the extent to which a platform has attributes that make the entire development process more accessible, for instance, using functionalities like auto-completion for textual languages, guide-through mechanisms, multi-view modeling, palette show/hide, and palette element search. *User error protection* reflects the extent to which the platform provides some protection to avoid errors, such as static analysis or on-the-fly error handling. *User interface aesthetics* is the extent to which the platform provides pleasant and satisfying user interfaces. Finally, *Accessibility* can be considered as to whether the platform is easily reachable in case of needs, either being locally or online.

5.3.6 Security

This characteristic refers to how successfully a platform protects information and data so that users, services, and external systems have appropriate access to data according to specific authorization levels. Its six sub-characteristics have been renamed as follows: *Confidentiality* refers to the extent to which a platform assures that data is exclusively available to those who have been granted access. *Integrity* can be defined as the extent to which an IoT platform prohibits unwanted access, modification of the platform, or data.

Non-repudiation can be measured by the extent to which the platform enables methods for recording all actions conducted during development and proves that they have been performed so that they cannot be contradicted later. *Accountability* can be measured as the extent to which a platform enables tracing attributes like versioning, historical action retrieval, and so on. Finally, *Authenticity* can be described as the ability of the platform to support different types of authentication mechanisms while accessing platform resources.

5.3.7 Maintainability

This characteristic refers to the degree to of a platform to be improved, repaired, or adapted to changes. Its corresponding sub-characteristics are defined below: *Modularity* can be measured as the extent to which a platform is decoupled into discrete sub-systems that can be modified independently of other parts. *Reusability* reflects the extent to which a platform or a component of a platform can be used in more than one system. While evaluating this, we can focus on the platform's ability to be decomposed into small reusable sub-systems.

Analyzability assesses the efficacy and efficiency with which it is feasible to determine the impact of modifying parts of the platform by either removing it or injecting failures into it. In the IoT engineering context, whether the platform provides means supporting the analysis of the system under development and/or the platform itself is also looked at. In terms of *Modifiability*, the assessment can be performed on the extent to which a platform can be successfully and efficiently modified without introducing flaws or deteriorating the quality of existing products. Finally, *Testability* can be assessed in terms of the capability of the platform to provide testing supports for its components or the system under development.

5.3.8 Portability

This characteristic refers to how fast and successfully a platform can be moved from or to different hardware and software operational environments. Its corresponding sub-characteristics are defined below *Adaptability* refers to the extent to which a platform can be effectively and efficiently adapted to different environments. *Installability* reflects the degree of effectiveness and efficiency with which a platform can be successfully installed and/or uninstalled in a given environment. *Replaceability* measures the extent to which a platform can be updated, replaced, and redeployed in the same environment and still performs as expected.

5.4 Quality assessment of IoT engineering platforms

This section shows the quality assessment process followed to analyze 17 IoT engineering platforms using the product quality model discussed in the previous section. The platforms of interest have been identified as presented in Section 5.4.1. The research questions that we answered through the performed evaluation are presented in Section 5.4.2. The quality assessment process that has been followed is presented in Section 5.4.3.

5.4.1 Selection of the evaluated IoT engineering platforms

In [8], we examined 16 different platforms to gain a better understanding of the current state of the art in supporting the development of IoT systems, with a focus on languages and tools available in the MDE field and emerging LCDPs. In [13], we examined 22 IoT modeling environments, assessing their strengths and weaknesses in terms of cloud-based modeling capacity, accessibility, openness, and artifact generation. Combining the two data sources, we started with a total of 38 approaches, which have been filtered by considering the following exclusion criteria:

- Approaches that were published before 2012 have been discarded;
- Duplicated approaches have been removed;
- Approaches that do not permit the development of fully functional IoT applications have not been considered.
- Approaches that depend on already-included platforms have been filtered out;

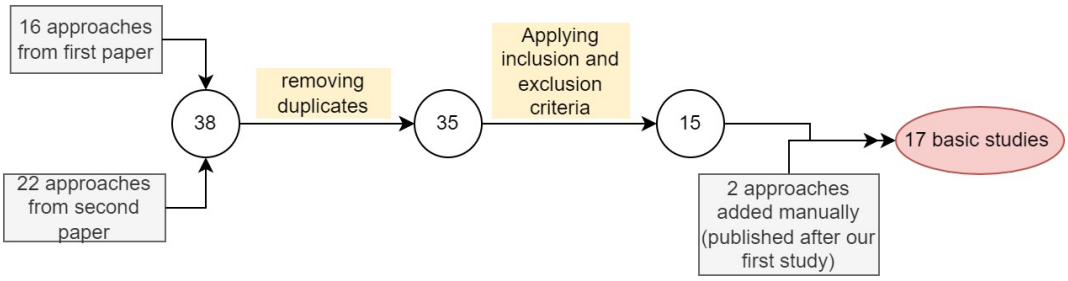


Figure 5.2: Primary studies selection process

- Approaches that are generic and that do not explicitly target the IoT domain have not been considered in this study.

By applying such criteria, 15 out of the initial 38 IoT development platforms were identified. In addition, we have added two more approaches that we believe are very promising and were published after our first study, bringing the total to 17. Figure 5.2 depicts the detailed approach selection procedure that we used. To better understand the selected basic studies, we have categorized them quantitatively based on publishing year, publisher type, article type, and the distribution among MDE and LCDP approaches. The result is shown in Figure 5.3a, 5.3b, 5.3c and 5.3d respectively.

5.4.2 Research questions

The performed assessment aimed at answering the following research questions:

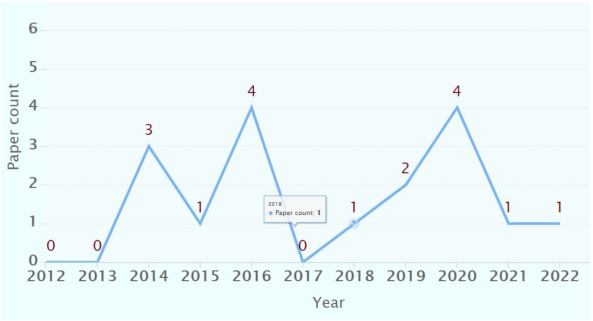
- ***RQ1****: To what extent do the considered IoT engineering platforms meet the characteristics of the proposed quality model?*
- ***RQ2****: What are the most and the least addressed quality sub-characteristics by the considered IoT engineering platforms?*

5.4.3 Assessment process

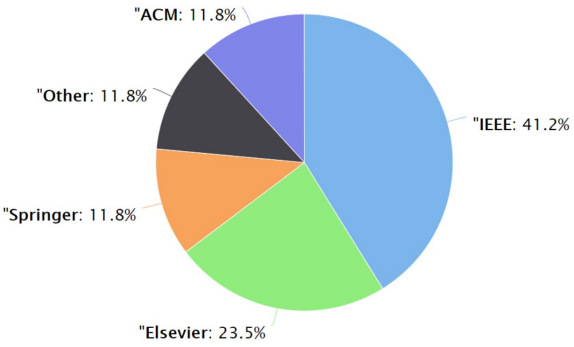
The quality assessment process of the considered IoT engineering platform has been done iteratively by going over all of the reference approaches. We established a set of questions for each sub-characteristic that must be addressed to confirm the platform's competence with respect to what is included in the model described in Section 5.3. By doing this, we made the evaluation process result easier and more reflective of the proposed model. Following that, we read the entire document and responded to the questions. Each question can be answered with "Yes" or "No". For instance, the following questions have been formulated to help in assessing the platform's *Functional Suitability*:

- Does the platform support the design and development of IoT systems?
- Does it support all layers (Edge, Fog, Cloud)?
- Does it mention any support for dealing with different communication protocols?

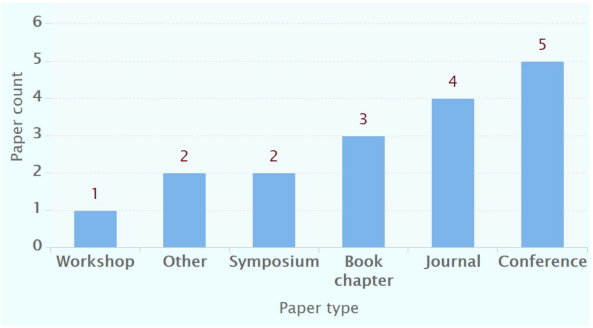
Consequently, given the approach under analysis, if the corresponding presented platform succeeds on at least 50% of the questions, it is marked as supporting the quality characteristic of interest. This procedure has been developed and implemented for all 31 sub-characteristics. The defined questions have been created purely to facilitate the review process and are entirely consistent with what is provided in the model. Unfortunately, due to space constraints, we could not present the extended table of questionnaires used to assess all of the quality characteristics. In this regard, a full set of evaluation questionnaires used in the evaluation has been published and can be accessed from an online database available at [220] as well as at the Appendix A.



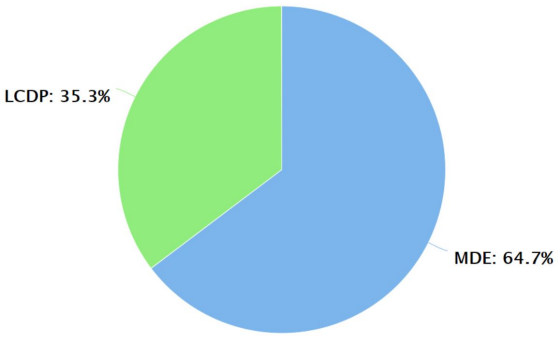
(a) Publications by year



(b) Publications by publisher



(c) Publications by approach type



(d) Publications by category

Figure 5.3: Overview of the selected basic studies

As mentioned in the previous section, some of the characteristics heavily depend on the dynamic properties of the underlying system infrastructures as well as during their usage. For instance, verifying sub-characteristics related to performance efficiency and usability might be difficult in general. Thus, during the performed analysis we relayed on information provided in the reference approaches when available. We considered such kinds of characteristics unsupported if the corresponding articles do not mention any mechanism addressing them.

5.5 Assessment results

This section presents the comprehensive findings of our assessment by answering the research questions. Table 5.1 summarizes the results of the study. The MDE and LCDP platforms supporting each quality characteristic of the considered product model are shown.

Quality characteristics support of IoT engineering platforms (RQ1)

According to the proposed model, eight characteristics with their associated sub-characteristics were evaluated on the approaches considered in this work. This section elaborates on the overall quality performance of such development platforms by comparing LCDPs and MDE platforms. As shown in Table 5.1, MDE approaches to support an average of 12 of the 31 possible sub-characteristics, whereas LCDPs support an average of 18. Furthermore, different sub-characteristics are not supported at all. Figure 5.4 shows the overall characteristic-level performance of the analyzed platforms. Figure 5.4a and Figure 5.4b depict the overall performance of MDE platforms and LCDPs against the eight main characteristics, respectively. To this end, an aggregated performance sum from all sub-characteristics was produced for each characteristic.

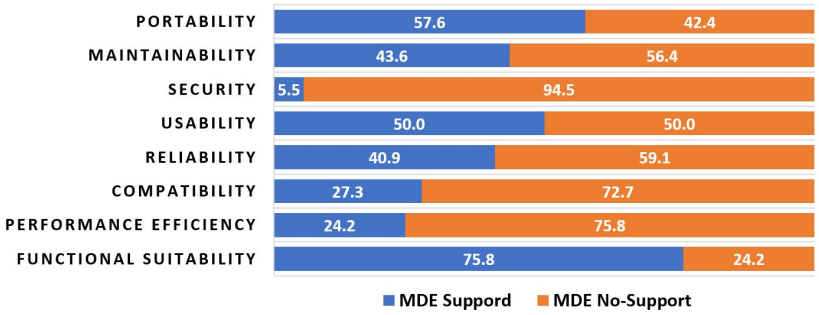
Both categories (MDE and LCDPs) perform well enough in quality characteristics related to *Functional suitability*, *Portability*, and *Usability*, with general supporting rates of 75.8%, 57.8%, and 50%

Table 5.1: Assessment overview

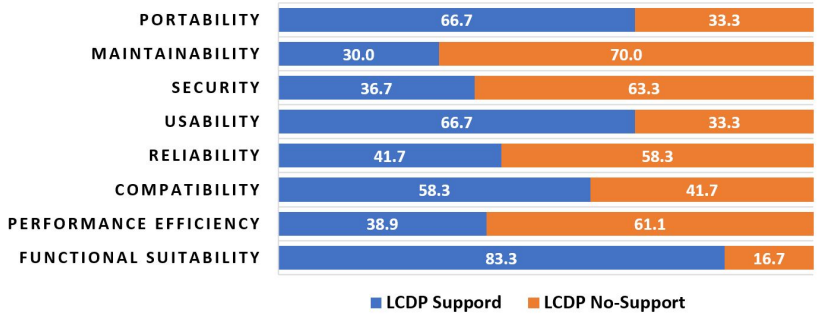
Characteristic	Sub-characteristic	MDE platforms	MDE(%)	LCDP platforms	LCDP(%)
Functional suitability			**75.8**		**83.3**
	Func. Completeness	[38] [61] [150] [157] [37] [161] [140] [40] [17]	81.82	[139] [89] [90] [138] [147]	83.3
	Func. Correctness	[36] [61] [38] [157] [37] [160] [140] [40] [17]	81.82	[89] [90] [138] [147] [144]	83.3
	Func. Appropriateness	[36] [150] [38] [37] [140] [40] [17]	63.6	[89] [90] [138] [139] [144]	83.3
Performance efficiency			**24.2**		**38.9**
	Time-behavior	Unsupported	0	[89] [90]	33.3
	Resource Utilization	[38]	9.09	[89] [90]	33.3
	Capacity	[61] [38] [37] [161] [140] [40] [17]	63.3	[89] [90] [139]	50
Compatibility			**27.3**		**58.3**
	Co-existence	Unsupported	0	[89] [90] [138]	50
	Interoperability	[38] [157] [37] [160] [140] [40]	54.5	[89] [90] [138] [147]	66.6
Reliability			**40.9**		**41.7**
	Maturity	[36] [150] [38] [157] [37] [160] [140] [40] [17]	81.82	[89] [90] [138] [139] [147]	83.3
	Availability	[37] [17]	18.18	[89] [90]	33.3
	Fault tolerance	[36] [61] [140] [40] [17]	45.45	[89]	16.6
	Recoverability	[38] [17]	18.18	[89] [90]	33.33
Usability			**50**		**66.7**
	Appropriateness	[61] [150] [38] [37] [17]	45.45	[89] [138] [139]	50
	Learnability	[140]	9.09	[89] [90] [144] [147] [40]	83.3
	Operability	[36] [38] [37] [160] [161] [140] [40] [17]	72.72	[89] [90] [138] [139] [144]	83.3
	User error protection	[37] [161] [140] [40] [17]	45.45	[89] [90] [138] [144]	66.6
	User interface	[36] [61] [150] [38] [157] [37] [160] [161] [140] [40] [17]	100	[89] [90] [138] [139] [144] [147]	100
	Accessibility	[37] [17]	18.18	[89] [90]	33.3
Security			**5.5**		**36.7**
	Confidentiality	[38]	9.09	[90] [138]	33.3
	Integrity	[38]	9.09	[90] [138] [147]	50
	Non-repudiation	Unsupported	0	[90]	16.6
	Accountability	Unsupported	0	[89] [90]	33.3
	Authenticity	[38]	9.09	[90] [138] [147]	50
Maintainability			**43.6**		**30**
	Modularity	[150] [38] [157] [37] [40] [17]	54.54	[89] [90] [139] [147]	66.6
	Reusability	[36] [40] [61] [38] [37] [140] [17]	63.3	[89]	16.6
	Analyzability	[61] [38] [140]	27.3	[89]	16.6
	Modifiability	[61] [150] [38] [37] [140] [40]	54.5	[89] [147]	33.3
	Testability	[40] [17]	18.18	[89]	16.6
Portability			**57.6**		**66.7**
	Adaptability	[38] [157] [140] [17]	36.36	[89] [90] [138]	50
	Installability	[36] [61] [150] [38] [157] [37] [160] [161] [140] [40] [17]	100	[89] [90] [138] [139] [144] [147]	100
	Replaceability	[38] [37] [40] [17]	36.36	[89] [90] [147]	50
	Overall	*135 out of 341 possible*	**39.6**	*95 out of 186 possible*	**51.1**
	Standard deviation		30.6		25.5

for MDE and 83.3%, 66.7%, and 66.7% for LCDPs, respectively. It is important to note that the listed supporting/non-supporting rates reflect the aggregated characteristic performance calculations from its corresponding sub-characteristics. Although this provides us with an overall picture of characteristic performance, it does not allow us to make a final judgment on whether certain sub-characteristics are better supported than others. For example, in terms of *Usability* aspects, all 11 MDE platforms satisfy the *User interface* quality characteristic, although only two of them satisfy the accessibility sub-characteristic.

MDE platforms have limited security and performance efficiency support, with overall support rates of 5.5% and 24.2%, respectively. For instance, in MDE, from the studies considered, only IoTML [38] promotes security by enhancing authentication, confidentiality, and integrity mechanisms while accessing the platform. On the other hand, LCDPs fall short concerning security and maintainability, with overall support rates of 36.7% and 30%, respectively. Each of the security aspects is implemented at least once by LCDPs, with the AtmosphereIoT [90] platform supporting all. Consequently, as shown in Table 5.1, MDE accounts for 135 points out of 341 possible support, representing approximately



(a) MDE platforms



(b) LCDPs

Figure 5.4: Quality characteristics support

39.6%, whereas LCDPs account for 95 points out of 186 possible, which would be about 51.1%.

Answer to RQ1: Overall, MDE quality is around 39.6%, while LCDPs account for 51.1%, resulting in the overall quality of the selected engineering platforms being approximately 45.5%.

Quality sub-characteristics support of IoT engineering platforms (RQ2)

In the previous section, we focused on overall quality characteristic support; in this section, we focus on individual sub-characteristic by highlighting the most and least addressed ones. Even though the average supporting rate of studied quality sub-characteristics for both MDE and LCDPs will be equal to the average supporting rate for the quality characteristics presented above (Sec. 5.5: MDE:39.6%, LCDP:51.1%), they differ in their deviation from the mean of individual supports. According to the standard deviation indicated in Table 5.1, MDE platforms have an average supporting standard deviation of 30.6, while LCDPs have a standard deviation of roughly 25.5, which is less than of MDE platforms. Such figures suggest that LCDPs are more likely to consistently touch all of the model's individual quality sub-characteristics than MDE approaches.

As shown in Table 5.1, IoT MDE and LCDPs cover all quality sub-characteristics such as *user interface* and *installability*. This is expected, given that the fundamental principle of MDE and LCDP technologies centers around enabling an easy-to-use and executable environment for developing applications with less effort, which cannot be accomplished without providing a user interface.

Figure 5.5 depicts an overall performance of sub-characteristics among IoT MDE and LCDPs engineering platforms. As indicated, none of the chosen MDE approaches address quality sub-characteristics such as non-repudiation, time-behavior, accountability, and co-existence. LCDPs, on the other hand, fall short on fault-tolerance, non-repudiation, reusability, analyzability, and testability, with a consistent minimal support rate of 16.7%, which implies a generic rate of at least 1 out of 6 LCDPs supports at least one of the sub-characteristics. On the other hand, Figure 5.6 depicts the average quality performance for both MDE and LCDP platforms. As can be seen, sub-characteristics

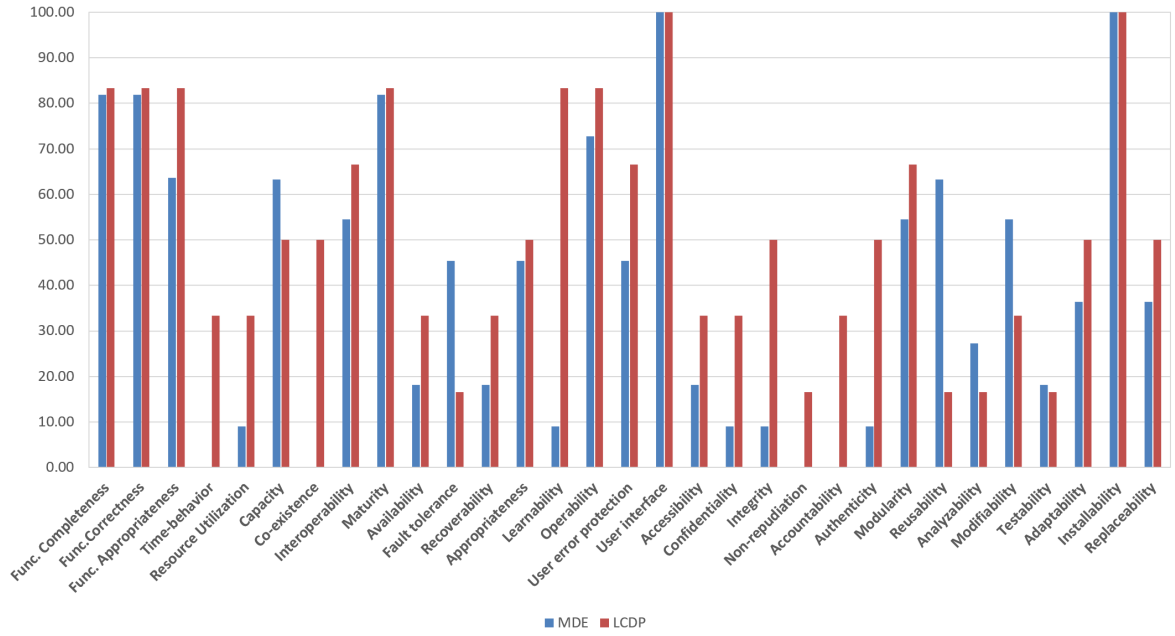


Figure 5.5: Quality sub-characteristics performances

such as user interface, installability, and functional completeness are the most well-supported by both categories, while testability, time behavior, accountability, and non-repudiation are the least supported.

> **Answer to RQ2:** The top three quality sub-characteristics addressed by both technologies are *user interface*, *installability*, and *maturity*. On the other hand, selected MDE approaches were unable to address quality sub-characteristics such as non-repudiation, accountability, co-existence, and time behavior. In contrast, LCDPs fall short of addressing fault tolerance, non-repudiation, and reusability with a consistent rate of 1 out of 6 LCDPs.

5.6 Discussion

In this section, we discuss the proposed model's suitability and its limitation with respect to general software quality assessment of software systems.

5.6.1 Model suitability

The proposed model aims to assist practitioners who have to design and develop IoT systems by exploiting low-code or MDE platforms. According to the results presented in Section 5.5, security-related characteristics are the least addressed. This is particularly pertinent given how IoT security concerns are dynamic and unstructured, leading to uncertainty among software developers in terms of concepts and terms [221]. MDE approaches are most affected since most conventional MDE platforms are deployed locally and used offline, making incorporating any form of security capabilities less required (e.g., authentication). We can argue that the dominant Eclipse Development Environment[1], which hosts a lot of classical MDE-based platforms, has a significant impact on this problem. On the other hand, although LCDPs neither excel in such security-related aspects, it is critical and rational to be integrated since such platforms are deployed in cloud-based environments, which are more likely to be attacked by unwanted intruders.

Furthermore, besides security, the results show that LCDPs lack quality criteria for general maintainability. For instance, according to Table 5.1, the reusability of LCDP is shown to be less supported

[1] https://www.eclipse.org/

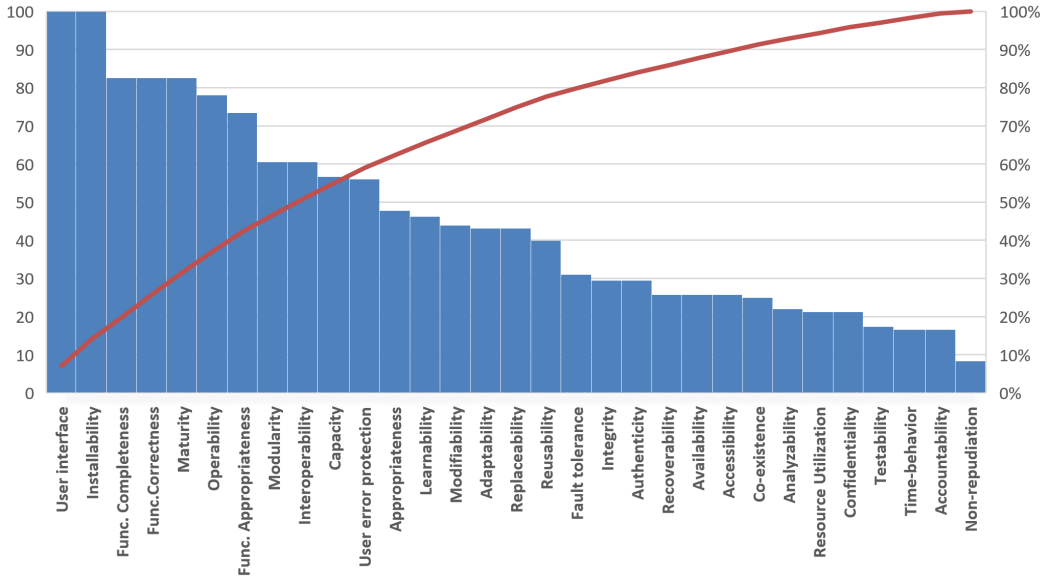


Figure 5.6: Average quality sub-characteristics performances

among others. This is generally true and can mainly be since more LCDPs are tailor-made, and most of their application developments and deployments are bound to a particular technology [58], making them difficult to modify and reuse elsewhere. In our study, only Node-RED [89] showcased means for supporting the usability of aspects of its components through the Node-RED modules that can be composed, built, deployed, and reused separately from one instance to another.

Due to the tight coupling between functions and their sub-functions found in different LCDPs and some MDE platforms, analysability becomes very hard to achieve. Our proposed model defines software analysability as the means to assess the efficacy and efficiency with which it is feasible to determine the impact of modifying parts of the platform by either removing it or injecting failures into it. According to the obtained results, only 27% of MDE platforms support such features, while for LCDPs, only one out of 6 support them. This quality characteristic is generally considered in the software design phase through different model-based system analyses. Still, in the actual implementation of the system, such quality is often ignored [222]. Concerning software *testability*, as indicated above, only 2 out of 11 of the analyzed platforms showed means for supporting it. The testing can either be done at the platform sub-functions level or at the system they develop. In general, developers tend to disregard this aspect. For instance, in the LCDP domain, low-code testing is still in its early stages, with no formal structure to the domain's ideas, concepts, and hypotheses which undoubtedly contributes to such lacking [215].

Another interesting fact is that the overall performance of IoT engineering platforms (MDEs and LCDPs combined), regardless of characteristics and sub-characteristics, is about 45.5%, in which MDE accounts for 39.6%, whereas LCDPs have 51.1%. Although we acknowledge that these measures cannot be regarded as a definitive measure of the IoT engineering platforms' quality questions, we believe they can point researchers in the right direction regarding the present state of the art in IoT engineering quality evaluation support. Consequently, we can finally draw the line of how the proposed model satisfies the quality evaluation procedure as "promising"; it unveiled several concepts that reflect what is already available in the IoT engineering domain.

5.6.2 Model limitations

In this chapter, we have extended the ISO/IEC 25010:2011 standard model to propose a software product model for assessing the quality of IoT engineering platforms. To evaluate the effectiveness of the proposed model, we employed it to determine the quality of 17 IoT platforms selected from our previous studies [8, 13]. In terms of limitations, we can state the followings:

- Although the presented model contributes positively to the software product quality assessment of IoT engineering platforms, it only performs very well when the evaluation is done at the platform's technical implementation level. However, the proposed model performed poorly regarding run-time software product quality assessment, such as quality linked to performance efficiency. This is primarily due to the implementation nature of the LCDP and MDE platforms. We believe that assessing such quality could be heavily influenced by the environment in which such software is deployed.
- The evaluation methods employed in this study and the reported results are critical because they are exclusively based on what was identified in the selected tool's approaches. However, in some instances, the published content of the approach may not correctly reflect the full capabilities of the platform under consideration. Furthermore, software platforms evolve, and new development is regularly contributed to the platforms. Therefore, we believe that integrating the results found in the approaches and the actual inputs from the tool vendors can significantly increase the legitimacy of the findings. We intend to address this in future research.

5.7 Conclusion and Future work

The rising market adoption of LCDPs has pushed businesses to incorporate LCDPs into their general-purpose solution stack. However, the effectiveness of such platforms' quality assessment can be controversial since it is linked to how satisfying the platform is to the party evaluating it. To address such concerns, it is strongly encouraged to rely on well-agreed and established standards to eliminate prejudice during decision-making. In this chapter, we presented an extension of the ISO/IEC 25010:2011 product quality model for assessing the quality aspects of IoT engineering platforms. We evaluated the software product quality of 17 IoT engineering platforms using the proposed model. The findings revealed that the overall performance of IoT engineering platforms is roughly 45.5%, with LCDPs doing slightly better than MDE platforms. Furthermore, security and maintainability aspects are found to be less addressed, whereas functional appropriateness, portability, and usability were found to be the most addressed. In the future, we plan to evaluate the quality in the use of the IoT engineering platform by extending the *quality in use model* of the ISO/IEC 25010:2011 standard in which we will be able to accommodate other quality aspects beyond the software product quality model.

Chapter 6

CHESSIoT: An approach for engineering multi-layered IoT systems

The challenges related to the complexity and heterogeneity of IoT systems are present in all of its aspects. On the one hand, the development processes need to take into account different design options, such as physical, functional, and behavioral architecture. MDE has demonstrated a significant benefit in automating software development by promoting the use of domain-specific languages (DSLs) tailored to a specific application domain. These models provide abstract system properties in which different sub-systems can be independently modeled, developed, and analyzed before being integrated to construct a fully functional system. As a contribution toward answering the third research problem (RP3), this chapter present the CHESSIoT, an approach for engineering multi-layered IoT systems. Initially the chapter evaluates potential contribution of our proposed approach in terms of modeling language coverage as well as engineering support. We do so by presenting two comparative analysis results that aim to clearly establish the potential of CHESSIoT in the above engineering support by taking into consideration the multi-layered engineering support. In addition to that, we also presents CHESSIoT domain specific languages covering the system, software and deployment aspect of a multi-layered IoT system. Finally a brief discussion is drawn around the above contributions.
The chapter is organized as follows: Section 6.1 presents an introduction to the topic. Section 6.2 briefly presents the high-level CHESSIoT engineering methodology. Section 6.3 presents the motivating comparative analysis between CHESSIoT engineering methodology with respect to 12 existing platforms. Section 6.4 presents in detail three DSL extensions used in the whole engineering process. Finally, Section 6.5 discusses the findings, whereas Section 6.6 concludes the chapter.

6.1 Introduction

Due to the inherent heterogeneity present in the IoT domain, engineering platforms such as MDE4IoT [36], ThingML [37], IoTML/BRAIN-IoT [38, 39], SimulateIoT [40] and Montithings [17] (just to name a few), have demonstrated the potential of MDE to be a realistic alternative for engineering scalable IoT systems. While that is the case, finding a platform capable of fully integrating different core engineering features becomes critical. From now on, we will constantly use the term **"Engineering"** to refer to a process that integrates the "development, analysis, and deployment" support from a unique environment when realizing an IoT system.

In this chapter, we introduce CHESSIoT, an approach and a tool for engineering multi-layered IoT systems. We briefly discuss the general methodology supported by CHESSIoT as well as a comparison with the existing engineering approaches in the industry. The CHESSIoT environment is built on top of the existing CHESS toolchain [16] with the aim of providing a fully decoupled extension for supporting the modeling, development, analysis, and deployment of the IoT systems. The CHESSIoT DSLs namely *SystemDSL*, *SoftwareDSL*, and *DeploymentDSL* are aligned with different modeling views as well as the engineering task that they correspond to. Through CHESSIoT, a user can benefit from a multi-view development environment in which each of the supported views has its own underlined constraints that enforce its specific privileges on model entities and properties that can be manipulated

For instance, System-level DSL was designed to satisfy the high-level physical representations and their relationships within a typical multi-layered IoT system. The DSL supports the specification of a typical IoT system covering from the low-level edge layer, Fog-layer as well as to the cloud. Note that, at this level, the model does not include any information related to the functional behavior of elements rather than their main physical construct. Furthermore, this model can later be annotated with failure behavior rules following CHESS-FLA constructs [44] which ultimately is used to conduct the early safety analysis tasks on the model.

In addition to that, the CHESSIoT Software DSL supports the system's functional and behavioral aspects of the system. The DSL extends the rich UML modeling language by means of defining new IoT-specific stereotypes and their interrelation targeting the low level. It is worth noting again that to enforce the model correctness as well as error avoidance during the design phase, palette elements can be hidden or shown based on the current state of the modeling process (eg: diagram type or view type). When the model is complete, a *CHESSIoT2ThingML* model transformation is launched to generate a series of fully functional ThingML [37] source models.

Finally, CHESSIoT offers means for modeling IoT system deployment plans as well as its runtime service provisioning. The IoT system components of a typical IoT system can be deployed at any layer namely edge, fog, and cloud. Designing the deployment plan of such a complex and heterogeneous system has to take into consideration several aspects and be aware of different satisfactory requirements [223]. The deployment model connects the software to the actual system nodes in which the software program will be executed. The model decomposes the inter-dependency between different nodes, machines, and services deployed to it. When the model is complete, a model-to-text transformation can be launched which generate a .yaml configuration file ready to be executed on a docker server.

In addition to the comparative analysis between CHESSIoT and other existing related approaches in terms of their abilities for supporting the engineering features specified above, we have presented the results from the evaluation between CHESSIoT DSL specification and modeling application entities across all layers in relation to the existing DSL. The results have shown huge gap in which CHESSIoT can potentially contribute. We believe that an effective tool should be capable of modeling all aspects of a typical IoT system, from the low-level edge layer to the fog and cloud layers, while maintaining consistency throughout the process.

6.2 The CHESSIoT engineering methodology

Through CHESSIoT, a user can benefit from a multi-view development environment in which each of the supported views has its own underlined constraints that enforce its specific privileges on model entities. In the end, the user can perform different engineering activities on the CHESSIoT models such as generating IoT device code, early safety analysis as well as deploying and managing the deployed services.

Figure 6.1 depicts the high-level illustration of the CHESSIoT methodology proposed in this dissertation. As from Figure 6.1, the 3 main primary DSLs namely SystemDSL, SoftwareDSL, and DeploymentDSL are the basic starting points for conducting additional engineering tasks supported by our tool. These DSLs are aligned with different modeling views as well as the engineering task that they correspond to. A more detailed explanation of the DSLs is presented in Section 6.4. Depending on the user's needs, as well as the stage of the development process, a specific view-compliant model corresponding to a specific metamodel is picked for usage.

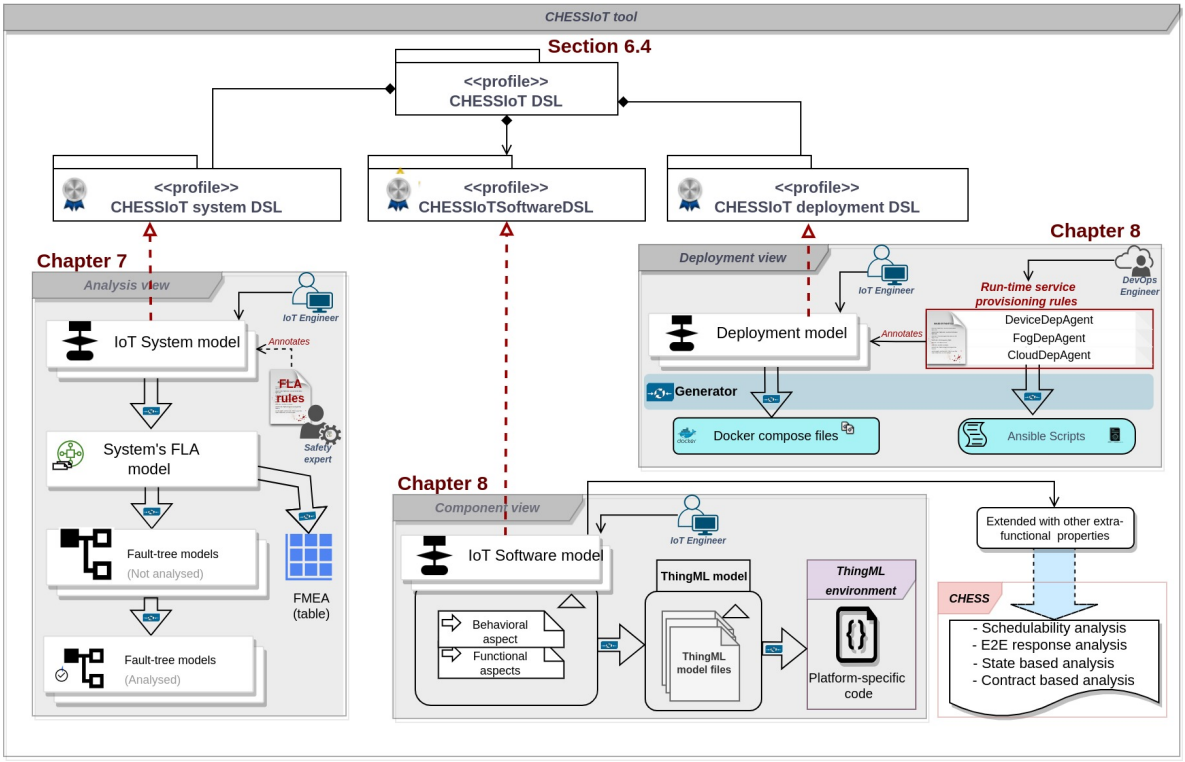


Figure 6.1: High-level approach

In CHESSIoT, different aspects of the system can be designed independently and then interlinked to satisfy specific engineering tasks to be performed on the model. For instance, under the *"System view"*, an IoT Engineer model the IoT system-level architecture that contains all of the system's major physical components, parts, sub-parts, and their interconnections. This model can then be handed to a safety expert who comes up with the system's failure logic behavior as well as basic component failure rates to be annotated to the model in order to perform the safety analysis. In the end, a series of model-to-model transformations is performed for achieving both qualitative and quantitative Fault Trees Analyses. More on the safety analysis approach is thoroughly discussed in Chapter 7

In addition to that, under the "component view", the user can define a functional model which contains the system's key software components, sub-functions, and interrelationships. Furthermore, a behavior model entitles each system's main sub-function to its own state machine in which aspects such as events, actions, and guards are associated with states and their transitions to realize the desired behavioral goal. When the model is complete, a *CHESSIoT2ThingML* model transformation is launched to generate a series of fully functional ThingML source models which is then used to gen-

erate platform-specific code ready to be deployed on low-level IoT devices. The same CHESSIoT software model extended with other extra-functional properties and benefited from the existing supported analysis. For instance, in our previous work, we demonstrated CHESS support for performing early real-time schedulability analysis on CHESSIoT models [42, 43]. More details on the CHESSIoT development approach is discussed in Chapter 8.

Finally, under the "Deployment view", the same user could be able to define the IoT system deployment plan as well as define rules on how to manage deployed services at runtime. The deployment model decomposes the inter-dependency between different node layers, machines, and one or more services deployed to it. In addition to that, a DevOps engineer can come through to define a runtime service provisioning model by automatically configuring software services based on a predefined model. When the model is complete, a model-to-text transformation is performed which generate a full .yaml configuration as well as Ansible [224] playbook scripts ready to be executed on a docker server. More details on the CHESSIoT deployment approach is discussed in Chapter 8.

6.3 Motivating comparative analysis

In the Chapter 3, we provided an overview of existing approaches for engineering IoT systems. These approaches covered categories such as modeling and development, safety analysis, and deployment support. In this section, we aim to conduct a comparative analysis between CHESSIoT and 12 approaches selected from these approaches to highlight their strengths and weaknesses, thereby emphasizing the need for a novel approach like CHESSIoT.

Generally, a comparative analysis is a systematic approach used to compare two or more things to identify their similarities and differences and evaluate their relative strengths and weaknesses. In the model-driven community, this approach was widely used. The results of comparative analysis can potentially provide valuable insights that can help developers, researchers, or other interested parties make better decisions about which technology to utilize for a particular task.

The upcoming section will compare existing model-based approaches for engineering IoT systems based on two primary relative contexts: the tool's supported modeling aspects and their engineering supports. Section 6.3.1 presents an overview of the analyzed approaches, whereas Section 6.3.2 presents the comparative assessment related to the tools' capability of supporting different modeling features in achieving a multi-layered architecture. To be more specific, this part looks at the tools support for modeling application entities across all layers, namely the low-level edge layer, fog, and cloud layers elements. Additionally, in Section 6.3.3, existing tools are discussed and compared with respect to their support for different IoT engineering tasks, including system development, safety analysis, deployment, and run-time service provisioning.

6.3.1 Selected platforms

Table 6.1 lists the 12 approaches, which have been selected according to the following criteria:

- Basic support for IoT system modeling: The approach focuses on modeling IoT systems and may provide advanced features for manipulating the model.
- Tool maturity: The approach has advanced beyond its initial or conceptual stages and is more mature.
- Age of the tool: The approach has been implemented within the last 10 years.
- IoT-specific: The approach is explicitly designed for engineering in the IoT domain.

The selection process was conducted iteratively, and we considered all three categories presented in Chapter 3 especially Sections 3.2, 3.3 and 3.4. From the selected approaches, only one approach was chosen from the analysis category. This is because most of the presented approaches do not

provide any means for designing IoT system models but rather focus on the manual development of FTs. Moreover, most of the approaches in the analysis category are considered to be outdated and conceptual, which does not meet our established criteria.

Table 6.1: Selected approaches for the comparative analysis

Tool name	Title	Year	Type
MontiThings [17]	MontiThings: Model-driven development and deployment of reliable IoT applications	2021	Journal
ThingML[37]	ThingML: A language and code generation framework for heterogeneous targets	2016	Conference
MDE4IoT[36]	MDE4IoT: Supporting the Internet of Things with model-driven engineering	2017	Conference
SysML4IoT[50]	Modeling IoT applications with SysML4IoT	2016	Conference
Monitor-IoT[26]	A domain-specific language for modeling IoT system architectures that support monitoring	2022	Journal
Simulate-IoT[40]	Simulate-IoT: Domain-specific language to design, code generation and execute IoT simulation environments	2021	Journal
DSL-4-IoT[133]	Design of a domain-specific language and IDE for Internet of Things applications	2015	Conference
UML4IoT[150]	UML4IoT-A UML-based approach to exploit IoT in Cyber-Physical manufacturing systems	2016	Journal
IoT-ML/ BRAINIoT[39]	BRAIN-IoT: Model-based framework for dependable sensing and actuation in intelligent decentralized IoT systems	2019	Conference
CAPS[72]	CAPS: Architecture description of situational aware Cyber-Physical systems	2017	Conference
Node-RED[89]	Node-RED: Low-code programming for event-driven applications	2016	Open tool
Silva I. et al. [180]	A dependability evaluation tool for the Internet of Things	2013	Journal

6.3.2 IoT modeling support

This section presents the comparative analysis of the considered approaches in terms of their abilities to model application entities across all layers of a typical IoT system (see Tab. 6.2). Our evaluation criteria aim at identifying tools that can effectively model all aspects of an IoT system, from the low-level edge layer to the fog and cloud layers, while maintaining consistency throughout the process. To achieve this, we broke down each layer into more detailed elements. For instance, at the edge layer, we considered modeling node elements such as sensor/actuators, their functionality and behavior, and hardware modeling. We also evaluated support for wireless communication modeling, which we believe is crucial for enabling access to data generated by physical devices and making the edge layer system operational in the digital world.

In the fog layer, we evaluated the tools' support for various fog components such as fog devices, gateways, and fog servers. Similarly, for the cloud layer, we assessed the modeling capabilities of the tools for cloud-based elements like cloud nodes, machines, and services. We also investigated if the tools support multi-view modeling and if they come with a graphical user interface. Additionally, we acknowledged that some tools may have unique modeling capabilities that may not be captured in the checklist, so we added a column to highlight any additional features.

The comparative findings from the assessment presented in Table 6.2 are highlighted below relative to CHESSIoT-supported modeling features that will be presented with details in the next section:

1. From Table 6.2, it can be seen that all of the selected platforms provide a modeling environment, with most of them offering a graphical modeling option, except for ThingML[37], which only offers a textual modeling option. While textual-based approaches may be more scalable, graphical ones are usually more accessible and user-friendly. Textual interfaces can become overwhelming, particularly when the system becomes more complex, and can often come with their learning curve in terms of understanding new textual languages. Similar to MontiThing [17], CHESSIoT

Table 6.2: Comparative table on supporting different IoT modeling features

Tool	Graphical user interface	Multiview modeling	Edge layer					Fog layer				Cloud layer			Other supported modeling capabilities
			Device node	Functional design	Behavior design	Hardware design	Wireless support	Fog node	Fog device	Fog gateway	Fog server	Cloud node	Cloud machine	Services	
MontiThings[17]	Yes	No	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No	Error handling design capabilities, deployment planning, featuring deployment design suggestions
ThingML[37]	No	No	Yes	Yes	Yes	No	Yes	No	No	No	No	No	No	No	Textual Component and Connector architectures, asynchronous messaging
MDE4IoT[36]	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No	No	Software to hardware allocations, consistency assurance, and run-time self-adaptation design
SysML4IoT[50, 148, 225]	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No	No	System quality of service (QoS), Publish/subscribe paradigm, system's self-adaptive designs
Monitor-IoT[26]	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Synchronous and asynchronous dataflows design across the edge, fog, and cloud layers to support the monitoring.
Simulate-IoT[40]	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Database designs, wireless sensors and actuator network (WSAN) design support
DSL-4-IoT[133]	Yes	No	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No	A visual domains specific modeling language for modeling IoT wireless sensor network
UML4IoT[150]	Yes	No	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No	Source code level annotations in case the UML design specification is not available
IoT-ML/ BRAINIoT[38, 39]	Yes	No	Yes	Yes	No	No	No	No	No	No	No	Yes	Yes	Yes	Run-time system deployment and dynamic remote edge/cloud reconfiguration designs
CAPS[72, 154]	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No	No	Physical space view modeling to describe the area involved in situation awareness
Node-RED[89]	Yes	No	Yes	Yes	No	No	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Cloud-based data flow modeling. Wiring together pieces of code blocks to carry out tasks.
Silva I. et al. [180]	Yes	No	No	No	No	No	No	Yes	Yes	Yes	Yes	No	No	No	Modeling of IoT network layer as a graph of devices (vertices) and edges (links).
CHESSIoT	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Support for the design of system failure logic behavior as well as run-time service provisioning

has adopted an approach that integrates both textual and graphical modeling approaches, limiting the use of the textual interface to simple definition tasks such as failure logic behavior rule annotation and run-time service provisioning, while major and complex system modeling is supported graphically.

2. After analyzing the selected approaches, it is evident that a considerable number of them do not support the multi-view modeling approach. This modeling approach is crucial in improving the accuracy of system design and enforcing the separation of concerns, where the model is simplified and designed from various perspectives. Multi-view modeling generally complements component-based design [107], which is also supported by CHESSIoT. These two methodologies have tremendous potential in dealing with the complexities of IoT systems. Among the 12 considered tools, only MDE4IoT [36], SysML4IoT [50], and CAPS [72] platforms provide support for these methodologies.

   We acknowledge that there are other platforms that implement alternative approaches that might complement multi-view modeling depending on the modeling context supported by the tool. For example, Node-RED [89] supports a multi-flow modeling approach, allowing different parts of the system to be developed separately from various flows while still sharing a common development context. However, Node-RED practically supports only a single data view that is shared among different flows, and other views are not supported. Therefore, it may not be suitable for more complex IoT systems that require multiple perspectives and separation of concerns.

3. As it can be seen from Table 6.2, the majority of approaches support modeling at the edge layer elements, namely components such as sensors/actuators, computing boards, and so on [17, 26, 36, 89, 225]. Except for IoT-ML [39], which exclusively focuses on the functional aspects targeting

cloud-based resource allocation, other platforms offer even more advanced design mechanisms, such as run-time self-adaptation [36, 225] as well as runtime error handling capabilities [17, 26] at the edge. While technically allowing predefined functional node behaviors to be defined, Node-RED [89] can model the data processing logic of the device-layer elements; however, it does not explicitly support any form of out-of-loop logical behavior specification. CHESSIoT, on the other hand, combines both functional and behavioral modeling of the element at the edge, and through different modeling views, a model portion can then be used for different engineering purposes.

In addition to that, CHESSIoT can explicitly model wirelessly communicated ports that support the MQTT protocol [226]. We have stressed this necessity as an essential factor in making the device layer components alive and suitable to be integrated into the digital world. Eight out of twelve platforms support this feature, which is very promising evidence of how mature MDE approaches are ready to address the scalability and interoperability issues faced in the IoT field.

4. Analysis of the table reveals that only a few of the considered platforms, namely [26, 40, 89, 180], support the modeling of fog layer elements. This lack of support for fog layer modeling is a significant limitation and is frequently cited as one of the drawbacks of MDE approaches in IoT. In our experience, IoT language developers tend to prioritize device-level modeling and development over the fog layer, even though implementing a robust code generator capable of generating fully integrated fog-layer code is a complex task due to the required designs and heterogeneity. Nevertheless, we believe that considering the fog layer is crucial for realizing a fully functional IoT system. While CHESSIoT does not explicitly focus on fog-layer code generation, our approach does provide deployment modeling and artifact generation targeting fog-layer deployment.

5. In the realm of cloud-layer modeling support, it is apparent from the table that only IoT-ML [39] and SimulateIoT [40] offer complete support for cloud-based design. IoT-ML is specifically designed to enable run-time system deployment modeling and dynamic remote edge/cloud reconfiguration, while SimulateIoT provides an IoT simulation and execution environment. This highlights the same issue discussed earlier, namely that most approaches concentrate on low-level development at the expense of other layers. For instance, Node-RED [89] and MonitorIoT [26] focus on service-oriented modeling while ignoring the context in which such services are deployed. In contrast, CHESSIoT enables the modeling of inter-dependencies between different nodes, machines, and services, and facilitates the provision of services deployed on such nodes across all layers.

6.3.3 IoT engineering capabilities

In this section, we evaluate the selected IoT approaches based on their ability to support various engineering tasks in developing, analysing, and deploying IoT systems. Table 6.3 shows our investigation of whether a platform follows a well-structured development approach that integrates all its supported engineering tasks and how these tasks are emphasized and followed during the entire process. Specifically, we looked at the tool's capability to generate platform-specific code for development and assessed its support for safety analysis and other types of analysis. Regarding deployment, we focused on the platform's ability to generate deployment-related artifacts and support run-time service provisioning mechanisms. Additionally, we highlighted each approach's specific focus and the typical validation methodology used. We acknowledge that safety-related analysis is not the only important aspect for IoT systems, and we also considered other types of analysis that are supported by each platform.

The results of the assessment are summarized in Table 6.3. Based on these results, we can highlight several interesting findings related to the engineering support provided by CHESSIoT, which is presented in the next section.

1. Table 6.3 shows that three of the considered approaches, namely [133, 150, 180], do not provide any details on the supported engineering methodology throughout their development process. We strongly believe that engineering platforms should have a well-defined methodology that guides

Table 6.3: Comparative table on supporting different IoT engineering capabilities

Tool	*Following a well-defined engineering methodology*	**Development**	**Analysis**		**Deployment**		**Empirical assessment**	
		Generate code	*Safety analysis*	*Others*	*Generate config.*	*Service provision*	*Approach*	*Tool's specific focus*
MontiThings[17]	Yes	Yes	No	No	Yes	Yes	*Proof of concept and a case study*	Engineering reliable IoT systems by separating concerns, handling errors, and enabling deployment to heterogeneous edge devices.
ThingML[37]	Yes	Yes	No	No	No	No	*Proof of concept and a case study*	A modeling tool and a highly customizable multi-platform code generator targeting only the edge layer
MDE4IoT[36]	Yes	Yes	No	No	No	No	*Case study*	Support design, development, and run-time management of IoT systems
SysML4IoT[50, 148, 225]	Yes	Yes	No	System QoS	No	No	*Proof of concept and a case study*	System functional design, publish/subscribe and self-adaptations at the edge.
Monitor-IoT[26]	Yes	No	No	No	No	No	*Proof of concept, case study, and experimental results*	Multi-layer visual modeling language for monitoring architectures of IoT systems based on the ISO/IEC30141:2018 reference architecture[1].
Simulate-IoT[40]	Yes	Yes	No	No	Yes	Yes	*Proof of concept and a case study*	Define an IoT simulation environment and execute it. Support for databases, complex-event processing engines, or message brokers
DSL-4-IoT[133]	No	No	No	No	Yes	No	*Case study*	A high-level visual programming language tailored to develop IoT applications compatible with the OpenHAB framework.
UML4IoT[150]	No	Yes	No	No	No	No	*Proof of concept and experiment*	IoT environment to support in the integration of CPS components into modern IoT manufacturing environment.
IoT-ML/ BRAINIoT[38, 39]	Yes	Yes	No	Risk analysis	Yes	Yes	*Proof of concept*	Edge/cloud deployment marketplace. Orchestration of distributed IoT systems leveraging dynamic and heterogeneous designs
CAPS[72, 154]	Yes	Yes	No	No	No	No	*Proof of concept and a case study*	A multi-view modeling approach that uses ThingML for code generation at the edge/device layer.
Node-RED[89]	Yes	No	No	No	No	Yes	*Well established tool*	Multi-flow based development approach. Acts as an interpreter for the data flow-related aspect of the system.
Silva I. et al. [180]	No	No	Yes	No	No	No	*Proof of concept and experiment*	A dependability evaluation tool for IoT that considers hardware faults and permanent link faults performing safety analyses and generating system FTs
CHESSIoT	Yes	Yes	Yes	Existing real-time	Yes	Yes	*Proof of concept and a case study*	Multi-layered system and software design, code generation, safety analysis, and deployment for IoT systems within a unified platform.

the development process of IoT systems. Such a methodology can potentially reduce complexity and provide users with fewer complications while using the platform. On the other hand, as shown in Figure 6.1, CHESSIoT offers a well-defined component-based design approach that supports different engineering tasks, such as code generation, safety analysis, and deployment, at different stages of the design and through different viewpoints. This approach enhances the tool's suitability

and significantly contributes to the model's correctness throughout all engineering stages.

2. In terms of code generation, three platforms - MonitorIoT [26], DSL-4-IoT[133], and Node-RED[89] - do not support any form of code generation that can be deployed on IoT devices. However, one of the main goals of Model-Driven Engineering (MDE) is to enhance automation in the development process [35]. We believe that generating partial or full system code is one of the most critical factors in speeding up the development process. While we recognize the complexities involved in generating fully functional code that can be deployed at any layer without manual intervention, continuous improvement of code generators that attempt to cover the different heterogeneous aspects of an IoT system could help bridge this gap.

   ThingML is a platform that provides a code generator capable of producing fully functional code in various programming languages, such as Java, C, C++, JavaScript, Python, and Go, with a particular focus on the edge layer. While the number of supported languages may vary depending on the ThingML version and code generator, additional languages can be integrated by implementing new code generators or expanding existing ones. To save time and resources, CHESSIoT uses ThingML's code generation system. This is done by transforming CHESSIoT's software models into ThingML's models. More information will be provided in Chapter 8.

3. Developers of IoT systems often assume that their devices or systems will always succeed, but this is not the case [29]. Failures can occur for various reasons, including device age, communication protocols, data sources, deployment environment, and human error. In our assessment of 12 platforms, we found that only Silva I. et al.'s approach [180] supports safety analysis through Fault-Tree. However, even this approach only analyzes the fog layer network, which is only a small part of the overall functionality of an IoT system. It is important to note that safety requirements for IoT systems are still emerging and do not always keep up with changing technologies [46]. Many safety analysis approaches presented in Section 3.4 are conceptual and do not support automated fault-tree analysis.

   In addition to safety analysis, we discovered that only two out of 12 platforms we examined offer different types of analysis. SysML4IoT [148] supports reliability analysis of IoT systems through verification of the system's QoS properties, while the BRAINIoT platform [39] uses IoT-ML [38] to support security and privacy risk analysis through a decentralized process that ranks vulnerabilities into four levels: negligible, limited, significant, and maximum. This is a significant engineering challenge in the IoT field.

   To enhance the capabilities of CHESSIoT models, additional analyses can be performed using the underlying CHESS infrastructure [16] on which the platform is built. For example, in previous research [42, 43], we showed how timing characteristics could be added to CHESSIoT functional models to conduct real-time schedulability analyses.

4. When it comes to deployment support, MontiThings and SimulateIoT have different approaches. MontiThings uses a deployment manager to capture device states at runtime, generate a valid docker-compose.yml, and send it to devices for execution. SimulateIoT, on the other hand, utilizes Docker swarm to manage deployed docker containers across all node layers. Only four of the twelve platforms examined offer runtime deployment artifact generation or service monitoring. However, CHESSIoT provides an environment that allows for modeling and generation of docker-compose files reflecting services that need to be deployed on machines running at a given node.

6.4 The CHESSIoT domain specific language

The CHESSIoT modeling environment has been built on top of the Eclipse Papyrus[2] in terms of extensions of UML/SysML. The three profiles that make up the CHESSIoT DSL are explained in detail below.

6.4.1 System-level DSL

The System DSL has been designed to satisfy the high-level physical representations and their relationships within a typical IoT system. The DSL supports the multi-layered specification of a typical IoT system ranging from the low-level edge layer, Fog-layer as well as the cloud. The language extends the rich SysML modeling language in terms of new IoT-specific stereotypes and their interrelations. Note that, at this level, the model does not include any information related to the functional behavior of elements rather than their main physical construct.

The modeling concepts underpinning the system DSL are shown in the metamodel depicted in Fig. 6.2. The ***System*** metaclass represents an IoT system as a collection of physical devices and other entities connected to collect, process, send, receive, and store data. These device entities can range from tiny sensors to much larger items like cars and planes. As the top-level representation element, the system can encapsulate other subsystems, allowing the IoT system-of-systems architectures to be supported.

The ***IoTElement*** represents things that can be physically represented in the IoT ecosystem. This can be of any type depending on the layer from which such an element is regarded. This can range from a tiny micro-controller at the thing layer, a gateway at the fog layer, and a cloud server when looked at from the cloud side. In the physical world, the *IoTElement* can also represent an object as bigger as a car, a plane, or a house. The system can have one or more IoTElements; each with one or more communicating ports. The modeling constructs can be conceptually grouped with respect to the main layers they define, i.e., edge, fog, and cloud layers as described below.

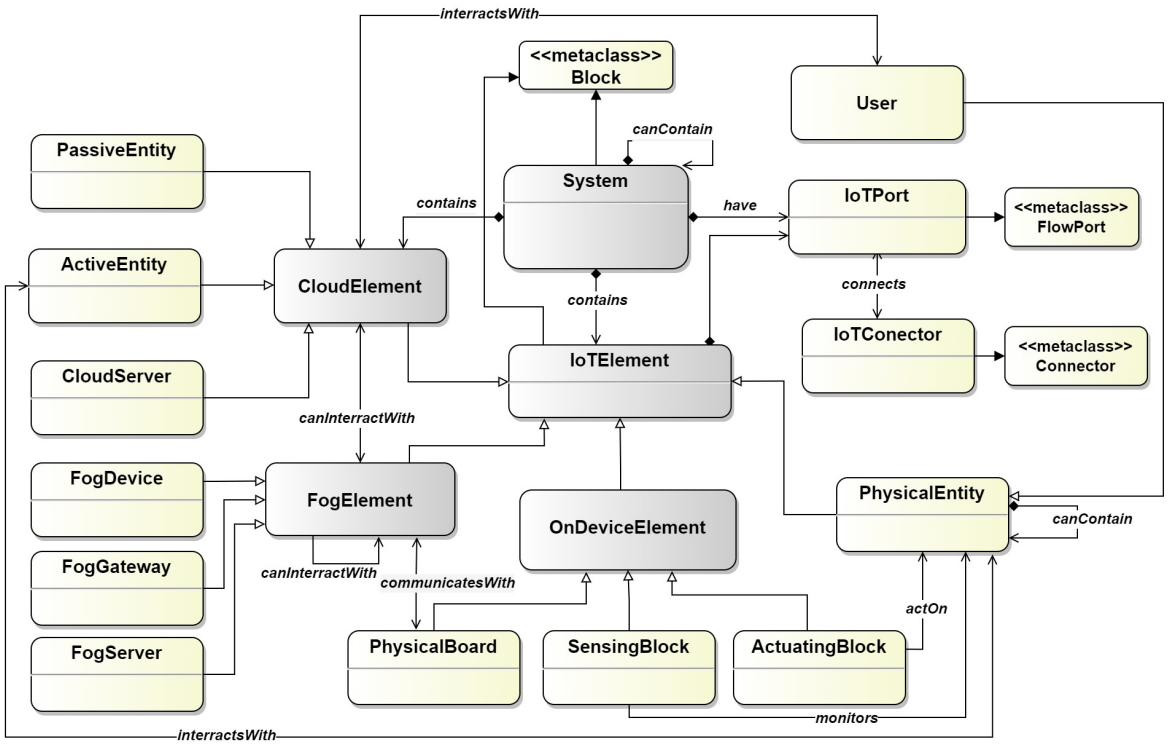


Figure 6.2: CHESSIoT System-level metamodel

[2] https://www.eclipse.org/papyrus/

Edge Layer: *OnDeviceElement* represents any form of low-level IoT device that may contribute to the system's functional behavior at the edge layer. A *SensorBlock* is primarily responsible for detecting changes in its surroundings and reacts accordingly by generating signals that can be interpreted by either a human or a machine. A sensor, lacks a physical input port and, in the event of a failure, can react differently based on the nature and severity of the internal failures. *ActuatorBlock* is a device responsible for reacting to received electric signals and acting upon them by changing the shape, position, or state of the component or part of the system to which it is attached. An electric servo motor, for instance, responds to a signal by turning on, off, changing direction, or speed. In the case of a door-locking system, it can either close or open the door.

PhysicalBoard represents a hardware controller on which the software runs. This can include a number of IoT-related boards that are expected to execute the actual code, thus interfacing the sensor and actuator. A RaspberryPi or Arduino board, for example, processes data from various sensors and sends appropriate signals to actuators and other connected devices as the Fog layer. *PhysicalEntity* can be almost any physical object or environment on which a *OnDeviceElement* can act up. A self-driving car software, for instance, runs on various boards attached to the car but not on the car itself. So a car is a physical entity, while those controlling elements can be classified as any type of on-device element.

It should be noted that a physical entity may host other physical entities and interact with other physical entities. In general, we consider physical entities to be passive elements, and in the event of a system failure, they cannot be considered the root cause unless they are categorized as "User". In particular, the concept of *User* refers to a human actor that uses the system or, in certain contexts, is part of the system itself. A user is a special type of *PhysicalEntity* that interacts with other parts of the system at all levers. For example, a user might interact with an IoT application deployed on a remote server while actively participating in such a system's decision-making process. It's worth noting that a user doesn't necessarily need to be a human; it can also be an autonomous entity that's intelligent enough to interact with the system.

Fog Layer: *FogElement* is any device that serves as a computational link between the physical and virtual worlds, in this case, cloud infrastructures. If necessary, these components can do preliminary computations and convey the results to the on-device elements. This implies that they may have varied storage and processing capacities depending on the use case and completely different hardware and software features. Any IoT device installed at the fog layer for data processing and storage is represented by a *FogDevice*. The *FogGateway*, on the other hand, transfers information between fog devices and fog servers, as well as cloud servers connected to it. Finally, *FogServer* computes this data to determine the next operation. This layer is critical because it regulates processing speed and information flow. Fog node configuration involves understanding different hardware compatibility, the devices they influence, and networking capabilities.

Cloud Layer: A *CloudElement* is a type of IoT device that operates at the cloud level and contributes to the overall functionality of the system. It builds upon standard IoT elements and can be shown as follows. Similar to a *FogServer*, a *CloudServer* hosts various cloud-based services and applications. A consumer entity refers to any third-party element that can communicate with the server to access its data, and can be classified as active or passive. An example of an active consumer entity is a computer running software to monitor and control sensors remotely. On the other hand, a passive consumer entity is a traffic light actuator that receives commands from the server to function.

6.4.2 Software DSL

The CHESSIoT's software DSL has modeling constructs that allow for the specification of IoT system behavior. These constructs are displayed in the metamodel shown in Figure 6.3. It is important to note that the software meta-model mainly supports low-level devices at the edge layer, but in some cases, these devices could also be deployed at the fog layer if they fall into that layer. The DSL extends the UML modeling language by defining new IoT-specific stereotypes and their interrelation.

The CHESSIoT Software metamodel can be divided into two main sections, for specifying *functional* and *behavior* aspects of the constituting components as described below.

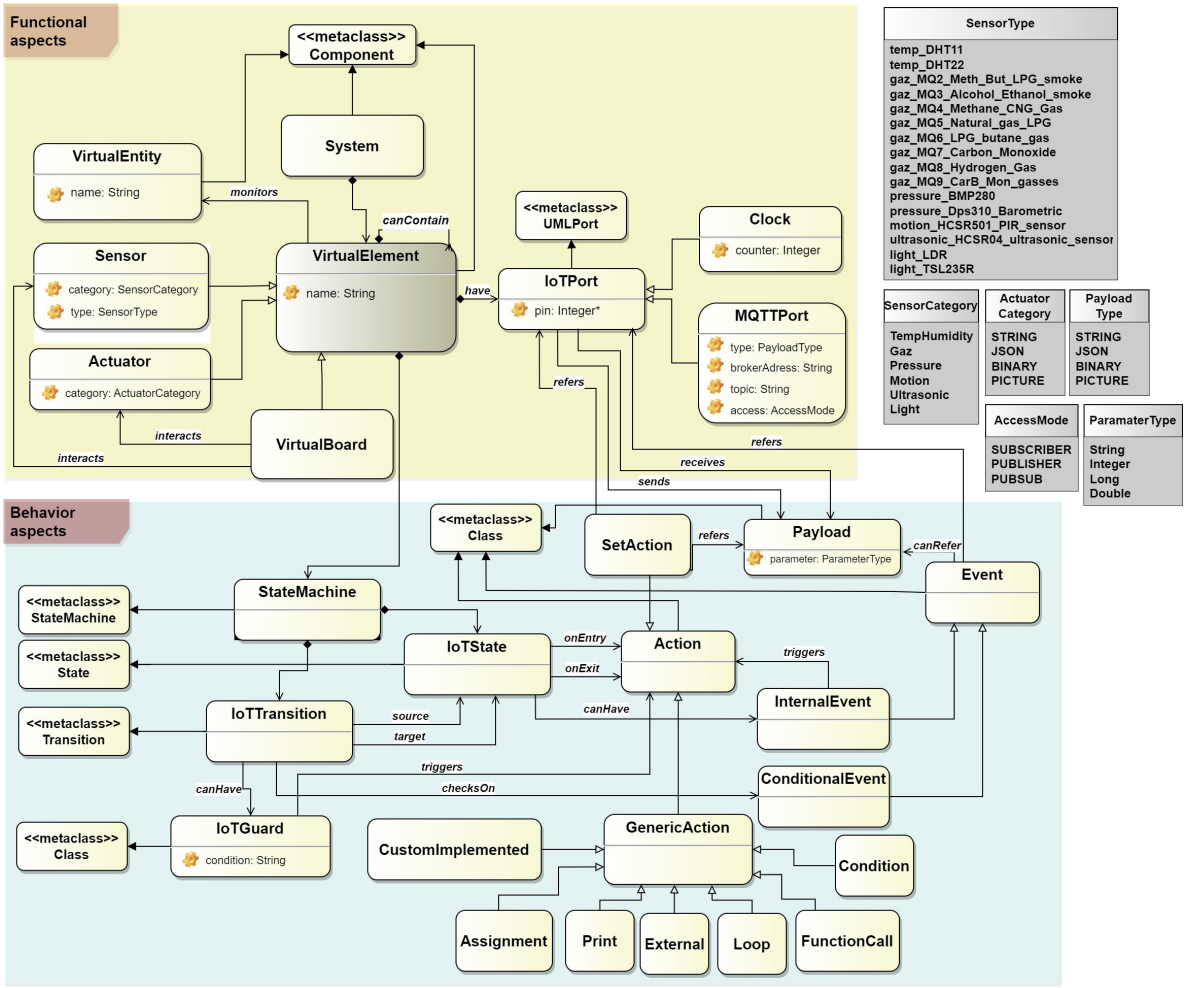


Figure 6.3: CHESSIoT Software Metamodel

Functional aspects: *VirtualElement* represents an *IoTelement* in the virtual world. As mentioned in Section 6.4.1, this could be classified as any IoT device, an element that could be of interest at the edge. As shown in Fig. 6.3, the *System* can consist of one or many virtual elements, and depending on the use case, a virtual element could contain one or more virtual elements.

VirtualEntity is a virtual representation of the *PhysicalEntity* from the system model in the digital world. This element can represent any object or place where IoT devices or equipment are installed. In a room monitoring system, for example, a room is usually represented as a virtual entity in which other sensors and actuators are installed.

One of the most fundamental components of the IoT ecosystem is the *Sensor*, which is responsible for transforming relevant information from its surroundings into an electric signal that the computing board can process. On the other hand, the *Actuator* converts electric signals from the board into physical events or states, depending on its type. The language supports different sensor categories and types. A combination of sensor category and type servers is a crucial determining factor during transformation, and it is also the same case for the actuator. We understand that there are many more types of sensors and actuators than our approach supports, but for the sake of simplicity, we focused on only a few, as illustrated by the proposed meta-model.

IoTPort allows message exchange between two different components by exposing or requiring the data from components. An IoTPort can have one or more integer pins used to generate pin-related code on the virtual board. Two special types of ports, *MQTTPort* and *ClockPort*, are employed in specific

cases. For instance, MQTTPort specifies the MQTT-related interface that wirelessly communicates with a remote broker. This port contains information about the payload type, broker URL, device topic, and access mode (i.e., publisher, subscriber, or both). When necessary, the clock port is utilized to define logical delay checks. It should be noted that these two special port types are not required to be physically connected to others.

VirtualBoard is a virtual representation of the computing board device where the code runs. This device interacts with sensors and actuators and uses the *IoTPorts* to communicate with the external components.

Behavioral aspects: Every type of VirtualElement has a state machine with a behavioral specification. The following syntax is used to define the behavior of the component. *Payload* is a simple and stand-alone message object that transports data between components. These elements can have zero or more parameters that define the type of data that must be sent.

*Event*s are triggered in various ways based on the component's current processing phase. An event can generally be triggered based on the payload condition detected at the ports. An *Event* can be a *ConditionalEvent*, which occurs during the transition process from one state to the other, or an *InternalEvent*, which occurs internally within the state of a certain component.

Depending on the sort of action to be taken, *IoTAction(s)* can be of many forms. These actions can be customarily defined, or they can reuse the information about the payload and the port where such action must be carried out. For example, when entering or quitting a state, an action can be classed as *OnEntry* or *OnExit* actions. There are two main types of actions:

- *SetAction:* This is used during external communication between two components through a predefined port.
- *GenericAction:* It is a specific type of action that can be implemented during the design phase for particular measures such as assignment, print, loop, checking a value status, function call, and so on. These actions require different arguments and can be customarily implemented with platform-specific code.

IoTState defines the situation of the component from its initial engagement to its final disposal in the ecosystem. *IoTState* extends existing UML states by collecting all behavior information relating to events and activities that must be performed at a specific time. From a transition standpoint, an IoTState can be classified as a source or target, along with an initial, intermediate, or final state.

IoTTransition makes it possible to transition from a source state to a target state while preserving the trigger from the invoking condition as well as the guard value. *IoTGuard* expressions are boolean expressions defined by state values. They enable a state transition by determining whether the *OnExit* action was correctly completed.

6.4.3 Deployment metamodel

The CHESSIoT's deployment DSL has been defined to aid in the design of the deployment strategy. The primary purpose is to provide an intuitive way for the user to define the deployment architecture as well as runtime service provisioning procedures that can be applied to configure such generated services remotely. The DSL addresses service-oriented deployment topologies, mostly at the fog and cloud layers. The main concepts of the deployment metamodel are shown in Fig. 6.4.

A *Node* is a central component that connects all other deployment elements. It represents a computing cluster at the center that combines one or more data processing units. These nodes can be found at any layer, including edge, fog, or cloud, and are labeled as *DeviceNode*, *FogNode*, and *CloudNode*, respectively.

The *Machine* construct is for specifying a dedicated middleware server that can host one or more services running on it. Machines could be anything from small computer boards at the edge and fog layers to huge cloud-computing servers. In our context, the machine is always declared inside a node

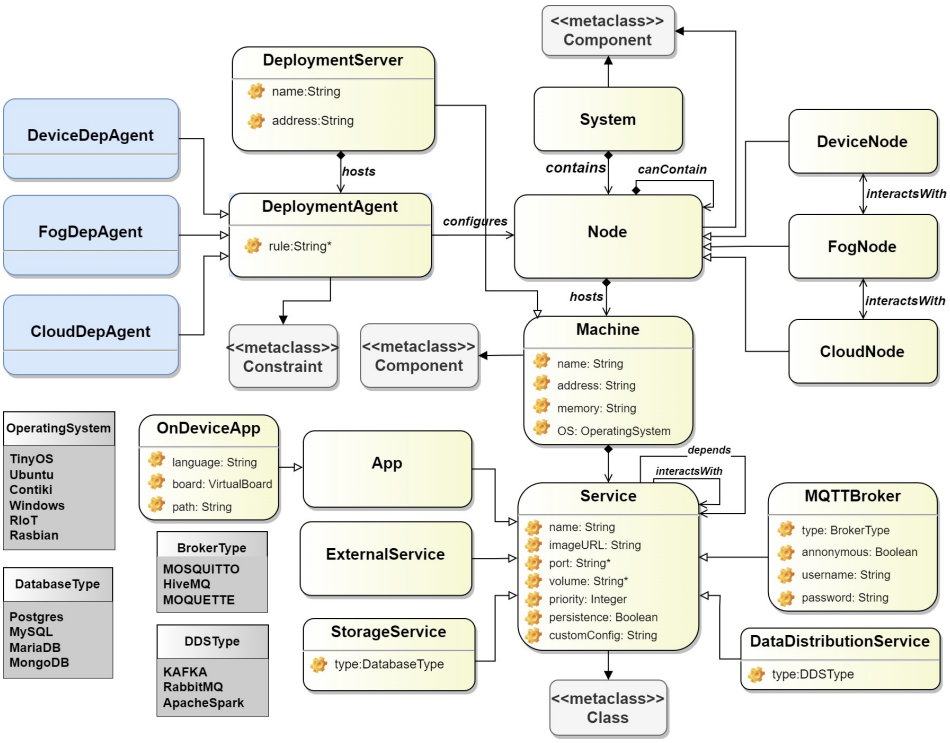


Figure 6.4: CHESSIoT Deployment Metamodel

and should eventually have an IP address that the service operating on could be identified from. Other properties, such as memory capacity and operating system, can be also specified by the user.

In IoT, a *Service* is a self-contained entity that can consume acquired data and apply computational logic to achieve a goal. It can be deployed at practically any layer of the system, depending on the type of need and the computation capabilities of the node in which the service is to be run. Services can connect via Web protocols (e.g., HTTPS, MQTT) and may also depend on one another.

As with CHESSIoT, the end goal of deployment modeling is to generate docker configuration files (.yaml files), therefore, a service must be established with basic parameters like imageURL, ports, persistence, and so on. If the service needs to persist data on the platform, a *volume* attribute must be specified, as well as the boolean *persistence* value set to true. The *priority* attribute specifies the order in which individual services are prioritized in case of a machine memory shortage.

We are mainly concerned with IoT services that are generally involved in a typical IoT ecosystem. An *MQTTBroker*, for example, is used to define a remote MQTT server service, and attributes such as broker type (Mosquitto, HiveMQ, Moquette) are supported. The broker, which is also a service, captures its specific properties such as type, anonymous access, persistence, username, and password. The current implementation enables a user-friendly environment, and in case no data is provided for a given property, default values are used instead. Other services, such as *DataDistributionService* like KAFKA, RabitMQ, and ApacheSpart, are due to be supported.

Furthermore, the environment enables customary configured services, and when such a property is employed, the definition is added to the generated file unchanged. Furthermore, any IoT-specific *ExternalService*, such as Node-Red[3], as well as *StorageServices* such as database containers, could be specified. Finally, *OnDeviceApp* can be defined, allowing it to be distributed on edge devices.

A *DeploymentAgent* is a collection of predefined expressions determined at the node level to demonstrate the run-time service provisioning behavior on the machines deployed at the nodes. *DeviceDepAgent*, *FogDepAgent*, and *CloudDepAgent* are defined to perform this task at the edge, fog, and cloud layers, respectively. Details on the developed textual deployment language and the corresponding code generator are given in Chapter 8.

[3] https://nodered.org/

6.5 Discussion

In the preceding sections, we presented the results of a comparative study that examined the capabilities of CHESSIoT in relation to 12 other related works. Our focus was on assessing the platform's ability to provide a multi-layered modeling environment and offer engineering support in terms of development, analysis, and deployments. In this section, we will discuss the significant findings by highlighting key outcomes from the two contexts considered.

Regarding the support for IoT modeling, when evaluating a cumulative sum of all 12 platforms, we found that out of the 154 feasible possibilities (excluding the CHESSIoT row), 80 were supported, while 74 remained unsupported. This translates to an average gap of 48.08%. On the other hand, in terms of IoT engineering support, out of the 66 possible points, only 26 were supported, leaving 40 unsupported, resulting in a gap of 60.6%. Figure 6.5 provides an overview of the study's outcomes, displaying the percentage of supporting and non-supporting aspects in both contexts.

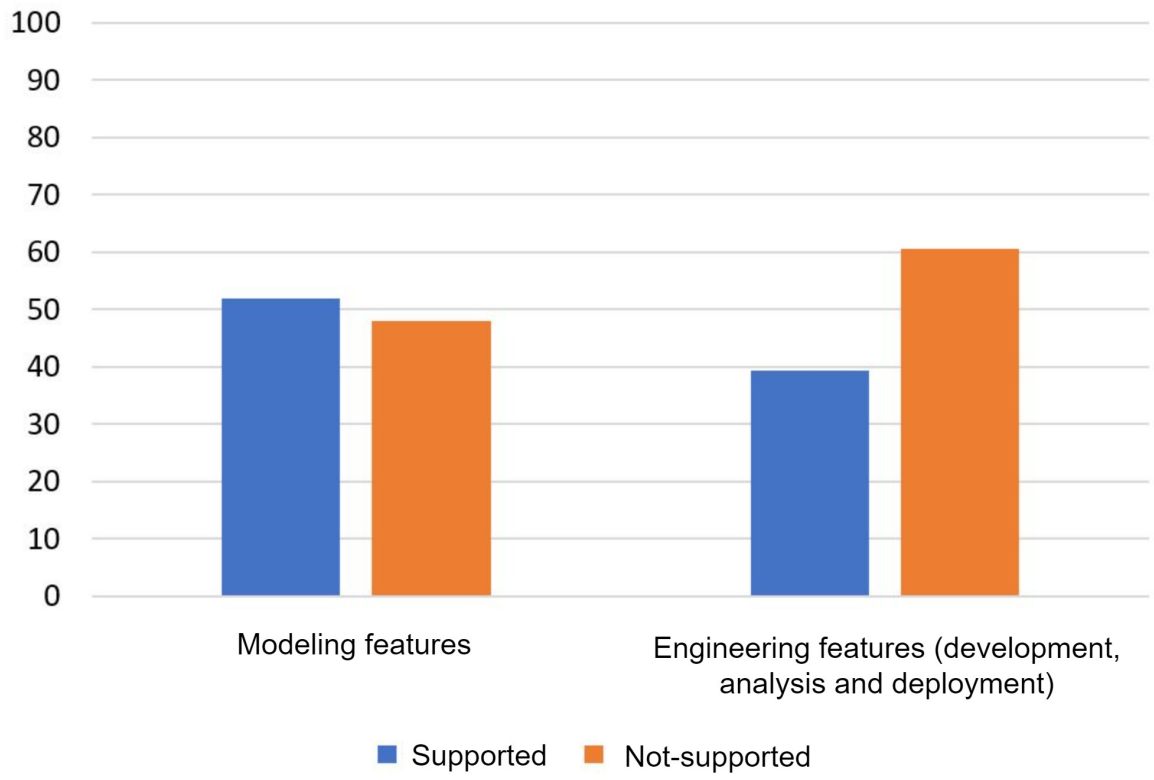


Figure 6.5: Overall comparative supporting results

According to Figure 6.5, when considering both modeling and engineering non-supporting aspects, the average gap where CHESSIoT could potentially contribute is 54.34%. While SimulateIoT [40] was found to be the best-performing platform, covering 13 out of the 14 basic modeling features of interest, it lacks certain features that CHESSIoT supports, such as multi-view modeling, system failure logic design, and run-time service provisioning.

In Sec. 6.3.3, we observed that SimulateIoT [40] does not support various engineering tasks related to IoT system analysis and takes a different approach to run-time management of IoT services. Although it may not be feasible for CHESSIoT to incorporate all platform-specific modeling capabilities supported by other platforms, the flexibility and customizability of the CHESSIoT platform allow for the potential integration of relevant modeling features in the future.

One notable gap in the analyzed IoT platforms is their performance in IoT system analysis, including verification, validation, and analysis of IoT systems under development [42]. Due to the complexity and scale of IoT systems, physical replication, and testing become challenging [69]. The lack

of standardized realistic reference models that accurately capture the interactions between sensors, apps, and actuators further exacerbates the issue. CHESSIoT addresses this by extending models with extra-functional properties and supporting analysis capabilities, as demonstrated in previous work on real-time schedulability analysis [42, 43].

While CHESSIoT contributes to both system modeling and engineering aspects of IoT systems, including code generation, safety analysis, deployment, and run-time service provisioning, there are limitations that need to be addressed in the future. The absence of standardization and agreed-upon reference architecture in the IoT domain often results in platforms not adequately addressing essential requirements. Although CHESSIoT's modeling language drew inspiration from the IoT-A reference architecture's multi-view approach [7], it deviates from it by introducing new concepts like failure logic modeling and deployment-related design. However, other modeling constructs related to information flow, security, and more are not yet covered in CHESSIoT.

Furthermore, the Fault-Tree Analysis approach used for safety analysis in system dependability analysis needs to comply with international software dependability and safety standards, such as [227]. While the current tool has not yet achieved international certification, it is being tested in industrial settings with large models and increased complexity to validate its effectiveness. Lastly, CHESSIoT currently lacks the means for testing generated software to support safety analysis results that reflect real-world conditions.

6.6 Conclusion

In this chapter, we present briefly introduce CHESSIoT engineering methodology. we have presented the general overview of CHESSIoT support for modeling, developing, analyzing, and deploying IoT systems. We have presented a motivating comparative analysis that draws the CHESSIoT contribution in relation to 12 existing platforms. To put things into context we have presented the CHESSIoT DSL which constitutes 3 main primary DSLs namely SystemDSL, SoftwareDSL, and DeploymentDSL. In order to assess the contribution of our DSL with respect to other existing DSLs, we have presented an evaluation between CHESSIoT and the same 12 platforms considered above in order to present their abilities for model IoT system entities that cover all layers. Finally, we have observed that in overall modeling features and engineering capabilities, an average gap of 54.34% gap was discovered to which CHESSIoT is potentially going to contribute.

Chapter 7

CHESSIoT safety analysis support for safety-critical IoT systems

Safety-critical IoT systems are those systems whose failure could result in loss of life, significant property damage, or environmental damage. Although the IoT industry seeks to cut development costs and bring new products to market as fast as possible, the safety analysis of such products is sophisticated and time-consuming, and sometimes it is not seriously taken into account, or it is simply neglected. Early-stage safety analyses can not only potentially reduce the cost of late failures, but they can as well help to easily trace and determine the source of the failure if something goes wrong within the system. As a contribution toward answering the fourth research problem (RP3), this chapter presents a two-fold safety analysis approach built on the extended CHESS Failure Logic Analysis (FLA) technique to support the safety analysis of IoT systems based on Fault Trees (FTs). In addition to its ability to generate the system FTs, the new Fault-Tree Analysis (FTA) approach automatically performs qualitative analyses by eliminating unnecessary paths as well as redundancies in the FT's events. Furthermore, the presented FTA performs a quantitative probabilistic analysis by calculating the system-level top failure event probability from the failure rates of the system's internal parts. This approach contributes to an improvement in terms of time spent performing safety analysis, as well as the correctness, consistency, and modularity of the analysis process. Furthermore, we employ a Patient Monitoring System (PMS) case study to demonstrate the efficiency of the proposed approach as well as the capability of the supporting tool. Finally, to assess the effectiveness of the presented approach, we used an experimental approach to compare it with 14 existing techniques.

The chapter is organized as follows: Section 7.1 presents an introduction. Section 7.2 presents the proposed safety analysis approach; Section 7.3 presents the evaluation mechanism while Section 7.4 presents the experimental results from the evaluation. In Section 7.5 concludes the chapter and draws some perspective work.

7.1 Introduction

Dependability is regarded as the ability of the system to provide services that can be trusted within a specific period [57]. It is mainly characterised by five essential attributes: *availability*, *reliability*, *maintainability*, *integrity*, and *safety*. In this chapter, we focus on *safety*, which is defined as the absence of catastrophic effects for the user(s) and the environment [4]. It is one of the major features that must be investigated and taken into account during the development phase of such systems to prevent further disasters. Again, from the healthcare system's point of view, when something goes wrong, it is critically important to react quickly and with high precision to isolate the danger before it happens. However, these systems are very complex and involve high-tech interconnected devices in which the process of discovering faults can be very long and tedious.

In the past, safety engineers relied on different informal design artifacts and documents, such as requirements documents, to measure the safety compliance of the system with less or no involvement of system engineers. Later, several approaches, such as [4, 30–33] (to mention a few), have emerged in the field. These approaches add a degree of automation during the analysis process, bridging the gap between the system and safety engineers. However, these approaches were designed and developed to fit domains such as aerospace, automotive, and cyber-physical systems. In some cases, they might not fully suit the IoT domain. This is mainly due to the inherent huge degree of heterogeneity in IoT ecosystems, not to mention its current rapid evolution.

This chapter presents both a modeling and an early-safety analysis environment for safety-critical IoT systems based on the Fault-Tree Analysis (FTA). The approach runs on top of CHESSIoT, a model-driven environment for engineering industrial IoT systems [42, 43]. The CHESSIoT environment is built on top of the CHESS tool [16], an open-source model-driven tool that offers cross-domain modeling, development, and analysis of high-integrity systems. CHESS supports different kinds of analysis including but not limited to real-time schedulable analysis [16], Contract-Based Analysis [111], and Quantitative Reliability Analysis based on Mobius [116]. In addition to this, CHESS also offers means to perform early safety analysis based on Failure Logic Analysis (CHESS-FLA) [44], however, that existing infrastructure is not suitable enough to support the IoT domain as well as not mature enough to support the Fault-Tree Analysis as one of the common and necessary artifacts used in the process of safety analysis [78, 79].

This chapter mainly focuses on CHESSIoT's extension to CHESS for supporting the early-safety analysis of IoT systems based on the FTA. The presented approach relies on and extends the existing CHESS-FLA infrastructure [44]. Normally, CHESS-FLA offers means to: *i)* model the system's failure behavior through the decoration of the system's simple components following the Failure Propagation Transformation Calculus (FPTC) annotation [33], *ii)* run the Failure Logic Analysis (FLA), *iii)* and propagate the analysis results back onto the original model [45]. Throughout the CHESS-FLA analysis process, the entire system's behavior is automatically determined solely from the composition of its elements. Thus the potential of automatic model-based safety analysis is significant. It is achieved by calculating the failure behavior from the composite parts up to the system level. This can help in predicting the impact of a component change or architectural change on a system in a very cheap way [33]. Furthermore, suppose an essential failure behavior occurs at the model system level. In that case, it will be easy to discover the source of the fault immediately and identify where the fault tolerance measures should be directed in the architecture to mitigate them.

The new proposed extension extends the CHESS-FLA by supporting the definition of the failure behavior of a simple component with no input ports. In addition to that, the extension support the generation of the system's complete Fault-Trees as well as performing qualitative and quantitative FTA Analysis. In the proposed approach, the qualitative analyses help in eliminating unnecessary paths as well as redundancies in the FTs' events. Quantitative analysis, on the other hand, allows the user to set the basic event probabilities and calculates the failure probabilities of an entire system from its constituent parts' failure event probabilities. This calculation is automatically performed following the well-known logic probabilities calculation mechanism techniques [46–48]. Aside from the FTA, the existing CHESS infrastructure also supports the generation of the Failure Mode Effect Analysis

(FMEA) table [83], one of the common and necessary artifacts used in the process of safety analysis.

Throughout the chapter, we present the evaluation mechanism and the outcomes of the experimental evaluation we conducted to better assess how the proposed approach performs compared to 14 existing approaches drafted from the literature. In doing so, five main features have been considered: *i)* support for system design modeling, *ii)* support the failure behavior modeling, *iii)* automated FT generation, *iv)* support for automated qualitative FT analysis, *v)* and support for automated quantitative FT analysis. In addition, we used a Patient Monitoring System (PMS) case study to demonstrate the effectiveness as well as the capability of the supporting tool. As a result, we summarize this chapter's key contribution as follows:

- We present CHESSIoT's system-level modeling environment for designing IoT systems, considering the system's physical aspects.
- We presented the CHESS-FLA extension to support the safety analysis suitable for the IoT domain.
- We introduce an automated Fault Tree generation approach that can handle large and complex models while still supporting critical features like event tracking and component sub-tree generation.
- We present both qualitative and quantitative FTA analysis approaches on the generated FaultTress.
- We present the experimental results from a relative evaluation mechanism conducted in comparison with 14 existing approaches in the literature.
- We present a Patient Monitoring System (PMS) case study to demonstrate the effectiveness of the proposed approach as well as the capability of the supporting tool

7.2 Proposed safety analysis approach

This section presents a model-based safety analysis approach based on automated calculus that can potentially reduce the time-consuming human effort and error-proneness of the IoT safety analysis procedure. In Section 7.2.1, we present the safety analysis process supported by the proposed tool; in Section 7.2.2, we describe the extended FPTC syntax while Section 7.2.3 describe briefly the transformation process. In Section 7.2.4, we introduce the FT generation process whereas Section 7.2.5 introduces the FTA approach detailing both the qualitative and quantitative approaches.

7.2.1 Model-based safety analysis process

IoT systems can experience failures due to various factors, including device age, data source problems, network issues, deployment environment, and external constraints. For instance, human error can also cause problems. The CHESSIoT safety analysis approach proposes an early safety analysis method using Fault-Tree Analysis, which involves annotating a system model with failure behavior rules using the Failure Propagation Transformation Calculus (FPTC) notation [33].

As illustrated in Figure 7.1, the safety analysis process typically commences with the *IoT system engineer* creating a model based on the gathered *system functional requirements* ① *in Figure 7.1*. These requirements are mainly acquired through close collaboration with the *client*. The system-level model encompasses the system's major functional components, sub-components, and their interconnections. These system components can be represented as blocks in SysML Block Definition Diagrams (BDD), which align with the abstract syntax meta-model illustrated in Figure 6.2. Internal Block Diagrams (IBD) are employed to illustrate the interdependencies between these components, facilitating the identification of error propagation paths. Each part or block can be assigned to a specific architectural subsystem or component. The physical architecture should possess extensibility

to accommodate the addition of new components or blocks as necessary. The entire safety analysis process is fully detailed in the next sections.

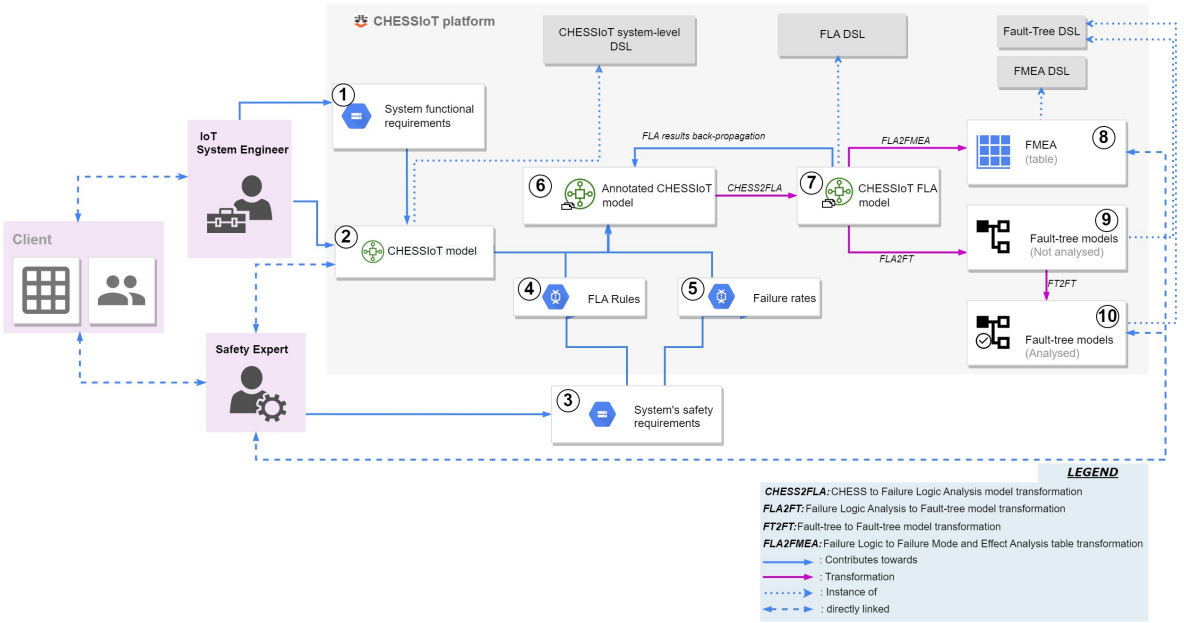


Figure 7.1: Safety analysis process

Once the system model is complete (see the *CHESSIoT model* ② in Fig. 7.1), it can be handed to the *safety engineer* for further safety analysis. The safety expert, similarly to the system engineer, can derive *safety requirements* ③ from the needs of the client, domain standards, and his or her expertise in order to ensure optimal safety. Starting from identifying the typical system-level failures, the safety engineer identifies the failure behavior for each component following the Failure Propagation Transformation Calculus (FPTC) notation. The FPTC technique enables the analysis of component-based systems with cyclic data, control-flow structures, and closed feedback loops. Such failure behavior referred to as *FLA rules* ④ are annotated to the system's simple comments to illustrate how failures might occur in a system component and how they are propagated from one component to another. At this stage, the safety engineer can additionally set *the component's failure rates* ⑤ to be used for quantitative analysis.

7.2.2 FPTC Calculus

The FPTC technique solves a significant limitation by allowing the analysis of component-based systems with cyclic data, control-flow structures, and closed feedback loops. The extended annotations explain how failures might occur in a system component and how they are propagated from one component to another. Based on its nature, a function/component can propagate a failure (carrying a failure from input to output), transform a failure (changing the nature of a failure from input to output), act as a source of failure (creating a failure despite no failure in input), or act as a sink (avoiding the failure to be either propagated of transformed). The following three abstract categories of failure modes are assessed: *service provision failures*, such as the omission or commission of the output; *timing failures*, such as the early or late delivery of the output; and *value domain failures*, such as the output value being out of a valid range, stuck, biased, exhibiting a linear or non-linear drift, or erratic behavior. In addition to that, a *noFailure* annotation is used to indicate a no-failure mode at the input port. Table 7.1 describes different failure modes and their description.

First integration of the CHESS-FLA in CHESS was done in [44] with the support for FI^{4}FA (Formalism for Incompletion, Inconsistency, Interference, and Impermanence Failures Analysis) [228], which is an extension of FPTC that takes into consideration Incompletion, Inconsistency, Interfer-

Failure type	Description
Early	output is provided too early
Late	output is provided too late
ValueCoarse	output out of range in a detectable way
ValueSubtle	output not in range in an undetectable way
Omission	no output is provided
Commission	an output is provided when not expected

Table 7.1: Failure types

ence, and Impermanence Failures, and their corresponding countermeasures. However, in the previous version, it was impossible to express a component's failure behavior when it possesses no input ports.

In the existing infrastructure, when modeling the failure expression at a certain port, always the port's name and the failure type are always necessary. This means that a *"noFailure"* mode is assigned to that port to depict the event in which an internal failure has occurred in the systems causing a failure at any output post of interest. However, it is not always obligatory for a component to an input port for it to fail. For instance, in IoT domain, a temperature sensor's role is to sense the surrounding environment and relay the information to the connected parties. In that case, the sensor needs only a physical output port in order to function properly. The new extension allows such internal failure scenario to be expressed using a *"(*)"* notation. The new extended syntax is depicted in Listing 7.1.

```
FLA:"LHS"=>"RHS"; #left/right hand side of an expression
LHS=portName."bL"|portName."bL"(,portName."bL")+|"(*)"
RHS=portName."bR"|portName."bR"(,portName."bR")+
bL=wildcard|"bR"
bR=noFailure|"FAILURE"
FAILURE=early|late|commission|omission|valueSubtle|valueCoarse
```

Listing 7.1: Extended FLA syntax expression

7.2.3 CHESS2FLA transformation

The *Annotated CHESSIoT model* ⑥, produced by the system expert as previously explained, gets automatically transformed by means of the *CHESS2FLA* model-to-model transformation to generate the *CHESSIoT FLA model* ⑦. During the transformation, each component is essentially considered a black box that can only exchange data through its ports. The FLA technique automates the calculation of a complete system failure behavior starting from the failure behavior rules of its separate composite components and interconnections. This, in turn, means that the failure behavior of composite elements is also determined by the failure behaviors of its instantiated simple components and their internal decomposition. Simple components have no other parts and rely on the failure behavior defined in the previous stages. When the failure modeling is finished, the model undergoes a CHESS2FLA model-to-model transformation which transforms it into an FLA model following the meta-model presented in Figure 7.2. This transformation is domain agnostic, and it does not consider any domain-specific construct other than linking each component and its final state of failure into a single model instance.

As seen from the FLA meta-model, a composite component represents a subsystem that contains one or more sub-components. As mentioned in the previous stages, this component does not possess the failure behavior by itself, but it relies on its sub-components to determine its failure behavior. On the other hand, a simple component represents a functional component that can contribute to system failure. Each component contains input and output ports and the annotated failures (according to FLA rules) for simple components. In contrast, in composite components, the propagated failure is assigned to the ports. Furthermore, each simple component is assigned rules where each rule contains input and output expressions reflecting failures and their respective ports. During the transformation, the extended notation of the internal failure of a component with no input ports creates a unique virtual port assigned with a "noFailure" failure type at the component input port to reflect the idea of the component's internal failure source.

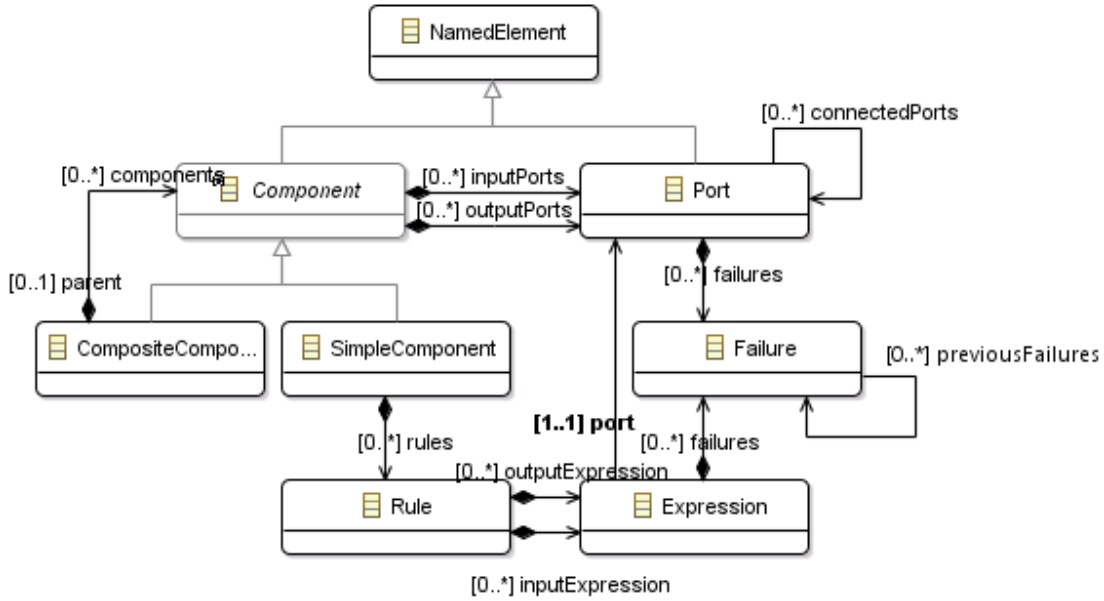


Figure 7.2: CHESS-FLA meta-model [3]

At each level of the FLA analysis, the results are back-propagated onto the original model to assign each component's failure state to be reflected in the model. The final failure state at simple and composite components as well as at the system level is reflected when the analysis is done. The system *FT* ⑨ and FMEA ⑧ table can be automatically generated and analyzed before being sent back to the safety expert for consultation. If something is wrong, changes can be made before the final inspection. In the following sections, we briefly review each step of the supported analysis process. Although the FMEA analysis is included in the CHESSIoT safety analysis process, this paper focuses primarily on the FT-based analysis approach.

7.2.4 Fault-Tree generation

The FT generation process is performed following a FLA2FT model for the transformation of a CHESS-FLA model into a conforming FT model. Figure 7.3 shows an FT meta-model adopted from [4]. The FTModel element is the top element of the tree, and it is instantiated for each failure that propagates to the system output ports during the FLA analysis. Each FTModel element contains a logical network of events and gates that together form an FT. The entire FT generation process is covered in the following sections.

Fault-tree events

In our proposed approach, each event can be graphically identified in the FT from its unique identifier (ID). An event ID is defined as a pair of "failure names" and "port names" in a given component. This ID never changes through the FT generation as well as in the FT analysis process. This can potentially help in event tracking when comparing generated and analyzed FTs. In addition to that, each event has its own name which by default combines the information regarding its corresponding failure and its effect in a given component. The effects can be of type *top failure* type at the system level, *local failure* caused by the system's intermediate failures across the tree, *injected failure* resulted from injected faults, and *internal failure* resulted from the component's internal faults. In the next part, we are going over the various event types that we use to construct the FT and we will describe how events are generated throughout the transformation process.

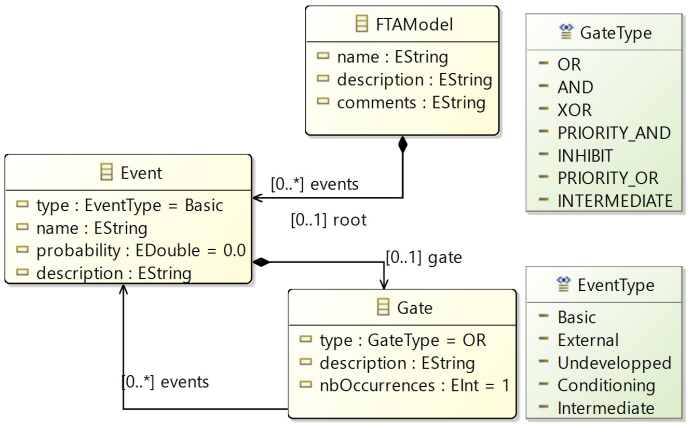


Figure 7.3: FT meta-model [4]

- **Basic events:** A simple component may suffer different kinds of malfunctions, generating either one or more kinds of internal failures. One or more notations may be required to define such events for a given component. A basic event is used to represent a failure that is initiated inside a simple component. This can be basically referred to as a simple component acting as a source of failure. In this case, a failure condition is present on any of the output ports despite no failure at its input port. In case a simple component does not possess any input port, the newly developed approach allows the definition of such condition following Expression *(7.1)*. On the other hand, in case a simple component possesses one or more input ports, its failure behavior can be defined by explicitly initializing all the input failures of the component with a *"noFailure"* condition (Expression 7.2). Figure 7.4a shows the basic event representation in an FT resulting from an internal failure of a component.

$$(*) \rightarrow p.failure_{(out)}; \tag{7.1}$$

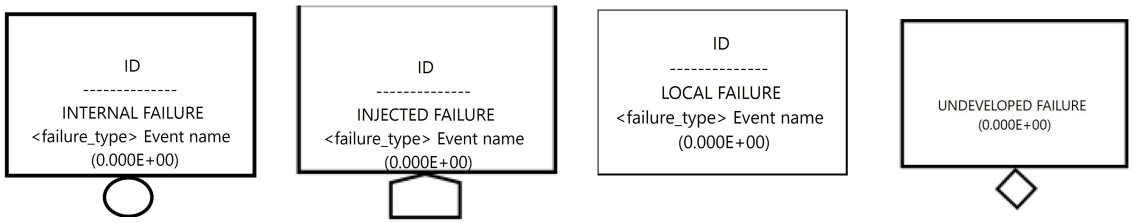
$$p_1.noFailure, ..., p_n.noFailure \rightarrow p_{(out)}.failure_{(out)}; \tag{7.2}$$

NOTE: *Considering p_1,p_2 to p_n to be the input ports and $p_{(out)}$ to be an outport of a simple component. If any of their corresponding failure condition is different to "noFailure", then the above condition is not met, so, all "noFailure" conditions on other ports are ignored as they do not contribute toward to the logical failure behavior of the component.*

- **External events:** External events are used to represent failures that can be introduced from the environment outside the system boundaries. CHESS provides the possibility to inject failures in the system through the system-level input ports. These faults are modeled with a comment annotation with *«FPTCSpecification»* stereotype attached to the relevant input port where the fault is being injected in SysML block diagrams (the case study in Section 7.4 demonstrates this concept in more details). The injected fault comment specifies the type of failure attribute being injected into the system. This fault injection can also be done on a composite sub-system under analysis. Figure 7.4b shows a graphical representation of an external event resulting from an injected fault in the system.

- **Intermediate events:** An intermediate event is used to describe the local failure effects resulting from a logical output of one or many events. In the presented approach, these events are generated to represent the failure condition at the input or output port(s) of the simple component resulting from an internal failure or other failure condition from the outside of that simple component. It is also used to represent the top event of an FT. Figure 7.4c is used to represent an intermediate event.

- **Underdeveloped events:** The underdeveloped event is used to specify an event resulting from a failure in which we do not have sufficient information about it. This can basically happen when a failure is introduced at the input port of a simple component without a preceding definition of how it was been propagated or injected at that input port. During the FT generation process the symbol in Figure 7.4d is used.



(a) Basic event **(b)** Injected failure event **(c)** Intermediate event type **(d)** Undeveloped event

Figure 7.4: Event types

Failure propagation

Failure propagation occurs in a component when a single input port failure condition of a component is directly transferred to the output ports of the same component without changing its nature. This failure propagation can be modeled in CHESS-FLA using the notation in Expression 7.3. A propagation also occurs between two connected components, when a failure condition at the output port of the preceding component is transferred to the input port of the following component.

$$p_{(in)}.failure_1 \rightarrow p_{(out)}.failure_1; \tag{7.3}$$

Failure transformation

A failure transformation occurs within a component when a failure condition present at the input port of a simple component is converted into another type before reaching the output port (Expression 7.4). A failure transformation can also occur when more than one failure expression of any type the exception of a "noFailure" or "wildcard" at multiple input ports is transmitted to a single output port (Expression 7.5). Even if the failure has the same type, the fact that the component converts two failures at its input ports to a single failure at the output port is regarded as a failure transformation.

$$p_{(in)}.failure_1 \rightarrow p_{(out)}.failure_{(out)}; \tag{7.4}$$

$$p_{(in1)}.failure_1, ..., p_{(inN)}.failure_N \rightarrow p_{(out)}.failure_{(out)}; \tag{7.5}$$

Fault-tree generation process

The system FTs are generated through a series of model-to-model transformation mechanisms written using the Epsilon Transformation Language (ETL) [229]. The process starts by instantiating a number of FT objects equal to the number of failures that propagates to the output port(s) of the system. At this stage, each error that propagates to the output port(s) of the system is represented in its own FT. Note that when a "noFailure" condition is propagated to the output, it is ignored. This technically means that the system acts like a failure sink and it is able to mitigate its propagation to the output of the system, which is also true for all other sub-systems. To achieve that, the Algorithm 1 is followed.

In the next steps, each FT is built separately and recursively. The initial action involves the creation of a top event among all. A top event is generated as a result of the failure propagation to the system output port. In terms of logic gates used in the FT, only "AND" and "OR" gates are adopted. An

Algorithm 1: Instantiate fault-tree

```
for p in allPorts do
    if p is output of the system then
        for f in failures assigned to p do
            if f is not "noFailure" then
                Create an FT relative to the failure f and p;
                Add FT to FTs sequence;
            end
        end
    end
end
```

AND gate is used to indicate a failure transformation from an input to an output port of a component (see Section 7.2.4). An **OR** gate, on the other hand, is used to depict a failure propagation situation (see Section 7.2.4). The OR gate can also depict a scenario in which one or more failure outputs from distinct components are passed to the input of the following component. The whole FT generation population algorithm is described in Algorithm 2.

Algorithm 2: FLAComposite2FT rule algorithm

```
Data: FLA composite component (system-level)
Result: FT model
count=0;
for port in allPorts do
    if port is output of the system then
        for f in failures assigned to port do
            if f.name is correspond to an FT then
                Create an intermediate event ← TOP FAILURE;
                Assign an OR gate to it;
                Add it to its corresponding FT;
                for con_port in port.connectedPorts do
                    recurseFT(f,con_port,FT,topEvent);
                    //Call Algorithm 3
                end
            end
        end
    end
end
```

When the top-event creation is done, the intermediate events are created and populated into the FT, based on the failure expressions and the components they are assigned to. The FT population involves a recursive transformation process in which, as indicated by the FLA meta-model (Figure 7.2), from a component, we can have information on ports, and from ports, we can get to rules, rules to expressions and back to the components. So, at this stage, the only crucial stopping case is when the transformation hits a condition matching an internal, underdeveloped, or injected failure. For instance, in Figure 7.5, a simple transformation example with indications showing a simple transformation mapping of the Expression 7.5 is shown. From the example, each of the output expressions is mapped to an output event of a logical combination of the input expressions. Each input expression is mapped to an event and the type of such event is determined by the expression condition. In addition to that, the logic gate is defined based on the nature of the input expressions to satisfy the failure propagation and

transformation concepts. A brief description of the following algorithm is described in Algorithm 3.

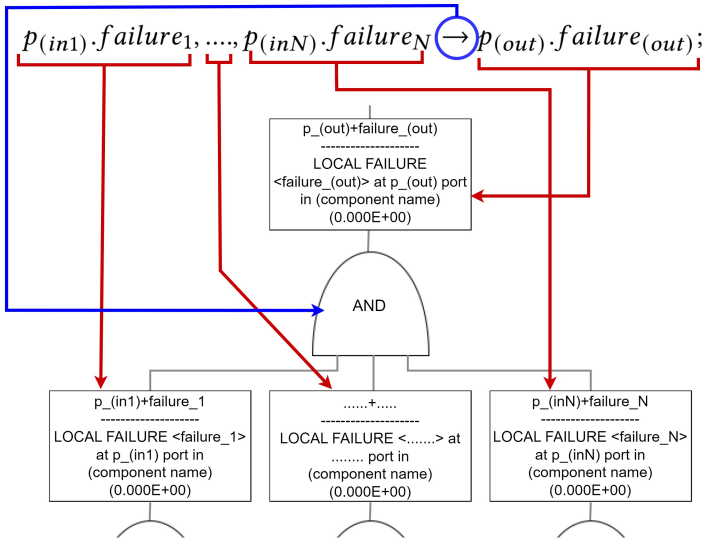


Figure 7.5: Expression 7.5 corresponding tree

In the newly developed environment, it is possible for a safety expert to assign failure rates of any of the events leading to a basic failure condition. These events include the internal failure condition and the injected failures. In addition to the failure rate assignment, the user is also able to add a failure description in a textual format to reflect the proper cause of the failure. This is potentially important when, for instance, a simple component might have two different internal failure conditions leading to two different outputs. For instance, an aging device can still work by providing the wrong output leading to a *valueCoarse* or *valueSubtle* failure at the output port whereas a blown device fuse will automatically halt its functionality, therefore, leading to an "omission" failure at the output port. Assigning such information to the FT model will eventually improve its readability. This assignment is done way before the FLA2FT transformation process and when the transformation finishes, a probability file is generated separately from the model in an Excel file format. This file is later loaded into the model to support the quantitative probabilistic analysis process.
Note: the full transformation code on FT generation can be found in the appendix B.1

7.2.5 Fault-Tree Analysis

The proposed approach support both the qualitative and the quantitative FT analysis through the use of rigorous model transformation techniques. In this section, we are going through the supported analysis.

FT Qualitative analysis

The FT qualitative study is conducted using an FT2FT model-to-model transformation (ie: transformation from (9) to (10)) in which the FT meta-model in Figure 7.3 serves as both the source and the target meta-model. This effectively creates new FT representations in the workspace, permitting users to reuse both the generated and the analyzed FTs at the same time. The goal of this qualitative analysis approach is to provide a new representation of the existing FT that only includes the essential representations. Although the current qualitative analysis does not fully reflect the calculation of the minimal event sets needed for a system to fail (minimal cut-sets [230]), it does provide a much shorter and more readable FT that still reflects the goal for the analysis.

During the qualitative analysis process, the following actions are performed:

- ***1. Removal of internal component failure propagation:*** One of the goals of an FT is to help users to discover and trace down the source event of a system failure in a more easy and intuitive

Algorithm 3: Recurse FT operation

```
Data: current_failure as f, current_port as p, current_FT as FT, event_to_construct as eG
Result: final populated FT
if p is of not system's port then
    if p is of a "Simple Component" then
        for outExp in p.owner.rules.outputExpressions do
            for f1 in outExp.failures do
                if f1==f then
                    Create an intermediate event e0 ← LOCAL FAILURE;
                    Assign gate to e0 based failures at the p port;
                    Add e0 to eG;
                    Add e0 to FT;
                    for inpExp in p.owner.rules.inputExpressions do
                        for f2 in inpExp.failures do
                            if f2 is not a "wildcard" then
                                if f2 is a "noFailure" then
                                    Create a basic event eT ← INTERNAL FAILURE;
                                    Add eT to eG;
                                    Add eT to FT;
                                else
                                    Create a intermediate event e1 ← LOCAL FAILURE;
                                    Assign an OR gate to e1;
                                    Add e1 to e0;
                                    Add e1 to FT;
                                    for p1 in inpExp.port.connectedPorts do
                                        recurseFT(f2,p1,FT,e1); //Recurse from the start of the
                                          Algorithm 3
                                    end
                                end
                            end
                        end
                    end
                end
            end
        end
    else
        for p1 in p.connectedPorts do
            recurseFT(f,p1,FT,ev); //Recurse from the start of the Algorithm 3
        end
    end
else
    Create a external event eX ← INJECTED FAILURE;
    Add eX to eG;
    Add eX to FT;
end
```

fashion. As described in Section 7.2.4, the internal component failure propagation occurs when a single input port failure condition of a component is directly transferred to the output ports of the same component without changing its nature. Although keeping such information in the FT is important, when the model becomes increasingly big, this information can be very exhausting to glance at. Therefore, each path meeting such a condition is omitted and removed from the FT. This process drastically reduces the vertical magnitude of an FT but it does not change its nature.

- ***2. Removal of external component-to-component failure propagation:*** This refers to a condition in which a component-to-component propagation is solicited from a single channel in the FT. For instance, if a single failure condition at the input port of a component is propagated from a single source, then this information is omitted in the analyzed FT. Mainly on the basic events, events involved in a failure transformation, and the top event are kept in the FT.
- ***3. Removal of basic events redundancy:*** A single failure can be initiated from a single source and passes through different propagation paths until it reaches the output port(s). If, in all the propagation paths, no transformation occurs, then, from the output failure conditions, only one path is considered, and from that path, all intermediate propagation representations are removed according to the two previous rules.

During the transformation process, only the paths that satisfy the internal or external failure transformation are kept in the tree. This is to help users to only care about the important information when tracing the origin of the failure. One special case of the propagation that is kept in the tree is when the FT that has to be analyzed contains a single path in which a single basic event propagates up the tree all the way to the top event. Then, in such cases, only the top event and the basic event are kept in the analyzed FT. Finally, each of the omitted intermediate paths as well as each gate resulting from a simple component internal failure transformation is replaced by a feed-forward intermediate gate to enhance the FT readability.

For example, taking the generated FT branch shown in Figure 7.6, the event ⓪ is obtained from a logical "AND" output from 3 subsequent paths, which makes this event a result of a component-to-component external transformation. Starting from event ② ("omission") failure at the input port to the event ① ("commission") event at the output port, indicates internal failure transformation. So in such a case, event ① and its next gate are kept permanently, while event ② is kept temporally for future analysis. Further down to event ③ ("omission") up to event ② ("omission") is component-to-component failure propagation, so event ③ will be permanently removed while event ② will be kept again temporarily for further analysis.

Next, we remain with event ④ ("omission") up to ② ("omission") which is a propagation as well (omission-to-omission). From here, normally event ④ is supposed to be removed; however, as event ④ is a basic event, event ② will be removed instead. Finally, the whole omitted part of the tree will be substituted by a feed-forward intermediate gate to enhance the readability of the FT. The final version of the FT is provided in the right-hand figure, 7.6(b). To sum up, we would say that "the internal failure ④ leading to an *"omission"* at the output port of a basic component had transformed into event a *"commission"* failure event ① at some point in the system, in which then combined with the other two failure sources had caused an *"omission"* at the top level of the system.

FT Quantitative analysis

The quantitative probabilistic analysis is meant to automatically calculate the system-level (top event) failure rate. In the proposed approach, the user is able to assign the failure probability rates of the basic failure events such as internal failure and injected failure. This information can be supplied from the device manufacturer's data sheet as well as the safety experts. In safety engineering, the device failure probability is often considered to be extremely low and often expressed as failures per million (10^{-6}), particularly for individual components [231]. The probability calculation follows a

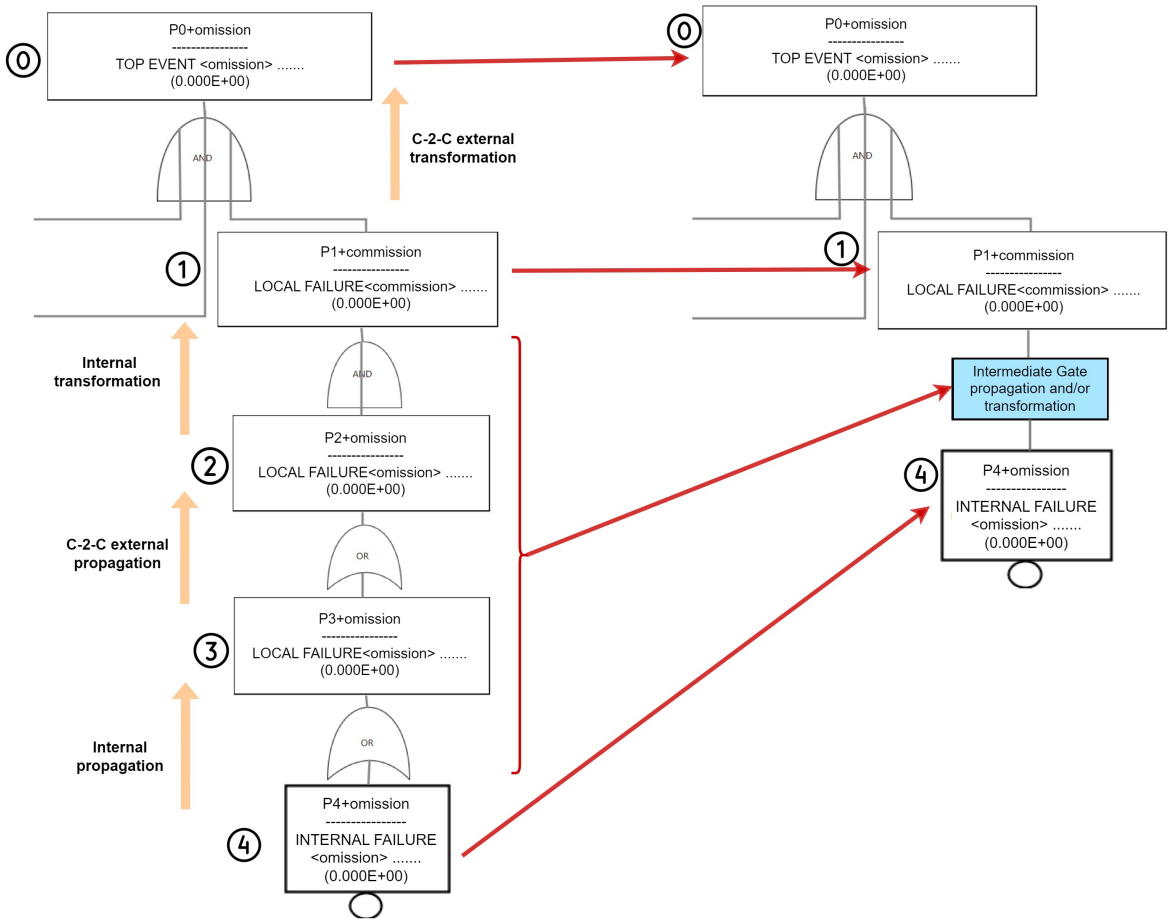


Figure 7.6: Qualitative transformation example (a) before, (b) after

widely used formula for conducting a logical output of an "AND" or an "OR" gates in the FT [46, 47]. The output of an "AND" gate means that the output event will only happen when a combination of independent events occurs at the same time. On the other hand, the output of an "OR" gate implies that the output event will occur if any one of the input events occurs.

For each FT to be analyzed, the system failure rate (the top event probability) is calculated following a recursive calculation of the intermediate probabilities to the intermediate events. Based on the probabilities of the basic events, the probability values of their parent event can be calculated from input event probabilities. The probability calculation follows the formula in Figure 7.7. Let N be the number of input events and P_{in} the probability of the input event, the output probability P_{out}, for both "AND" and "OR" gate types, is calculated as follows:

$$P_{out} = \begin{cases} \prod_{in=1}^{N} P_{in} & \text{for an } AND \text{ gate} \\ 1 - \prod_{in=1}^{N} (1 - P_{in}) & \text{for an } OR \text{ gate} \end{cases}$$

Figure 7.7: Probability calculation formula

During the probability calculation process, in the case of an internal component failure transformation condition, the probability corresponding to an input port failure event is forwarded to the output port event. When the qualitative analysis finishes, this in turn implies that the probability of an event at the input of an intermediate gate is forwarded to the immediate output event. In addition to that, in an event resulting from an undeveloped event being fed into an "OR" gate, that branch probability

is first set to zero while in the case of an "AND" gate, the branch probability is set to 1. This in fact does not affect the probability calculation process as 0 and 1 are neutral values in the addition and multiplication process respectively.
Note: the full transformation code on both quantitative and qualitative FT analysis can be found in the appendix B.2

7.3 Evaluation process

In this section, we describe the approach used to evaluate the performance of the proposed approach in supporting the safety analysis of safety-critical IoT systems with respect to other existing approaches in the literature. In Section 7.3.1, we present the following evaluation process; in Section 7.3.2, we introduce the case study exploited to support our evaluation; finally, Section 7.3.3 describes the research questions we targeted to address.

7.3.1 Evaluation process

The evaluation procedure we adopted followed six different steps. We first identified a use case that precisely fits in the context of safety-critical IoT systems. Next, we defined research questions that primarily focus on evaluating and demonstrating the effectiveness and potential of the proposed approach. Third, we briefly assessed the key features offered by our system and the supporting tool. This stage allowed us to recognize the most powerful safety analysis tools to be compared to demonstrate our approach's superiority. Following that, we presented the experimental results, emphasizing answering the posed research questions. Because our proposed approach can be employed in areas other than IoT, the evaluation approach will also consider the widely used FTA approach used in system engineering. Finally, we compared our approach with the most relevant state-of-the-art platforms, including but not limited to ISOGRAPH Reliability Workbench [169], FT generation approach based on CHESS-FLA implemented in [3], and the safety analysis approach for IoT presented in [180].

7.3.2 Motivating example: Patient Monitoring System (PMS)

Due to the general rapid evolution of electronics and information technology, more powerful bedside patient monitors capable of complex bio-signal processing and interpretation are becoming available, and they are usually equipped with some highly specialized communication interfaces [232]. This goes hand in hand with the huge advances in IoT technologies allowing the integration on such devices of the capability to connect to the internet, which makes it possible to monitor the health state of multiple patients remotely. To support our evaluation process, we adopted an "Efficient Patient Monitoring for Multiple Patients Using WSN" case study [5]. The case study is an advanced system capable of reliably monitoring the multiple parameters of up to six hospitalized patients simultaneously in real time.

The system investigates the potential of employing Wireless Sensor Networks (WSN) to reliably and wirelessly collect multiple parameters such as blood pressure, temperature, electrocardiography (ECG), electroencephalography (EEG), and pulse oximeter (SPO2). These parameters are collected through a set of sensors placed on different parts of the patient's body. For instance, the ECG Sensor is placed on the chest and on the limbs to extract the patient's heart rate data, the EEG sensor is placed on the patient's head to read electrical activity generated by the brain. Furthermore, the Blood Pressure sensor is placed on the arm to detect the level of the pressure in the blood; while the SPO2 sensor is placed on the patient's finger to measure the oxygen saturation of a patient's blood; and, finally, the Temperature sensor is placed to any part of the body to measure the temperature. Figure 7.8 describes the high-level architecture of the system.

As a result, the recorded parameters are wirelessly transmitted to a computer running PMS software, which feeds them on the monitor screen in the doctor's office. The software can also wirelessly

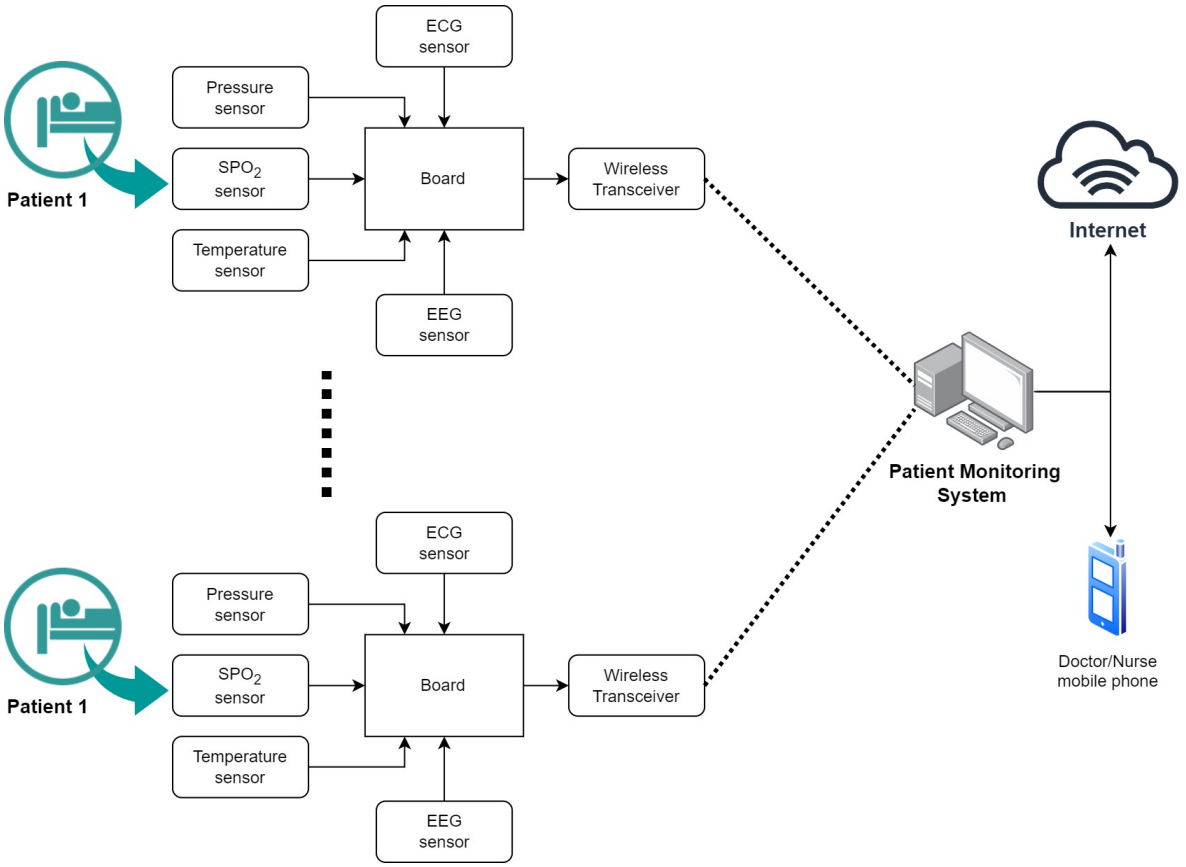


Figure 7.8: Basic architecture of Patient Monitoring System [5]

send alarming messages to the doctor's phone, if they are not present, to respond to the patient's requests. The architecture in great detail is discussed in Section 7.4.2.

7.3.3 Research questions

We study the performance of our proposed approach by considering the following research questions:

- ***RQ1: How does the presented approach distinguish itself from state-of-the-art techniques?*** We simply did a short review of the existing methodologies in relation to what the proposed approach offers in order to derive its unique contribution.
- ***RQ2: Does the system-level modeling infrastructure address all the aspects of modeling a multi-layered IoT system suitable for safety analysis?*** Apart from the modeling language, does the modeling approach captures all the required information to facilitate the safety analysis? For instance, are all the provided information taken into consideration when performing the analysis?
- ***RQ3: How well do the proposed FLA rules efficiently reflect the system failure behavior leading to the system top failure events?*** We look at the efficiency of the derived rules from the possible system failure events in clearly supporting actual logical analysis leading to the expected results.
- ***RQ4: Does the proposed FT qualitative and quantitative analysis improve the existing FT analysis techniques?*** We will concentrate reliability of the result from the supported automated calculus with regard to the existing approaches. We will also have a look at the final FTs and how they better describe the important system failure paths.

7.4 Experimental results

In this section, we report the results of the conducted experiment mainly focusing on answering the research questions formulated in Section 7.3. In Section 7.4.1, provide a brief assessment of the related tools in response to RQ1. In Section 7.4.2, present the PMS system's model architecture while attempting to answer RQ2. Section 7.4.3 present the system failure behavior modeling approach to answer RQ3. Finally, Section 7.4.4 present the results of the FT analysis aimed at answering RQ4.

7.4.1 Short literature review (RQ1)

RQ1: How does the presented approach distinguish itself from state-of-the-art techniques?
To sufficiently answer the research question, we needed to first understand the available tools that employ the FTA technique to conduct the safety analysis in the literature. We did not only focus on tools that solely support the IoT domain since we wanted to understand the methodologies used by the existing tools in their FT formalism and analysis process in comparison with our proposed approach. In the end, 14 different platforms were found in the literature with much closer relations to our approach.

Search and selection process

The search and selection procedure was divided into four major stages. Since the purpose of the chapter is not to undertake an empirical study, we did not conduct this review using well-known databases, but we only relied on Google Scholar results. We performed an automatic search using the keywords: "Model-based Safety analysis in IoT systems," "Fault Tree analysis in IoT systems," "Fault Tree analysis from SysML models," and "Model-based safety analysis or Fault-Tree analysis." The goal was to offer us a sample of current publications on topics related to such information in the search strings. Because each search returned a large number of results, we only analyzed the first two pages of the results. The second phase was conducted for each search in which we filtered out such results by reading through the title and abstracts. To pass this step, an approach title and abstract must include at least one of the terms "Fault-Tree Analysis" or "Model-Based Safety Analysis." Following that, we skimmed through the selected articles to exclude those that merely present an analysis approach but no supporting tool, resulting in a total of 13 approaches. Finally, we included ISOGRAPH [169] as one of the most extensively used workbenches in the industrial domain for dependability analysis, which lead us to a total of 14. Table 7.2 presents a list of selected primary approaches.

Results

To better answer RQ1, we have defined a set of fundamental features we think a safety analysis tool should possess. These features were evaluated on the selected approach in comparison with the proposed infrastructure. The features such as supporting system design modeling, failure behavior modeling, automated FT generation, performing qualitative FT analysis, and performing quantitative FT analysis. Table 7.3 summarizes the findings of the study, in which for each approach, a "Yes" or "No" label was used to indicate if that approach supports that particular feature. As indicated, all 14 approaches are represented against the 5 features. It can be seen that at least one feature is supported at least once. In this section, we go over the result of the study and have some discussion in line with the features defined above.

- *Support for system modeling*: This feature assesses whether the suggested tool supports system design before proceeding with its safety analysis. Conducting safety analysis needs to go hand in hand firstly with the design of the system under analysis. We believe that integrating the design and analysis infrastructure can improve transparency and consistency among system and safety experts. According to Figure 7.9, 78.6% of the selected approaches has the infrastructure for designing the system before starting the analysis. Table 7.3, on the other hand, indicates that

Table 7.2: Selected approaches

Tool	Title	Year	Type
ISOGRAPH [169]	Fault tree analysis in reliability workbench	1980s	Community tool
Haider et al. [3]	FLA2FT: Automatic generation of fault tree from CertoFLA results	2018	Conference
JARVIS [170]	A framework for model-driven engineering of resilient software-controlled systems	2021	Journal
Mehnni et al. [171]	Automatic fault tree generation from SysML system models	2014	Conference
Alshboul et al. [4]	Automatic derivation of fault tree models from SysML models for safety analysis	2018	Journal
Yakymets et al. [172]	Model-based system engineering for fault tree generation and analysis	2013	Conference
Hamed et al. [174]	Fault tree analysis for reliability evaluation of an advanced complex manufacturing system	2018	Journal
Silva et al. [180]	A dependability evaluation tool for the internet of things	2018	Journal
Chen et al. [181]	Application of fault tree analysis and fuzzy neural networks to fault diagnosis in the Internet of Things (IoT) for aquaculture	2017	Journal
Xing et al. [182]	Reliability modeling of mesh storage area networks for Internet of Things	2017	Journal
MetaFPA [178]	Transformation of failure propagation models into fault trees for safety evaluation purposes	2016	Conference
Clegg et al. [175]	Integrating existing safety analyses into SysML	2019	Conference
smartIflow [233]	Model based safety analysis with smartIflow	2017	Journal
Xiang et al. [177]	Automatic static fault tree analysis from system models	2010	Conference

approaches such as [175, 178, 181] do not provide such a feature and instead rely on manually created FT models, which are then transformed into FT graphs. In contrast to the previous approach, our proposed approach completely supports this feature by providing an environment in which system main blocks and sub-systems can be decomposed and analyzed separately. Even though approaches such as [4, 171, 175, 176, 234] extend the SysML language in the same way that we do, our environment is more user-friendly due to the advanced component-based and multi-view modeling infrastructure, where each view has its own underlined constraints that enforce its specific privileges on model entities and properties that can be manipulated [42].

- *Support for failure behavior modeling*: This feature determines whether the proposed approach provides mechanisms for explicitly stating the system failure modes as well as the system failure behavior, both of which contribute to the generation of the FT. As shown in Figure 7.9, 85.7 % of the investigated approaches support this feature, which is a good number given that it is the key driver for FT generation. In our approach, we extend FPTC [33] rules when modeling the system failure behavior because we consider it to be straightforward to understand. Other approaches use formalism including, but not limited to, *"IF-THEN"* or logical math association expressions to formalize that ([171, 178, 180, 181]). Nonetheless, all these approaches lack the concepts of external failure injection as well as internal failure transformation and propagation. On the other hand, [172] uses a formal method approach in modeling system failure logic; however, the complexity of formal method formalism can be difficult to handle. Furthermore, [4] depends on annotating failure information in the model state machines, which can be a difficult and time-consuming task due to the complexity of state machine definition.

- *Perform automated FT generation*: This feature determines whether the proposed approach automatically generates the FT from the model rather than manually constructing it. This is one of the main motivations for our proposed approach since we believe that automating the FT generation process is critical to reducing the time safety engineers spend in performing the safety analysis as well as increasing transparency in the process. According to the findings, around 64.3% of the techniques support such a feature. One commonly used tool in the industry

Table 7.3: Results from the studied approaches

Approach	System modeling	Failure behavior modeling	Perform automatic FT generation	Perform qualitative FT analysis	Perform quantitative FT analysis
ISOGRAPH [169]	Yes	Yes	No	Yes	Yes
Haider et al. [3]	Yes	Yes	Yes	No	No
JARVIS [170]	Yes	Yes	Yes	Yes	No
Mehnni et al. [171]	Yes	Yes	Yes	No	No
Alshboul et al. [4]	Yes	Yes	Yes	No	No
Yakymets et al. [172]	Yes	Yes	Yes	Yes	Yes
Hamed et al. [174]	Yes	No	No	Yes	Yes
Silva et al. [180]	Yes	Yes	Yes	Yes	Yes
Chen et al. [181]	No	Yes	No	Yes	No
Xing et al. [182]	Yes	No	No	Yes	Yes
MetaFPA [178]	No	Yes	No	No	No
Clegg et al. [175]	No	Yes	Yes	No	No
smartIflow [233]	Yes	Yes	Yes	Yes	No
Xiang et al. [177]	Yes	Yes	Yes	No	No
CHESSIoT	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

(ISOGRAPH [169]) does not support this; this could be due to the scale at which FTs can be used, not only in safety analysis but also in other domains like reliability analysis, risk analysis, and so forth. However, as technology advances, we strongly believe that this should change in order to remain relevant in the market. Our approach provides a solid FT generation mechanism that can support large and complex models with advanced features including but not limited to event tracking, component sub-tree generation, analysis, and undeveloped branch identification.

- *Perform automated qualitative FT analysis*: This feature determines whether or not the proposed approach supports any means for performing qualitative analysis on the generated FT, including detecting minimal cut-sets, FT path reduction, FT event redundancies, and so on. According to the findings of our study, 57% of the approaches support this. Since FTs can be large, depending on the system size and its complexity as well as the individual component failure behavior, it is vital for an FTA platform to make it easier for the user to navigate through the system's main failure paths in order to better help in defining how they might be easily mitigated. This can be accomplished either graphically, through actions such as path reduction, as well as textually through deriving minimal sets of events required for a system to fail (minimal cut-sets). Aside from that, alternative approaches may be completely platform-specific and dependent on the failure behavior modeling approach and FT generation mechanisms.

- *Perform automated quantitative FT analysis*: This feature determines whether the proposed approach allows quantitative analysis, mainly the top failure event probability estimation. Offering such support could potentially aid in quantifying the risk and determining how to manage it. However, this is regarded as optional in the FTA mechanism due to the lack of a standard means of determining individual component failure rates, since basic event failure rates encompass not just hardware failures but also software, human, and environmental factors. Only 35.7% of the approaches evaluated support such a feature, namely [169, 172, 174, 180]. Our proposed approach not only computes the intermediate and top event probabilities from the basic events, but it also recognizes underdeveloped branches and injected failures. It is also worth noting that, during the FT generation process, a file containing the probability information is generated, which allows you to still update the component failure rates and re-run the analysis,

in which the new values are picked up by the tool without having to re-generate the FT again.

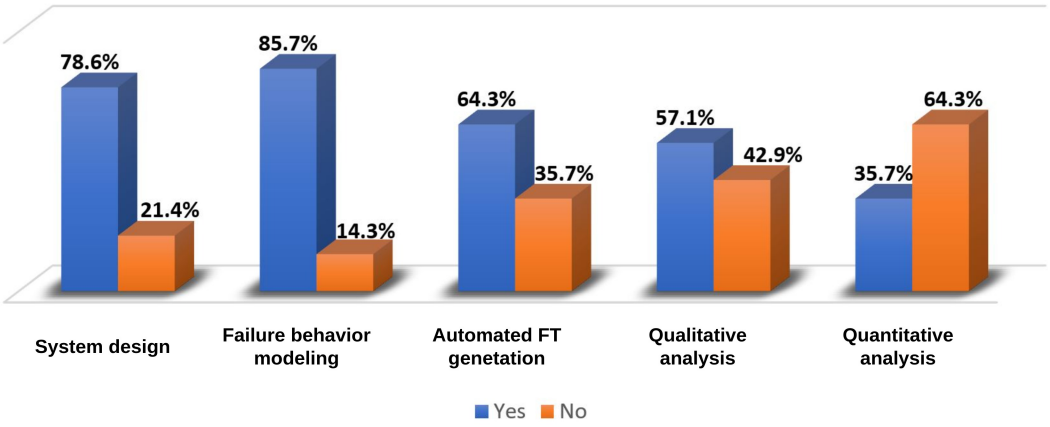


Figure 7.9: Feature support performances

As it can be seen from the table, our approach supports all five features; among the others, only [172] and [180] support all of them. However, as previously discussed, these differ significantly from our proposed approach in terms of capacity and efficiency. In summary, the proposed approach is completely unique in terms of advancing the state-of-the-art through various novel mechanisms such as support for system design modeling, failure behavior modeling, automated FT generation, automated qualitative FT analysis, and automated quantitative FT analysis.

7.4.2 PMS system modeling (RQ2)

RQ2: Does the system-level modeling infrastructure address all the aspects of modeling a multi-layered IoT system suitable for safety analysis?

Patient monitoring system design

In order to better answer this question, we showcase the capability of our proposed modeling environment employing the case study presented in Section 7.3.2. As can be seen in Figure 7.8, the Patient Monitoring System (PMS), uses a set of sensors to collect sick patient data and send them to a remote server. The system can display the data on the monitor as well as send an alarming signal when something gets wrong. Figure 7.10 represents the internal physical architecture of the proposed system. For the sake of simplicity and to facilitate the analysis process able to produce presentable results, we have considered the following changes to the architecture presented in 7.8. Firstly, we designed a PMS that only monitors a single patient. Secondly, we introduced a remote server component that acts as a bridge by hosting the service that saves the received data and exposes them to other third parties services that might need them. Thirdly, we replaced the doctor's phone subsystem with an alarming system component that receives data from the PMS software on the monitor side. Finally, we added a "Human" component to reflect the role of a doctor in the overall system functionality.

As shown in Figure 7.10, a *"SensingUnit"* composite component consisting of five sensors namely ECG, EEG, SPO2, pressure, and temperature sensors. All the sensors are placed on a patient's body to collect the patient's health parameters. They are directly sent to the controller, which aggregates all of that information and sends it to a gateway (in this case a transceiver). The gateway processes the data and forwards them wirelessly to a remote server. The server hosts the software services that save the data as well as exposes them to other authenticated parties in need. On the other hand, the monitoring

software deployed on the computer accesses such information and sends them to a displaying screen. When something goes wrong, for instance, in terms of sensor reading values that exceed or are below a certain threshold, the monitoring software can decide to raise an alarm in order to alert the doctor about the unusual condition of the patient. In this case, a doctor checks on the displayed data and decides to act accordingly by either shutting off the alarm, changing the configuration in the systems, or fixing some issues that might be related to the sensors.

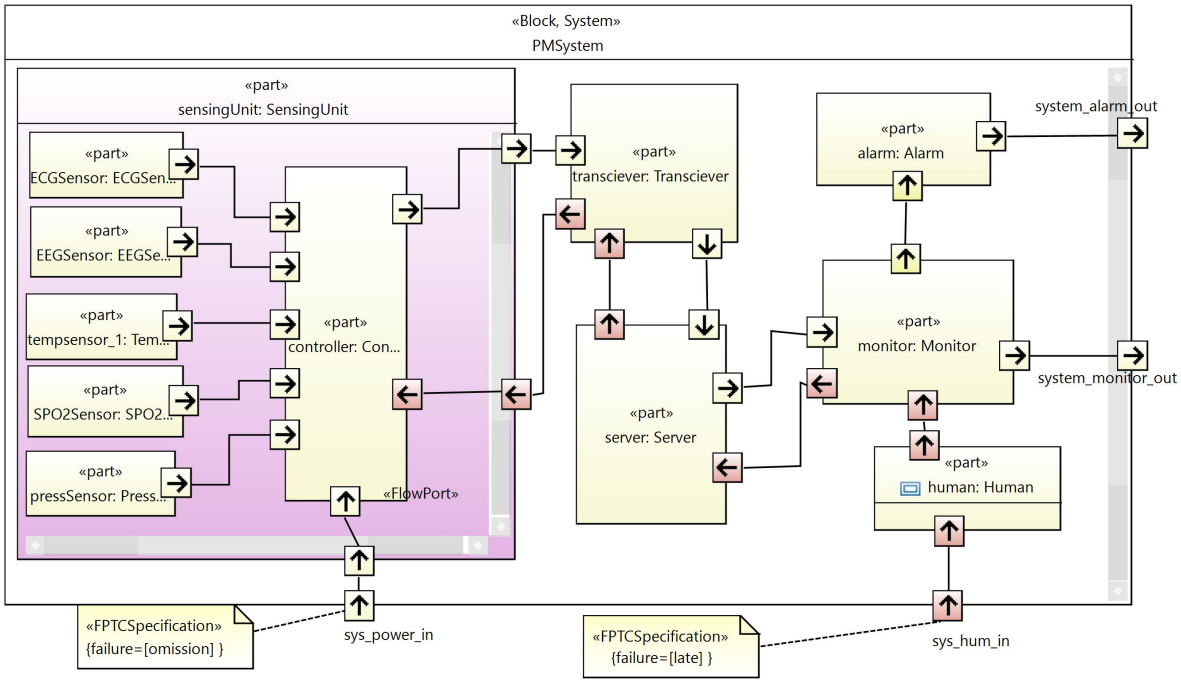


Figure 7.10: Patient monitoring system

PMS model includes failure behavior data

To facilitate the modeling of the system failure behavior data needed for the safety analysis data, the infrastructure allows annotating the failure behavior rules as well as the failure rates on each of the low-level simple components and this information is fully part of the model itself. As we all know, external influences can cause a system to fail. Through two system-level input ports, the presented architecture allows for simulation effects in which an external failure introduced in the system would affect the overall system functionality. For instance, the *sys_power_in* port, used to model the power source outlet, was been injected with an "omission" failure which basically models the event in case there is a power outage. On the other hand, the *sys_hum_in* port is used for modeling the external influence of the doctor. In our case, a "late" failure was used to simulate an event in which a doctor reacted late due to some external factors. Finally, the system contains two output ports, namely system_monitor_out for modeling the output port from the monitor, and the system_alarm_out to model the output of the alarm system. According to the direction of the ports at the system level as well as the type of failures that are able to propagate to them, different FTs are generated accordingly.

7.4.3 System failure behavior (RQ3)

RQ3: How well do the proposed FLA rules efficiently reflect the system failure behavior leading to the system top failure events?

As was previously anticipated, the above system is subjected to different kinds of failures either being internally generated from the system or coming from the surrounding environment. As we described in the previous section, it is possible to model the individual component's failure behavior

that later gets assessed in determining the failure behavior of a sub-system or an entire system. Note that we are not focusing on software-level functional behavior but on physical failure behavior which can be even understood by non-professional users. In order to understand the need for the conducted analysis, let's first discuss different top failure scenarios that we have taken into account in defining individual component rules.

- **The alarm sub-system malfunctions by sending out a false signal**: In normal settings, this can occur when the alarm component receives a wrong alarm notification. This is usually caused by the monitor software making a decision based on incorrect data from one of its input ports. The alarming system, on the other hand, can send false signals due to its internal failure for a variety of reasons such as poor internal configuration or simply aging.
- **The alarm subsystem has completely stopped working:** It is possible that the alarm system no longer works completely. This can be caused by several reasons; for example, the alarm system being physically disconnected, or internal failure which causes a total black-out.
- **The monitor is displaying incorrect data:** As it is obvious, the main cause of this could be due to incorrect data being sent to the monitor. However, other factors, such as a faulty monitor, losing connection to the internet make it display the last received data, and so on. It should be noted that these are only generalized assumptions; the extensive individual study, as well as their corresponding failure rules, is shown in Table 7.4.
- **The monitor completely fails to display data on the screen:** This can occur due to internal and/or external monitor issues such as the monitor not being physically connected to the system power source, being unable to connect to the server, internal monitor malfunction due to aging, and so on. On the other hand, this could be caused by the monitor is properly connected but the server not receiving any data from the sensing unit.

The next step is to derive internal failure rules as well as the propagation rules for the basic components. For instance, for each sensor, two rules were defined to model two different scenarios in which a sensor can fail. A sensor can fail internally leading to a complete omission in providing the data to the output port, thus an "omission" failure will be propagated to the output port of the sensor. On another hand, a sensor can start to fail but not completely due to age. This may result in providing incorrect data to the output; this can also be caused by sensor components that are not properly placed in the patient's body. In this case, we consider that the value sent to the output port is of "valueCoarse" type. Hence the two different types of failure can be propagated to the same output port in a different scenario, and they will be represented as indicated in Expression 8.1 and 8.2 respectively. We considered only the two failure conditions to apply for all of the sensors. As it can be seen from Table 7.4, a detailed set of failure behavior rules and their description are represented.

$$FLA : (*) \rightarrow ecgsens\_out.\textbf{omission} \tag{7.6}$$

$$FLA : (*) \rightarrow ecgsens\_out.\textbf{valueCoarse} \tag{7.7}$$

In CHESSIoT, to facilitate the quantitative analysis, the failure rates of the component internal failure events as well as the injected failures events have to be set separately. As we did not have the exact failure rates of the basic components, we considered the arbitrary failure rates of any component to be practically small in a range of 10^{-8} to the 10^{-7}. Figure 7.11 depicts the interfaces in which the internal failure and their description are set. Having the event description is practically good in order to facilitate the readability of the FT, but it is not mandatory to have it for performing qualitative analysis. When no data is provided to any of the rows, the default values are used. For instance, an unset basic event probability is assigned with a value of zero in the FT, while the unset basic event description will still follow the conventional naming of "<failure type> at <port name> in <component name>".

Component	Rules	Description
ECG, EEG, Temp, SPO2 and Pressure sensor	(1) FLA:(*) → ecgsens_out.**omission**;	Sensor fails internally which make it unable to read and push any at output port
	(2) FLA:(*) → ecgsens_out.**valueCoarse**;	Sensors begin to fail as they age and provide incorrect data to the output; this can also be caused by sensor components that are not properly mounted to the patient body.
	Rules (1) and (2) apply to other sensors	Same as other sensors
Controller	(3) FLA:ecg_cont_in.**noFailure**,eeg_cont_in.**noFailure** → cont_trans_out.**omission**;	Controller fails completely omitting to send the data
	(4) FLA:mon_power_in.**omission**, trans_cont_in.**omission** → cont_trans_out.**omission**;	The controller fails to function due to a power outage at its input power port, no backup solution is available *(trans_cont_in port)*
	(5) FLA:ecg_cont_in.**omission**, eeg_cont_in.**omission**, press_cont_in.**omission**, spo_cont_in.**omission**, temp_cont_in.**omission** → cont_trans_out.**omission**;	All of the sensors simultaneously stop sending data, preventing the controller from sending any data to the server
	(6) FLA:ecg_cont_in.**valueCoarse** → cont_trans_out.**valueCoarse**;	The controller receives inaccurate data from the ECG sensor and sends it to its output port
	(7) FLA:eeg_cont_in.**valueCoarse** → cont_trans_out.**valueCoarse**;	Controller receives inaccurate data from the EEG sensor and sends it to its output port
	(8) FLA:press_cont_in.**valueCoarse** → cont_trans_out.**valueCoarse**;	Controller receives inaccurate data from the blood pressure sensor and sends it to its output port
	(9) FLA:spo_cont_in.**valueCoarse** → cont_trans_out.**valueCoarse**;	Controller receives inaccurate data from the SPO2 sensor and sends it to its output port
	(10) FLA:temp_cont_in.**valueCoarse** → cont_trans_out.**valueCoarse**;	Controller receives inaccurate data from the temperature sensor and sends it to its output port
	(11) FLA:trans_cont_in.**valueSubtle** → cont_trans_out.**valueCoarse**;	The controller receives an undetected error at its from-system port, which impede sensor data transmission.
	(12) FLA:trans_cont_in.**valueSubtle** → cont_trans_out.**omission**;	The controller receives an undetected error at its from-system port halting the data transmission process.
Transceiver	(13) FLA:trans_in_fr_unit.**valueCoarse** → trans_out.**valueCoarse**;	The transceiver received wrong data and transmit to its output port
	(14) FLA:trans_in_fr_unit.**noFailure**,trans_in_f_serv.**noFailure** → trans_out.**omission**;	The transceiver fails internally causing the halt of data transmission process
	(15) FLA:trans_in_fr_unit.**omission** → trans_out.**omission**;	The transceiver receive no data and fails to transmit any data to its output port
	(16) FLA:trans_in_f_serv.**valueSubtle** → trans_o_2_unit.**valueSubtle**;	The transceiver receive an undetected errors at its server port and forwards it back to the sensing unit
Server	(17) FLA:server_in.**noFailure** → server_out.**omission**;	The server fails, bringing the transmission process to a halt
	(18) FLA:server_in.**valueCoarse** → server_out.**valueCoarse**;	The server routes the incorrect data received at the input port to the output port
	(19) FLA:server_in.**omission** → server_out.**omission**;	The server receives no data from its input port, and this error is forwarded to its output port
	(20) FLA:serv_in_f_mon.**valueSubtle** → serv_2_trans_out.**valueSubtle**;	The server sends an undetected error from the monitor back to the transceiver's port
Monitor	(21) FLA:monitor_in.**noFailure** → monitor_out.**omission**,mon_alarm_out.**omission**;	The monitor fails internally omitting to display the data on the screen as well as not communicating to the alarm component
	(22) FLA:monitor_in.**omission** → monitor_out.**omission**, mon_alarm_out.**omission**;	The monitor receiving no data from the server omitting to display the data as well as not sending any communicating signal to the alarm component
	(23) FLA:monitor_in.**valueCoarse** → monitor_out.**valueCoarse**, mon_alarm_out.**commission**;	The monitor receives inaccurate data and displays it on the screen, potentially sending a unexpected notification to the alarm component *(Commission)*
	(24) FLA:hum_mon_in.**valueSubtle** → mon_2_serv_o.**valueSubtle**, mon_alarm_out.**commission**, monitor_out.**noFailure**;	The monitor receive an unpredicted error from the human nurse component, the failure propagates in the system in various ways with no direct effect on the data displayed on the screen before (Refer to Rules 30 and 31 for possible causes)
	(25) FLA:hum_mon_in.**omission** → mon_2_serv_o.**valueSubtle**;	The monitor receives no engagement from the nurse intended to resolve the issue in the system, the cause of which we do not know. As a result, such a failure will go unnoticed by the system. (Refer to Rules 11 and 12 for possible effects)
Alarm	(26) FLA:mon_alrm_in.**commission** → alarm_out.**commission**;	The alarm component received an inaccurate notification and immediately rings because it lacks any type of logical reasoning on the signal receiving other than ringing.
	(27) FLA:mon_alrm_in.**noFailure** → alarm_out.**commission**;	The alarm starts failing due to internal failure which can make it malfunction by giving false alerts
	(28) FLA:mon_alrm_in.**noFailure** → alarm_out.**omission**;	The alarm component fails completely which makes it unable to make any alert
	(29) FLA:mon_alrm_in.**omission** → alarm_out.**noFailure**;	The alarm receive no data but that won't affect the internal functionality of the alarm component
Human nurse	(30) FLA:human_in.**late** → human_out.**valueSubtle**;	The human nurse reacts very slowly in the event of a system failure, which may or may not affect the system in some way, which is why a "valueSubtle" is considered.
	(31) FLA:human_in.**noFailure** → human_out.**omission**;	The absence of the doctor results in an omission at the output port

Table 7.4: PMS failure behavior table

As proved in the preceding discussion, our proposed approach is capable of satisfying all potential failure behaviors prescribed by the safety expert. As shown in Figure 7.10, our approach is capable of modeling the backward failure propagation pattern. For instance, a server failure will affect the monitor's behavior, preventing data from being displayed on the screen. On the other hand, an erroneous command sent by the doctor's absence (for example, to fix an unmounted sensor) may eventually propagate back to the sensing unit, causing a wrong value error to be transmitted at the controller output port (valueCoarse failure) or possibly suspending the data transmission process (omission failure). Well, it is also worth noting that the ability to integrate all of their component failure rules as well as their failure rates in the same model has the potential to boost model consistency as well as transparency in the modeling process.

Probability Registration Form

Name	Probability	Failure description
ALARM: FLA:mon_alrm_in.noFailure->alarm_out.commission;	0.00000004	Alarm system start to fail increasingly due to age
ALARM: FLA:mon_alrm_in.noFailure->alarm_out.omission;	0.00000005	Alarm system fails completery or broken
HUMAN: FLA:human_in.noFailure->human_out.omission;	0.00000006	Human not present at all
PRESSSENSOR: FLA:(*)->psens_out.omission;	0.00000007	Pressure sensor fails completely or broken
PRESSSENSOR: FLA:(*)->psens_out.valueCoarse;	0.00000008	Pressure sensor starts to fails by age and provide wrong readings
SPO2SENSOR: FLA:(*)->spo2sens_out.omission;	0.00000009	SPO2Sensor fails completely or broken
SPO2SENSOR: FLA:(*)->spo2sens_out.valueCoarse;	0.000000011	SPO2Sensor starts failing due to age and provides worng readings
CONTROLLER: FLA:ecg_cont_in.noFailure,eeg_cont_in.noFailure->cont_trans_out.omission;	0.000000012	Controller fails completely internally due to unknown issues
TEMPSENSOR_1: FLA:(*)->tempsens_out.omission;	0.000000013	Temperature sensor fails completely or broken
TEMPSENSOR_1: FLA:(*)->tempsens_out.valueCoarse;	0.000000014	Temperature sensor starts failing due to age and provides worng readings
EEGSENSOR: FLA:(*)->eegsens_out.omission;	0.000000015	EEG sensor fails completely or broken
EEGSENSOR: FLA:(*)->eegsens_out.valueCoarse;	0.000000016	EEG sensor starts failing due to age and provides worng readings
ECGSENSOR: FLA:(*)->ecgsens_out.omission;	0.000000017	ECG sensor fails completely or broken
ECGSENSOR: FLA:(*)->ecgsens_out.valueCoarse;	0.000000018	ECG Sensor starts failing due to age and provides worng readings

SUBMIT CANCEL

Figure 7.11: PMS components failures rates set

7.4.4 PMS Fault tree analysis (RQ4)

RQ4: Does the proposed FT qualitative and quantitative analysis improve the existing FT analysis techniques?

The FT analysis begins after the CHESS-FLA transformation, as described in Section 2. The FT generation process is performed prior to running the FT analysis, in which each of the top events described in Section 4 results in its own FT. For instance, FT leading to an "omission" failure at the system monitor out port is generated to show the entire failure contribution leading to that top event. Other FT representing the remaining 3 top events are generated as well. At this stage, the generated FTs are very large as they contain every detail related to failure propagation and transformation from component to component, making it tricky to read. Therefore, FT analysis can then be launched to automatically perform both qualitative and quantitative analysis on the model.

Figure 7.12 shows the analyzed FT of the event "the monitor fails to display data completely on the screen". The presented FT showcases only the important events and logical gate combinations. It can be clearly anticipated that the analyzed FT makes it easier to identify and trace any failure source events in their contribution to the top failure event. For instance, we can easily grasp that the monitor would display no data on the screen completely when any of the following events occur:

- Internal failure in the monitor (10^{-8} probability)
- Server is down ($2x10^{-8}$ probability)
- The transceiver (gateway) fails completely to transmit the data ($3x10^{-8}$ probability)
- A combination of events (low-left AND gate) in which there is a problem with the sensing unit power source and there is no person to fix that at the moment. ($5x10^{-15}$ probability)
- The controller of the sensing unit fails completely which halts the transmission process ($1.2x10^{-8}$ probability)
- An event in which all the sensor does not send the data at all (This is more unlikely but possible, that's why we have a lower-middle and gate combination with $2.088x10^{-38}$ probability).
- An unknown human error occurred from the monitor side ($2x10^{-7}$ probability)

When the event of the monitor not displaying any data occurs, medical personnel can rely on such a narrow series of events to determine the cause of the event. Furthermore, medical personnel can use the probability associated with each basic event in the list to quickly locate the source of failure, moving from the most probable basic event (highest probability) to the least probable event (lowest

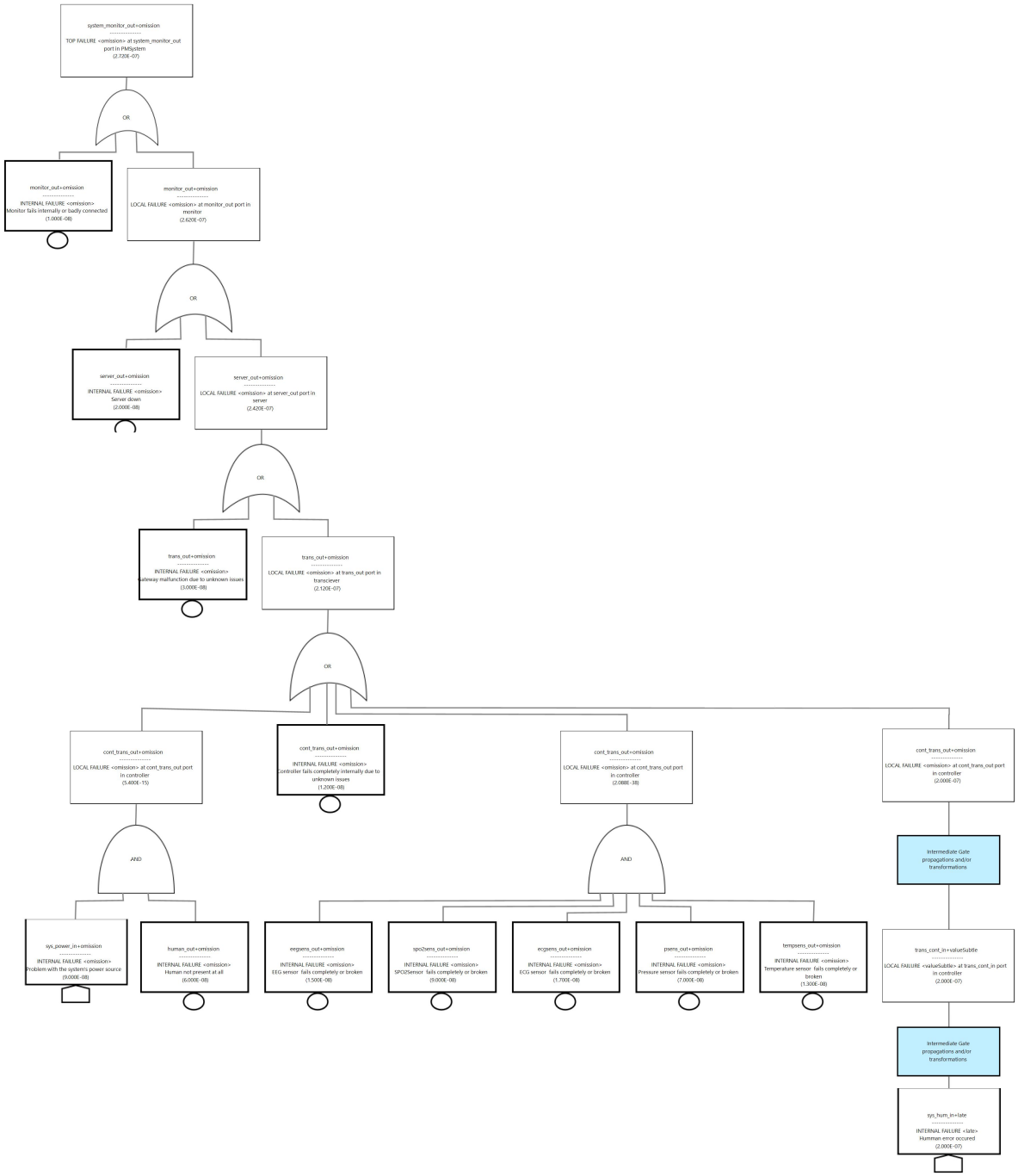


Figure 7.12: PMS monitor screen shown no data

probability). The overall probability of this system-level failure event occurring is calculated to be 2.72x10^{-}7, which is practically small, however, this value is calculated automatically and is solely dependent on the arbitrary basic event failure rates as well as their logical combination analysis, as shown in the FT.

Other analyzed FT on the event in which "monitor displaying incorrect data" and "PMS alarm sub-system alert false signal" is shown in figure 7.13 and 7.14 respectively. As it can be seen from figure 7.13, the event in which the monitor will display incorrect data can be caused by any of the sensors (OR gate), as well as an unforeseen human error that transforms throughout the system and hinders the data that are being transmitted. The overall probability in which such an event can occur is calculated to be about 3.39x10^{-}7 which is higher that the event in which the monitor can stop working a all.

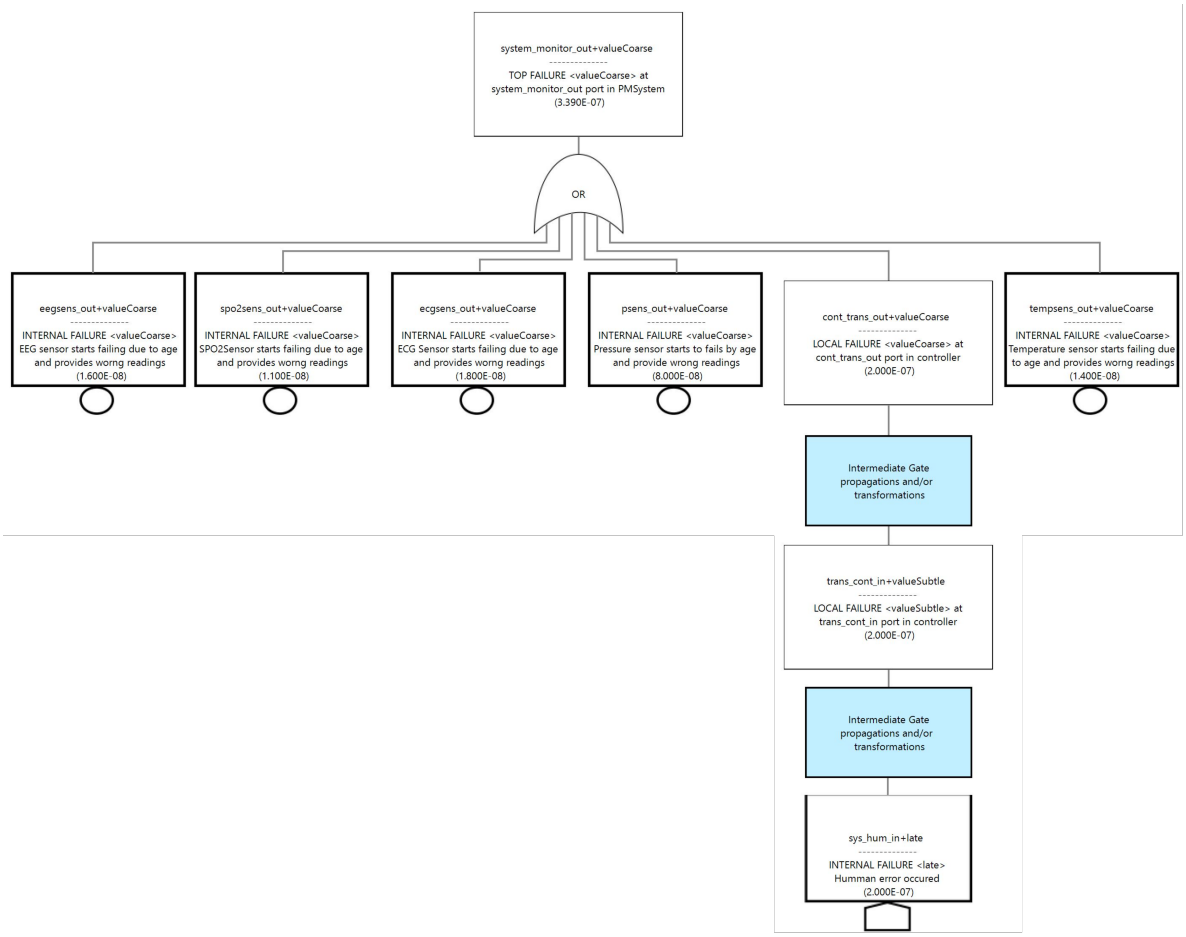


Figure 7.13: The monitor fails to display data completely on the screen

On the other hand, as shown in Figure 7.14, the same events that cause the monitor to display incorrect information can also cause the monitor to send a false signal via a failure transformation, resulting in a false alter event in the alarm system. It is also worth noting that an event like the alarm sub-system failing with a probability of $4x10^{-}8$ would also contribute to that cause. the overall probability of such an event to occur is projected to be around $5.79x10^{-}7$ which is much higher that the previous two top events. Furthermore, while the "Human error" basic event appears twice in the tree, such failure passes through different channels and eventually transforms into other types throughout the system. This is practically important to understand which component of a system's error would change its nature, potentially causing a lot more damage than expected. Finally, It is worth noting that this FT does not include the top event in which the alarm sub-system stopped working completely. An FT reflecting such an event was generated and analyzed separately.

Typically, safety engineers will collaborate with system engineers to keep the safety model up to date during the development process. Maintaining coherence between system architecture and the safety model can be difficult as the model gets larger and more complicated. Having a framework that can integrate modeling and analysis processes from a single place would potentially improve consistency, and transparency, and minimize analysis time. Overall, the proposed analysis approach is capable of achieving that by the means of automated qualitative and quantitative calculus.

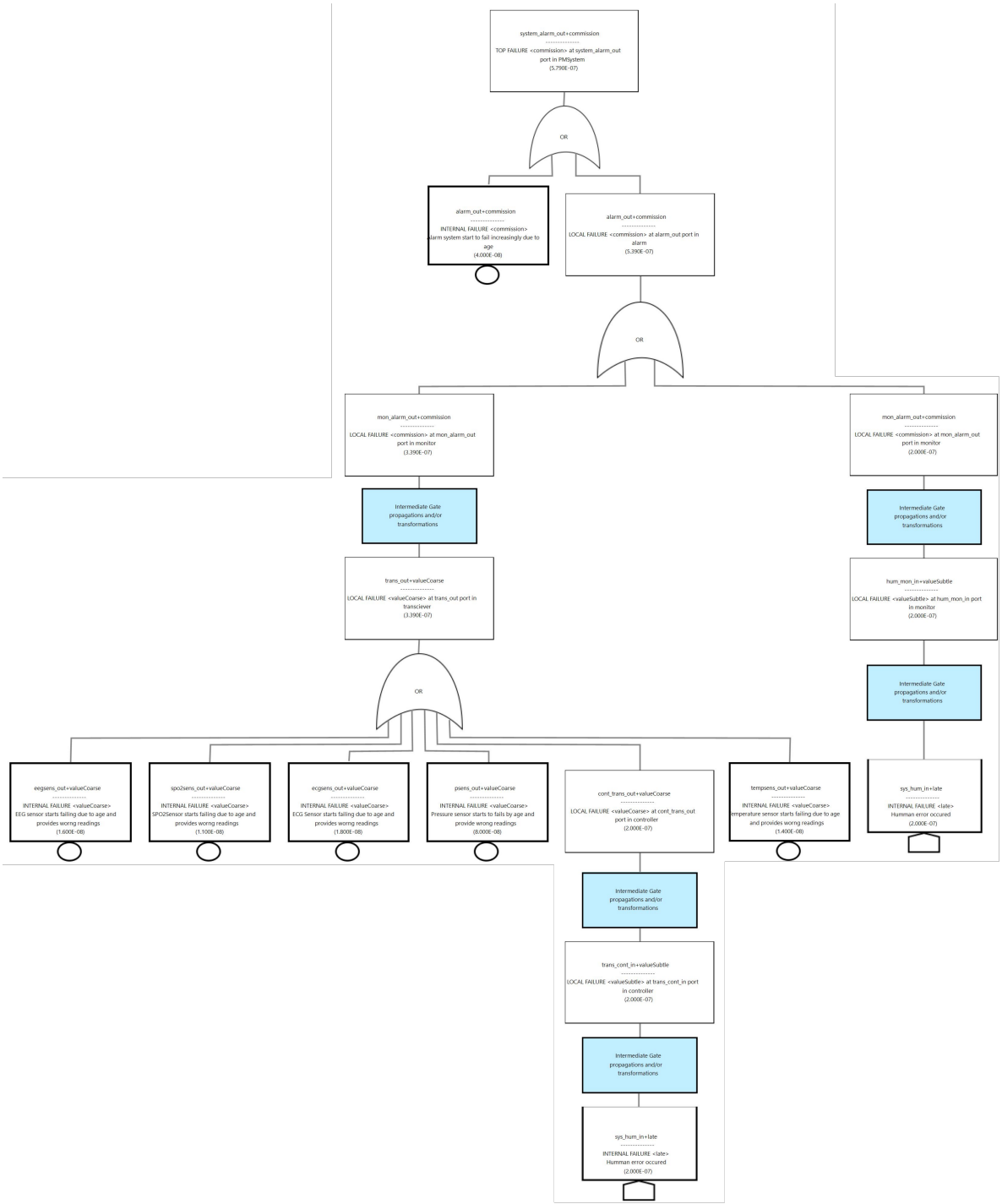


Figure 7.14: Alarm sub-system alert false signal

7.5 Conclusion and future work

Automated safety analysis is critical for increasing transparency and reducing the time required for manual analysis. However, when the system becomes too large and complex, it is very difficult to maintain the coherence between the safety analysis model with the corresponding system architecture. In addition, the architecture usually has to be reworked so many times, which can hinder the consistency of the process. This chapter presented CHESSIoT, a novel approach for developing and performing safety analysis on safety-critical IoT systems. The proposed method combines rigorous automated analysis procedures with annotated failure behavior on components and associated failure

rates to generate fault trees. The supporting tool can perform both qualitative and quantitative analysis on generated FTs. We presented an evaluation mechanism compared to existing techniques to showcase its novelty, and the results were very promising. The approach improves model composability and reuses while reducing the time required to perform the analysis. In the future, we want to integrate time-based failure logic analysis into our analysis approach if a given failure would only affect the system for a short period. Furthermore, we plan to investigate the feasibility of combining with Markov chain analysis land/or dynamic fault tree analysis using Monte Carlo simulation when performing the quantitative FT analysis.

Chapter 8

Supporting for development and deployment of IoT systems with CHESSIoT

Engineering tools that can handle this complexity while also reducing the development time will undoubtedly have a massive impact on the market. As a contribution toward answering the fifth research problem (RP5), this chapter present the CHESSIoT development and deployment approach as a part of the whole CHESSIoT-supported engineering process. In addition to the already presented safety analysis support in Chapter 7, CHESSIoT integrates high-level visual design languages, software development, and deployment mechanisms. Additionally, the tool offers means to define run-time service provisioning modules through deployment agents, which are then used to configure remotely deployed services. To showcase the effectiveness of our proposed approach, as well as the capability of the supporting tool, a Home Automation System (HAS) example was developed, covering the modeling, development, safety analysis, and deployment views.

This chapter is organized as follows: Section 8.1 presents a brief introduction to the topic. Section 8.2 presents CHESSIoT software development methodology and code generation process. Section 8.3 present the deployment methodology as well as the run-time service provisioning modeling approach. Section 8.4 presents a Home Automation System running example showcasing the capabilities of the proposed supporting tool. Finally, Section 8.5 concludes the chapter.

8.1 Introduction

Model-driven development and deployment of IoT systems is a challenging task that requires comprehensive and flexible support. With the increasing complexity of IoT systems, model-driven approaches have become a popular solution to enable efficient development and deployment [235]. These approaches use models to represent the structure and behavior of IoT systems and automate the development and deployment process. During the modeling phase, developers create models that capture the requirements, architecture, and behavior of the IoT system. These models are then used to perform analysis, such as simulation and verification, to ensure that the system meets the desired specifications [68]. Furthermore, developers use the models to generate code that implements the system's functionality. The code is then tested to ensure that it works as intended, and the system is finally deployed to the target environment.

However, MDE in IoT systems poses unique challenges, such as heterogeneity of devices and communication protocols, dynamic and unpredictable environments, and the need for real-time response [86]. To address these challenges, there is a growing need for tools and frameworks that support MDE for IoT systems. One promising approach is to use domain-specific languages (DSLs) that are tailored to the specific requirements of IoT systems [71]. DSLs provide a higher level of abstraction than general-purpose modeling languages, enabling developers to create models that are closer to the problem domain. Moreover, DSLs can be used to generate platform-specific code, making it easier to integrate with IoT platforms and devices [37].

In addition to that, efficient and scalable deployment of IoT systems is also critical for the success of IoT applications. Deployment of IoT systems involves a wide range of tasks, including configuration, provisioning, monitoring, and maintenance of devices and services [17]. To support efficient and scalable deployment, there is a need for tools and frameworks that automate these tasks, and enable seamless integration with IoT platforms and infrastructure. In particular, the deployment of IoT systems involves several challenges, such as managing the heterogeneity of devices and communication protocols, dealing with resource constraints, and ensuring the reliability and security of the system [166].

To address these challenges, there is a need for tools and platforms that support the model-driven deployment of IoT systems. These tools should provide a high-level abstraction of the system by enabling developers to focus on the system's deployment requirements. They should also support the automatic generation of deployment artifacts, such as configuration files, scripts, and Docker containers, based on the models of the system [223].

As introduced in Chapter 6, CHESSIoT integrates the modeling, software development, analysis, and deployment for engineering multi-layered IoT systems. In this chapter, we demonstrate in great detail CHESSIoT's development and deployment approach in addressing some of the potential challenges presented above. Concerning software development, CHESSIoT provides the user with means to define a functional model which contains the system's key software components, sub-functions, and interrelationships. Furthermore, a behavior model is entitled to each of the system's main sub-function in form of a state machine in which aspects such as events, actions, and guards are associated with states and their transitions to realize the desired behavioral goal. When the model is complete, a *CHESSIoT2ThingML* model transformation is launched to generate a series of fully functional ThingML source models which is then used to generate platform-specific code ready to be deployed on low-level IoT devices.

In addition to that, The CHESSIoT includes a deployment environment that aims at supporting the users with decomposing the IoT system deployment plan as well as managing deployed node services across all layers. As a matter of fact, IoT services are no different from other domains their deployment should also follow a multi-tenant approach in which a single service instance should be running on the host servers, and that single instance serves each subscribing customer or cloud tenant [236]. Runtime service provisioning refers to the process of allocating and configuring resources, such as computing power, storage, and network connectivity, at the time a program or application is run [237].

In runtime service provisioning, resources are dynamically allocated based on the needs of the

program or application at any given time. Mastering a variety of deployment languages can be tough and not to mention the tight coupling between the script being used and the environment they are intended to apply to. But always the question remains *"Should really the deployment plan change every time the target environment changes?"*

To potentially address such a problem, CHESSIoT offers a model-driven runtime service provisioning environment that allows the automatic configuration of software services based on a predefined model. The CHESSIoT provisioning abstraction is defined using deployment scripts referred to as agents. These agents are annotated to the deployment nodes in the model, to provide run-time monitoring of the deployed services. A textual language for defining deployment rules is used to describe the agents' behavior. These rules will eventually be transformed into Ansible deployment playbook scripts [224] that can be run manually on a remote machine. To evaluate the proposed approach as well as showcase the capability of the tool, we have demonstrated a Home Automation System (HAS) use case in which we used the tool for modeling, developing, conducting safety analysis, and supporting its deployment.

8.2 Software modeling and development approach

Software design and development is the process of creating and implementing software systems and applications. The software design phase involves creating high-level conceptual models of the system, identifying key components and interfaces, and defining the overall software architecture. CHESSIoT follows a multi-view design paradigm, the software design is done under the *"Component View"* to enable users to design functional and behavioral aspects of the software's edge layer.

In CHESSIoT, the user benefits from a dedicated IoT-specific graphical modeling environment consisting of specific diagrams and palettes that are hidden or shown based on the current design step via the *"IoT sub-view"*. Having such a sub-view enables CHESSIoT to be a completely decoupled environment from CHESS, which is relevant throughout the whole design process. Figure 8.1 shows the support of the complete design and development phase.

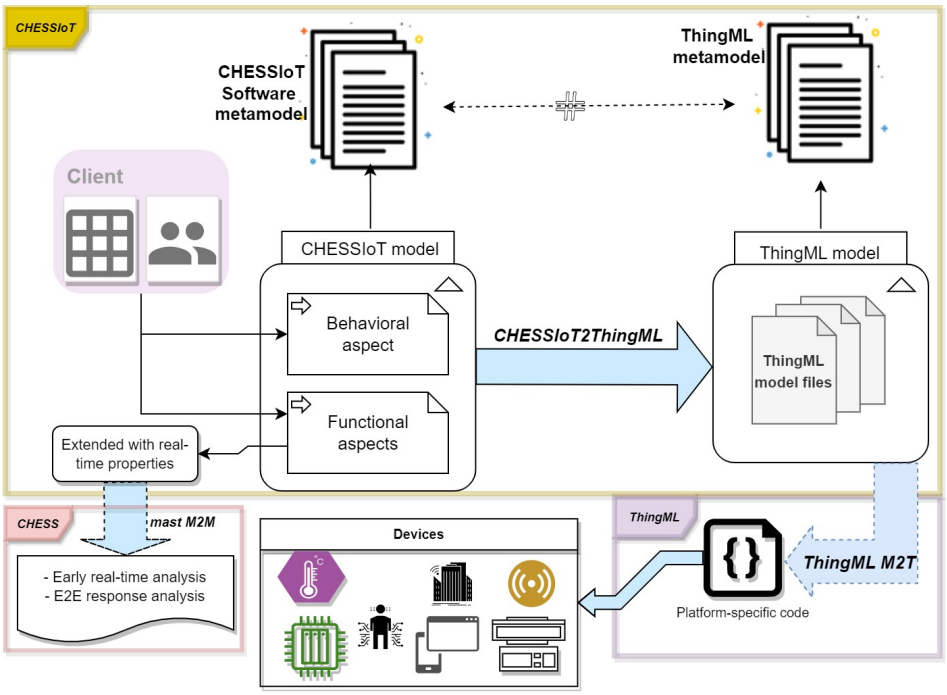


Figure 8.1: Software development process

The software development process initiates with the user creating functional and behavioral models that conform to the software metamodel shown in Fig. 6.3. Once the model reaches its final form,

a transformation called *CHESSIoT2ThingML* is executed to generate ThingML model files. These files can then be utilized within the ThingML environment to generate platform-specific code that is ready for deployment on devices. Alternatively, the functional model can be expanded to incorporate real-time properties, enabling real-time analysis to be conducted. This section does not delve into the runtime analysis, as it has already been covered in the work presented in [42]. However, it does provide specific details regarding the design strategy and code generation approach supported by the CHESSIoT tool.

8.2.1 Specification of CHESSIoT software models

In CHESSIoT, the modeling of software components is closely intertwined with the definition of their behaviors, ultimately resulting in the generation of platform-specific code. The software design approach encompasses both the functional design and behavioral design aspects of the system. The functional design entails a systematic definition of the primary software components, their subcomponents, and their interconnections. This process employs *component structure diagrams* that adhere to a component-to-connector design methodology [37]. During this stage, communication between components is exclusively facilitated through dedicated ports utilizing payload entities. For designing systems that involve wireless communication, such as MQTT-based systems, a special port with an MQTT stereotype is utilized. This MQTT port captures all MQTT-related information, including the broker URL, client type, and topic.

When modeling the internal behavior of a component, internal class diagrams are used, where only specific palette elements are displayed to the user at this stage. Each main sub-function of the system is assigned its own state machine, which encompasses events, actions, and guards associated with states and transitions to achieve the desired behavioral objective. Figures 8.2 and 8.3 presents the high-level mechanisms that are followed during the definition of the component's state machine as well as the event, action semantics definition process. For instance, according to Figure 8.2, an event can be categorized as either an *Internal event* or a *Conditional event*. The event references the payload values found at the corresponding port for verification. When an event is triggered, it initiates an action, which can be either a *SetAction* or a *GenericAction*, depending on the context.

A *SetAction* always sends a *Payload* through a given port, while a generic action would mostly be customarily implemented. A guard which enables a state transition based on the *OnExit* action status or can be customarily implemented. Such a condition is added to the code unchanged during the code generation. A combination of such events and actions is referenced throughout different states and transitions accordingly. Figure 8.2 depicts the basic activities that need to be fulfilled from one state to the other.

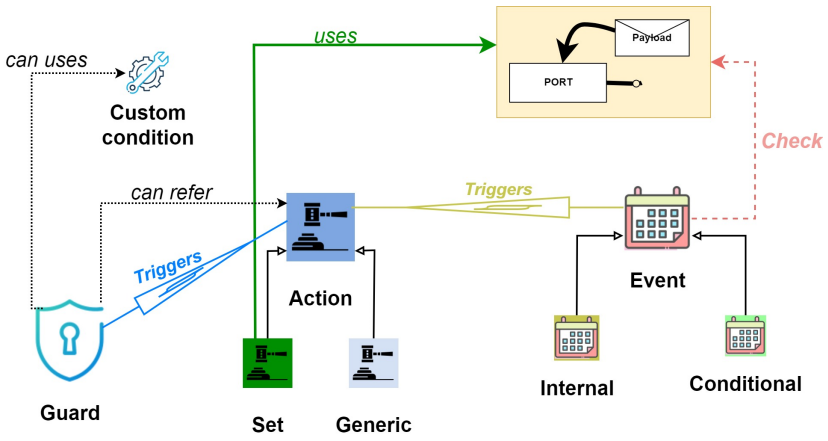


Figure 8.2: Behavior event & action relationship

Figure 8.3 depicts the general idea behind the basic state-based behavior process supported by our approach. The diagram illustrates an example of two states, i.e., *S1* and *S2*, and the requirements and

activities that must be met to perform the transitions among them. Moreover, when leaving a state, zero or more *OnExit* actions might be fulfilled. This is defined within a state and will be checked using the guard expression.

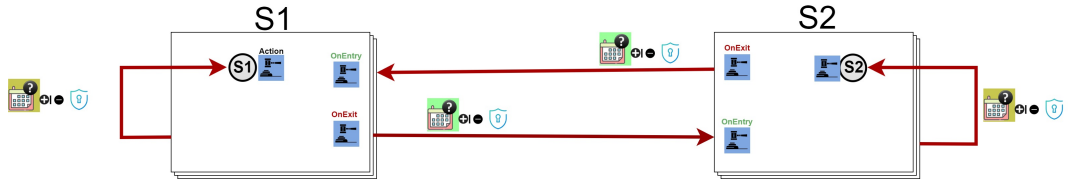


Figure 8.3: State and state transition

Conditional events must be attached to state transitions when moving from one state to another. Furthermore, zero or more *onEntry* actions may be performed when entering a state. An internal event is used within a state to trigger actions of interest. It is important to note that at this point, a conditional event always inspects the payload state at the ports to initiate a state change.

8.2.2 The CHESSIoT to ThingML transformation

The CHESSIoT2ThingML transformation process is done through model text transformations written in Accelleo[1]. Acceleo is an open template-based source code generation technology developed in the context of the Eclipse Foundation. In this section, we will delve into the details of ThingML and the steps involved in generating ThingML models from CHESSIoT models.

What is ThingML? ThingML is a model-driven development and code generation framework which combines a textual modeling language and a set of compilers targeting a range of different platforms (from micro-controllers to servers) to generate ready-to-use platform-specific code. ThingML code generators support the generation of three main languages (C/C++, Java, and JavaScript) and several libraries and open platforms (Arduino, Raspberry Pi, Intel Edison, Linux, and so on) [235].

The ThingML approach targets distributed reactive systems and is especially beneficial for applications that include heterogeneous platforms and heterogeneous communication channels. In ThingML, a Thing is an implementation unit, also referred to as a component or process. It can define properties, functions, messages, ports, and a set of state machines [37]. All the properties are local variables and can be accessed globally from within a thing through a function or a state machine. Same as properties, the functions are also local to a thing, and they can be used from anywhere in a thing. Same as CHESSIoT, things can be interfaced with other things through the ports employing sending and receiving a set of messages.

The ThingML language relies on two key structures: *Thing*, which represents software components, and *Configurations*, which describe their interconnection [37]. During the CHESSIoT2ThingML transformation, the generation of those two main sets of code is done separately, as described in the next sections. Over the years, the ThingML approach has continuously evolved and applied to cases in different domains, including commercial e-health applications such as fall detection systems called Safe@Home [235], Micro-aerial vehicle platform as well as the Arduino Yùn IoT-based projects [37].

CHESSIoT to ThingML generator: The CHESSIoT component's semantics differ from the ThingML, which is why mapping the elements is needed to solicit an efficient transformation. In the following, we discuss how the different CHESSIoT modeling constructs contribute to generating target ThingML elements. As shown in Figure 8.4, the transformation process starts from the top-level generation of main software components such as *VirtualElement, VirtualBoard, VirtualEntity, Sensor* and *Actuator* as the main building blocks elements. Each of those components is mapped to a ThingML thing. Each of these types undergoes a dedicated transformation route based on relevant semantics found in

[1] https://www.eclipse.org/acceleo/

the model and its typical properties to satisfy its existence in the entire ecosystem. When the transformation finishes, the tool generates CHESSIoT code licenses, the ThingML dependencies such as ThingML *DataTypes*, and *Times*. In general, Table 8.1 depict the CHESSIoT2ThingML transformation mappings implemented by the developed CHESSIoT to ThingML code generator.

```
[template public generateElement (aPackage : Package)]
[if (aPackage.name='modelComponentView')]
    [for (VE : Component |  aPackage.allOwnedElements()
                                ->filter(Component))]
    [comment check if we have a virtual entity /]
        [if(checkAppliedStereotype(VE,'VirtualEntity'))]
            [for(c:Component | getSubComponents(VE,aPackage))]
                [if(checkAppliedStereotype(c,'Sensor'))]
                    [generateSensorThing(c,aPackage,VE.name)/]
                [elseif(checkAppliedStereotype(c,'Actuator'))]
                    [generateActuatorThing(c,aPackage,VE.name)/]
                [elseif(checkAppliedStereotype(c,'VirtualElement'))]
                    [generateVirtualElementThing(c,aPackage,VE.name)/]
                [comment generate the virtual board code /]
                [elseif(checkAppliedStereotype(c,'VirtualBoard'))]
                    [generateMainThing(c,aPackage,VE)/]
                [/if]
            [/for]
            [if(checkIfVirtualCompIsThere())]
                [generateLicense()/]
                [generateDatatypes()/]
                [generateTimer()/]
            [/if]

        [/if]
    [/for]
[/if]
[/template]
```

Figure 8.4: High-level transformation steps

CHESSIoT element	ThingML element
VirtualElement, VirtualBoard, VirtualEntity, Sensor, Actuator	Thing
IoTPort	Provided/required port
Component's property	Thing's property
Component's operation	Thing's function
Payload	Message
Set of Payloads	Fragment
IoTState/Transition	State/Transition
IoTGuard	Guard
IoTEvent/Action	Event/Action

Table 8.1: CHESSIoT2ThingML transformation mapping

IoTPorts are used to support the communication between two or more components by exposing or requiring the interfaces from other components. During the transformation, *IoTPorts* of the components are transformed to the required/provided port of a ThingML's thing. Deciding on whether a given port is a required port or provided port depends on the desired direction of communication, and this can be specified in the language itself. Same as in ThingML, properties are used to retain the variable functional value of a Thing, in which during the transformation, the *component's property* is

transformed to *thing's property*.

Payload elements are mapped to Message ones in the ThingML model. For each component, all the *payloads* that it defines are collected into one ThingML element called a *Fragment*. The payload can have zero or many primitive or derived properties to be defined in a message. For instance, suppose a component message to be communicated among components contains a string value, an integer, or even an instance of another payload. In this case, a payload will include three different attributes, represented as message arguments in ThingML. Figure 8.5 depicts a fragment of the *Payload* fragment generator.

```
[template public generatePayloads(component : Component){
Payload : String = 'CHESSIoT::CHESSIoTSoftware::Payload';}]
[if (checkContainThatClass(component,'Payload'))]
thing fragment [component.name+'Messages'/] {
    [for (pay : Class | component.allOwnedElements()->filter(Class)
        ->select(pay2: Class | pay2.getAppliedStereotype(Payload)
            ->notEmpty() and not pay2.name.contains('timer')
                and not pay2.name.contains('_tic')))]
    message [pay.name/]([getPayParameters(pay)/]);
    [/for]
}
[/if]
[/template]
```

Figure 8.5: Payload generation process

```
[template public generateStateMachine (sm : StateMachine, owner : Component){
    IoTState : String = 'CHESSIoT::CHESSIoTSoftware::IoTState';
    IoTGeneric: String = 'CHESSIoT::CHESSIoTSoftware::Generic';
    StateTransition: String= 'CHESSIoT::CHESSIoTSoftware::StateTransition';}]
statechart [owner.name.toUpperCase()+'_SM'/] init [getInitialState(sm)/]{
[let states : Sequence(State) = sm.region.subvertex->filter(State)
                                                    ->asSequence()]
[for (state : State | states)]
state [state.name/]{
    [generateStateInternals(state,IoTState,owner)/]
    [let transitions : Sequence(Transition) = sm.region.transition->select(tr:
        Transition | not(tr.source.oclIsTypeOf(Pseudostate)) and
            not(tr.target.oclIsTypeOf(Pseudostate)))->asSequence()]
    [for (trans:Transition | transitions)]
        [if (trans.getAppliedStereotype(StateTransition)->notEmpty())]
            [if (trans.source.name.toLowerCase()= state.name.toLowerCase())]
            transition -> [trans.target.name/] [getTransEventCheck(trans,
                                                StateTransition,owner)/]
                [generateGuards(trans, StateTransition, owner)/]
            [/if]
        [else][printMe('Wrong state transitions used in the model component '
                                                        +owner.name)/]
        [/if]
    [/for]
    [/let]
    }
[/for]
[/let]
}
[/template]
```

Figure 8.6: State machine generation process snippet

IoTState elements are mapped to instances of *ThingML state*, same goes for *State transition* that will also be mapped to their corresponding transition provided by ThingML. As shown in Figure 8.6, the state-chart transformation process as a core is one of the complex generation processes in the

whole transformation process, and it involves two main steps. First, the generation of the internal state behaviors such as *OnEntry* action, internal events, and the *OnExit* actions. The Second step of the generation process involves generating the state transitions. This stage involves the generation of conditional events to be attached to the state transitions. In certain cases, the guards associated with these transitions are also checked for potential effects and transformed accordingly. Once the generation of Things is completed, the final task is to generate the configuration files, which encompass the instances of Things and their connections. This process aligns with a component-to-connector methodology, following the internal structure of the nodes. The above-mentioned transformation process only occurs when the corresponding behaviors have been specified and are present in the CHESSIoT model. For instance, not every system will necessarily have all three internal state behaviors present at all times.

8.3 Model-based deployment plan and run-time services provisioning

The deployment plan refers to the steps involved in planning and implementing the deployment of a software system or application. This can include identifying and specifying hardware and software requirements, determining the most appropriate deployment architecture, and creating a detailed deployment plan that outlines the specific steps and resources needed to deploy the system successfully. Docker[2] and Kubernetes[3] are two popular technologies used in modern software development and deployment. While Docker provides containerization capabilities, Kubernetes is an orchestrator for managing containerized applications.

Ideally, the software components of a typical IoT system can be deployed in the Cloud, at the Fog layer, and the Edge of the network. Designing the deployment plan of such a complex and heterogeneous system has to consider several aspects and be aware of different satisfactory requirements [223]. In fact, as in other domains, IoT software services need to follow a multi-tenant approach in which a single service instance should be running on the host servers, and that single instance serves each subscribing customer or cloud tenant [236].

IoT systems interact with humans and are always at the intersection between human survival, for instance, in the healthcare and transportation domains. As such, monitoring, reviewing and managing deployed services is necessary to avoid any operational mistake in the IoT cloud-based infrastructure. The CHESSIoT deployment environment aims to support the users in decomposing the IoT system deployment plan and managing deployed node services at all layers. The overall deployment design is depicted in Figure 8.7. Alongside the design of the deployment model, the environment also offers support for specifying deployment rules using textual grammar. This enables the expression of mechanisms to monitor the life cycle of deployed services through deployment agents.

This section comprises three parts. First, in Sec. 8.3.1, we present the approach for designing the deployment plan. Second, in Sec. 8.3.2, we delve into the design approach for service provisioning. Finally, in Sec. 8.3.3, we describe the approach for generating deployment artifacts.

8.3.1 Deployment plan design

In CHESSIoT, the deployment design showcases the physical hardware architecture for running IoT software services. It links the software architecture design to the real system architecture, outlining the nodes where the software program will be executed. The *Deployment view* allows users to break down the inter-dependency between different nodes, which may include a machine with one or multiple services running on it. The goal of this process is to generate ready to be deployed Docker compose files for each of the machines at a certain node.

The design process, illustrated in Figure 8.7, commences with the user defining the system's deployment model. This model, which aligns with the metamodel discussed in Sec. 6.4.3, primarily focuses on the interconnections between computing nodes, machines, and the IoT services they host.

[2] https://www.docker.com/
[3] https://kubernetes.io/

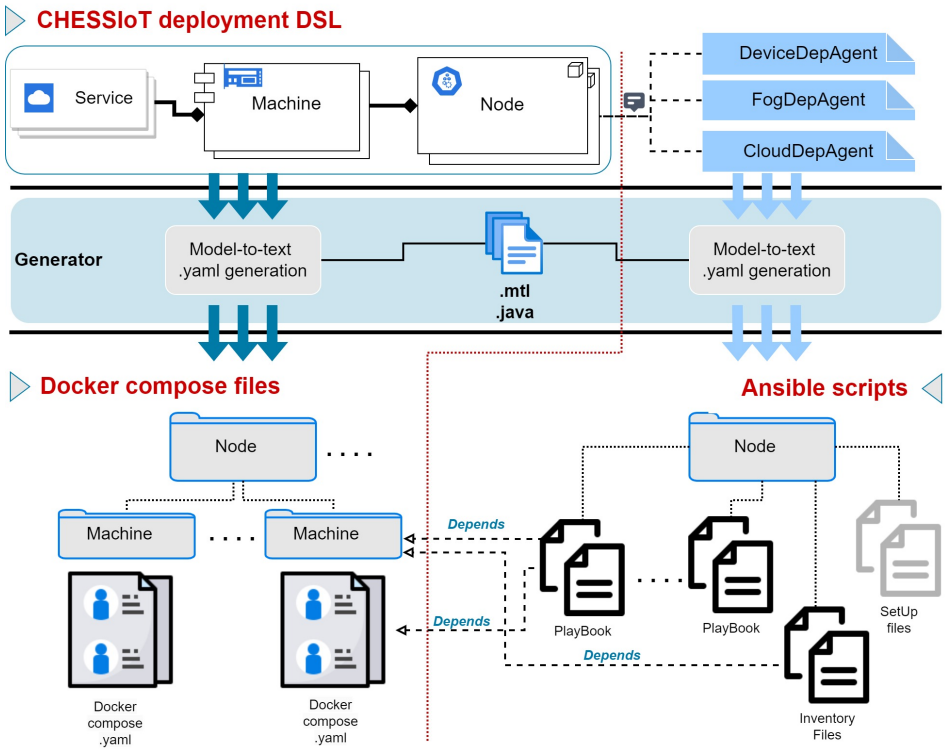


Figure 8.7: Deployment design process

Nodes play a crucial role in the entire design process, as they not only encompass the computing machines but also bear the responsibility of hosting deployment agent annotations. The inclusion of machines in the process serves to enhance the decoupling of how and where IoT services are deployed.

The definition of the deployment concrete syntax model is achieved by using the Papyrus modeling editors. A deployment context model was developed and used to create an IoT-specific deployment editor, which it easy to define element properties, inter-connection, and their intra-compositions using a rich editor. CHESSIoT context model in 8.8 primarily defines tabs, views related to the selected element, and a section as part of a view related to a given tab. The section includes the element's direct representation as a widget and a layout. Depending on the layer at which a node is, services deployed at the same layer or not could communicate between themselves. For instance, an MQTT client running on the device layer need to know the address to which a fog MQTT server is running to better communicates and vice versa.

The communication relationship between nodes can be explicitly indicated at the node level as well as down to the service itself. As previously mentioned, services could have a dependency relationship between themselves. This relationship is critical when determining the startup and shutdown dependencies between services. For instance, when running Apache Kafka in a distributed mode (i.e., with multiple brokers forming a cluster), ZooKeeper [4] is typically required to provide highly reliable coordination and synchronization for such distributed systems. In this case, Apache Kafka will have a dependency relationship to ZooKeeper in the deployment plan model. Hence, during the docker-compose file generation process a *"depends_on"* value is used and it is set to the corresponding service following the service-to-service dependency relationship it applies to (in our previous case "ZooKeeper"). Finally, the *service priority* property is used when determining the order in which individual service configurations are generated as well as their run-time prioritization later in the event of a machine memory shortage.

[4] https://zookeeper.apache.org/

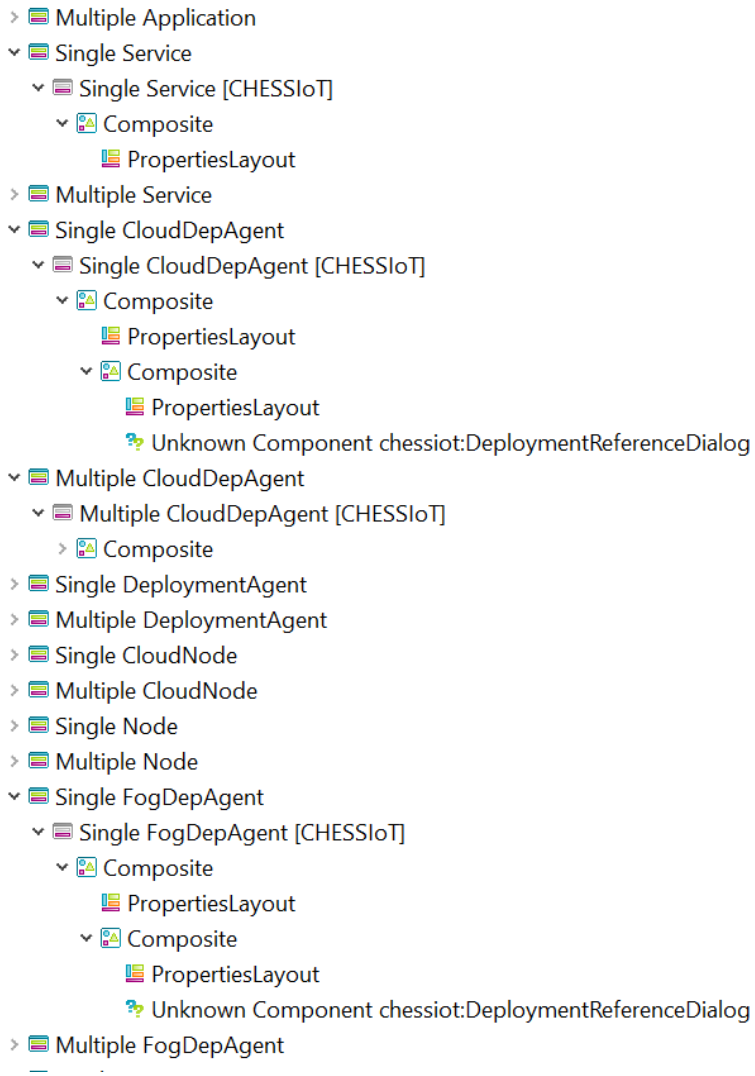


Figure 8.8: CHESSIoT context model

8.3.2 Service provisioning design

There are different ways to achieve runtime service provisioning, one of them is by using containerization technology. Docker and Kubernetes technologies enable users to package an application along with its dependencies into a container. This containerization approach facilitates the management of deployment, scalability, and runtime monitoring of these applications. However, in the present software deployment landscape, many runtime service provisioning approaches still rely on workflow-driven methods that utilize scripts and follow well-defined deployment steps. Mastering multiple deployment languages can be challenging, and there is a notable issue of tight coupling between the scripts and the specific deployment environments they are intended for. But always the question remains *"should the deployment plan change every time the target environment changes?"*

To address this challenge, the CHESSIoT approach utilizes a model-driven strategy for handling runtime service provisioning. This involves the automatic configuration and deployment of software services based on a pre-defined model. The runtime provisioning notations model integrates all the essential information about a specific type of action required at runtime, including its dependencies, requirements, and configuration settings. This information is presented in the deployment model, which includes nodes, machines, and deployed services. Depending on the client's needs, the model can be translated into any target configuration language for the desired environment. The abstract syntax for the service provisioning language is illustrated in Figure 8.9.

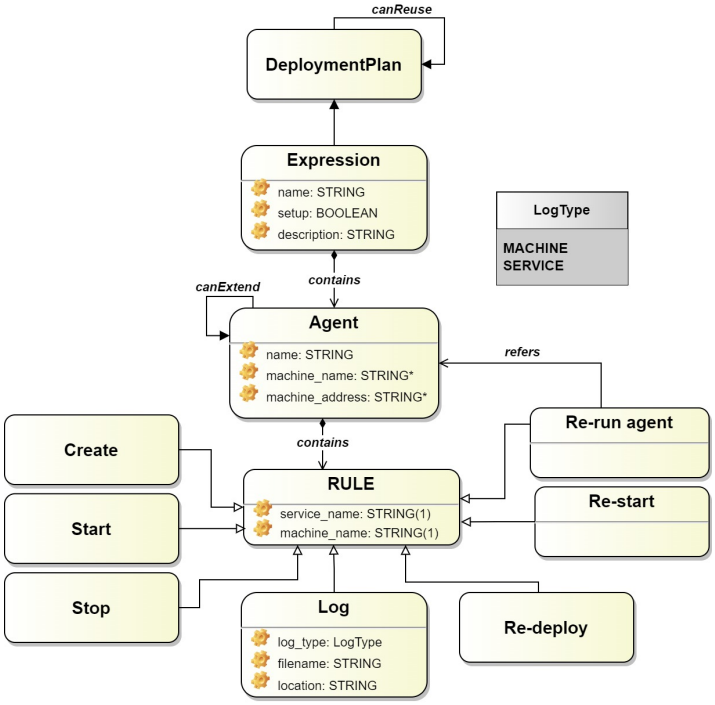


Figure 8.9: Service provisioning metamodel

CHESSIoT element	**Ansible element**
DepPlan - *Name*	PlayBook - *PlayBook filename*
AbstractAgent - *Description*	Play - *Name*
Rule - *Name* - *Arguments* - *MachineName*	Module - *Name* - *Arguments (depends on rule)* - *HostName* - *HostAddress (from the inventory)*
Set of rules	Task

Table 8.2: CHESSIoT2Ansible transformation mapping

To support the easy deployment and run-time service provisioning of the deployed services, CHESSIoT provides a textual grammar to express the means for monitoring the life-cycle of the deployed containers. At each node, a deployment plan is annotated, consisting of a collection of expressions. These expressions take the form of deployment agents, where each agent specifies a series of one-time actions to be executed on a remote machine's configuration. These activities are aimed at facilitating the deployment and provisioning of services.

As the Agents are attached to the nodes, their expressions are meant to be directly dependent on the number of machines running at such nodes, their names as well as their addresses. In practice, a deployment agent could extend another one to better avoid rewriting rules over and over in case the same or even with some additional run-time actions are applied from one machine to the other.

In addition to that, the rules which are meant to express the runtime actions that are meant to be performed on the machine are the only target "services" they are intended to support. Please note that the following rules are intended to support the services that have been already defined in the previous deployment model as well as other dependencies or supporting services that could be of interest to the efficient deployment and runtime service provisioning of a given system.

An example of run-tine service provisioning definition is depicted in listing 8.1. The *Create* rule takes into account the service name and the machine name; it is meant to create and install a containerized service at a given machine server. *Start/Stop/Re-start* rules are meant to start, stop, and re-start an already created or existing service, respectively. The *Log* rule is intended to capture either the machine logs at which the target service is deployed or the deployed service logs itself depending on the developer's needs. If needed, the location of the log file for the root directory as well as the filename can be defined. The *Re-deploy* rule is intended to recreate and restart a service on a given machine. The *Re-runAgent* is meant to re-run all the rules that are encapsulated in a given agent.

```
DepPlan:Setup{
  setup:true
}
DepPlan:Name1
{
  re-use-plan:Setup
  agent: newAgent1{
    Description:"This is a first agent"
    RULE:create=>"Service_name" on: "Machine_name"
    RULE:start=>"Service_name" on:"Machine_name"
    RULE:log=>"Service_name" log_type: machine
    filename:"Filename" location:"Filename"
    on:"Machine_name"
  }
}
DepPlan:Name2{
  agent:newAgent2 {
    Description:"This is a second agent"
    RULE:stop=>"Service_name"on:"Machine_name"
    RULE:re-deploy=>"Service_name"
    log_type: machine
  }
  agent:newAgent3 extends newAgent2{
    Description:"This is a third agent"
    RULE:log=>"Service_name" log_type:service
    filename:"Filename" location:"Location"
    on:"Machine_name"
  }
}
DepPlan:Name3{
  re-use-plan:Name1
  agent:newAgent4 {
    Description:"This is a fourth agent"
    RULE:re-runAgent=> newAgent1
  }
}
```

Listing 8.1: Run-tine Service provisioning definition example

8.3.3 Deployment artifacts generation

When the whole deployment plan design, as well as its service provision annotations, are finished, the user can perform the deployment artifacts generation through a series of model-to-text transformations. The two main types of transformations take generate different configuration files for two main tasks. First, by following the deployment metamodel presented in Sec. 6.4.3 and the concepts in Sec. 8.3.1, each node is transformed into a series of docker-compose files targeting each of the machines.

A Docker-Compose file[5], usually named docker-compose.yml, is used to configure the application's services, networks, and volumes. During the transformation process, each machine is allocated its docker-compose file which contains the docker set-up information of each service hosted by such machine. Depending on the nature of the service, another dependency file could be generated and placed in the same folder to fully satisfy the run-time requirements (e.g., security and storage).

```
[template public generateInfrastructure(name : String,machine : Component, elt:Package ){
ExternalService : String = 'CHESSIoT::CHESSIoTDeployment::ExternalService';
DataDistributionService : String = 'CHESSIoT::CHESSIoTDeployment::DataDistributionService';
MQTTBroker : String = 'CHESSIoT::CHESSIoTDeployment::MQTTBroker';
ServiceST : String = 'CHESSIoT::CHESSIoTDeployment::Service';
StorageService : String = 'CHESSIoT::CHESSIoTDeployment::StorageService';
}]
[file ('/'+name+'/'+machine.name+'/docker-compose.yaml', false, 'UTF-8')]
[generateCopyright()/]
[generateLicense()/]
version: "3.9"
services:
[for(c:Class|getSubClass(machine,elt))]
    [if(c.getAppliedStereotype(MQTTBroker)->notEmpty())]
        [generateBroker(c,machine,'/'+name+'/'+machine.name+'/')/]
    [elseif(c.getAppliedStereotype(DataDistributionService)->notEmpty())]
        [generateStorageService(c,machine)/]
    [elseif(c.getAppliedStereotype(StorageService)->notEmpty())]
        [generateDataDistributionService(c,machine)/]
    [elseif(c.getAppliedStereotype(ExternalService)->notEmpty())]
        [generateExternalService(c,machine)/]
    [/if]
[/for]
networks:
[for(c:Class | getSubClass(machine,elt))]
  [c.name/]_net:
    driver: bridge
[/for]
[/file]
[generateSetup(name,machine)/]
[generateAnsible(name, machine,elt)/]
[/template]
```

Figure 8.10: Deployment artifact generation (Acceleo)

Figure 8.10 depicts a fragment of CHESSIoT to .yaml translation code written in the Acceleo M2T transformation language. During the transformation, each service type goes through a separate transformation path before being added back to the parent configuration file. For example, if a service is of the type *"Broker"* and the anonymous access mode is set to false, different security-related files such as passwords are generated according to the user definitions. When the docker-compose configuration files generation is finished, the next step is to generate the Ansible script based on the service provision agents specified.

Ansible[6] is a powerful, flexible, and user-friendly tool designed for automating various infrastructure tasks, executing ad hoc commands, and deploying multitier applications across multiple machines [224]. Its simplicity lies in the usage of human-readable YAML templates, known as playbooks. With Ansible, users can easily program repetitive tasks to be executed automatically, without the need for advanced programming knowledge.

In the right-hand side of Figure 8.7, the generation of Ansible scripts involves three main components: set-up scripts, inventory, and playbook scripts. The set-up scripts are responsible for tasks such as installing and configuring Docker (if it is not already installed), updating the Ubuntu system, and performing other necessary setup actions. These scripts are typically used on cloud nodes. On the fog layer, the set-up process varies depending on the operating system running on the machines. Different mechanisms for basic setup and upgrades are employed based on the specific operating system requirements. The next files to be generated are *inventory files* which define the managed nodes

[5] https://www.docker.com/
[6] https://www.ansible.com/

to be automated. The host data from the deployment model are drafted to create the inventory file. The inventory file is created with groups of different machines and addresses so that the user can run automation tasks on multiple hosts at the same time. The creation of the inventory groups will be based on each deployment agent attached to the node. The Ansible playbooks are generated next.

Ansible Playbooks are sets of automated operations that need to be executed by the hosts on a remote server. They use several "*plays*" to manage multi-machine deployments on one or more hosts. Ansible Playbooks are frequently used to automate IT infrastructures, including networks, security systems, operating systems, and Kubernetes platforms. One or more *Ansible tasks* might be combined to make a play. A *Modules* have a specific activity to complete within a task. Each module contains metadata that identifies the user, the location, and the time and place at which a task is completed. During the transformation, the mapping in Table 8.2 is duly followed.

8.4 Case study: Home Automation System (HAS)

To demonstrate the capabilities of our tool, we conducted a case study on a Home Automation System (HAS), utilizing both the tool itself and the methodology described in this paper. In Section 8.4.1, we present the safety analysis of the system. Section 8.4.2 focuses on the system development, specifically addressing the modeling and code generation aspects. Lastly, in Section 8.4.3, we discuss the system deployment and the runtime service provisioning aspects.

The Internet of Things (IoT) has experienced significant market growth in sectors like industrial automation, healthcare, and transportation. As technological advancements continue to permeate various aspects of our lives, home automation is gaining increasing attention. A Home Automation System (HAS) is a technological solution that enables users to remotely control different aspects of their homes, including lighting, heating, appliances, and security, using smartphones or other devices. These systems typically combine software and hardware components, such as sensors, to automate various tasks and functions within the home. While home automation systems primarily serve energy-saving purposes, some also cater to the needs of elderly or disabled individuals, facilitating their interaction with home appliances. Figure 8.11 provides an overview of the high-level structure of the system implemented in this study.

With the scenario in mind, we could potentially explain our motivating example:

John is a software engineer and homeowner who works at a bank 40 minutes away from his home. John has installed a home automation system to control his house remotely while he is away for work. His house has many rooms, but we just consider two for simplicity. The major components he seeks to automate in a room are an air conditioning unit (AC), a light bulb, and double-hung windows. This will primarily be dependent on temperature sensor readings installed in each room, and based on that, the AC should switch on and off automatically, as will the window open and close down. Depending on his preferences, he can remotely turn on and off the light bulb as well as other appliances using his smartphone, regardless of sensor readings. The system board installed in the room is wirelessly connected to a RaspberryPi gateway, which interfaces the room system appliances with his Android phone's application. Finally, he can use his own PC at work to remote interface with his home system.

8.4.1 HAS modeling and Fault-Tree analysis

In the Home Automation System (HAS) example mentioned earlier, a temperature sensor is utilized to collect temperature values within the home. Based on these readings, the system can automatically perform certain actions, such as controlling the AC or adjusting the windows. Additionally, the system allows users to remotely control the light bulbs and windows regardless of the sensor data. Figure 8.12 illustrates the internal physical architecture of the system. For simplicity, we have depicted only two rooms and have assumed that the window actuation is directly connected to the window and represented by the servo motor. We have not accounted for alternative designs that incorporate electrical and mechanical configurations that could impact the physical functionality, such as appliances

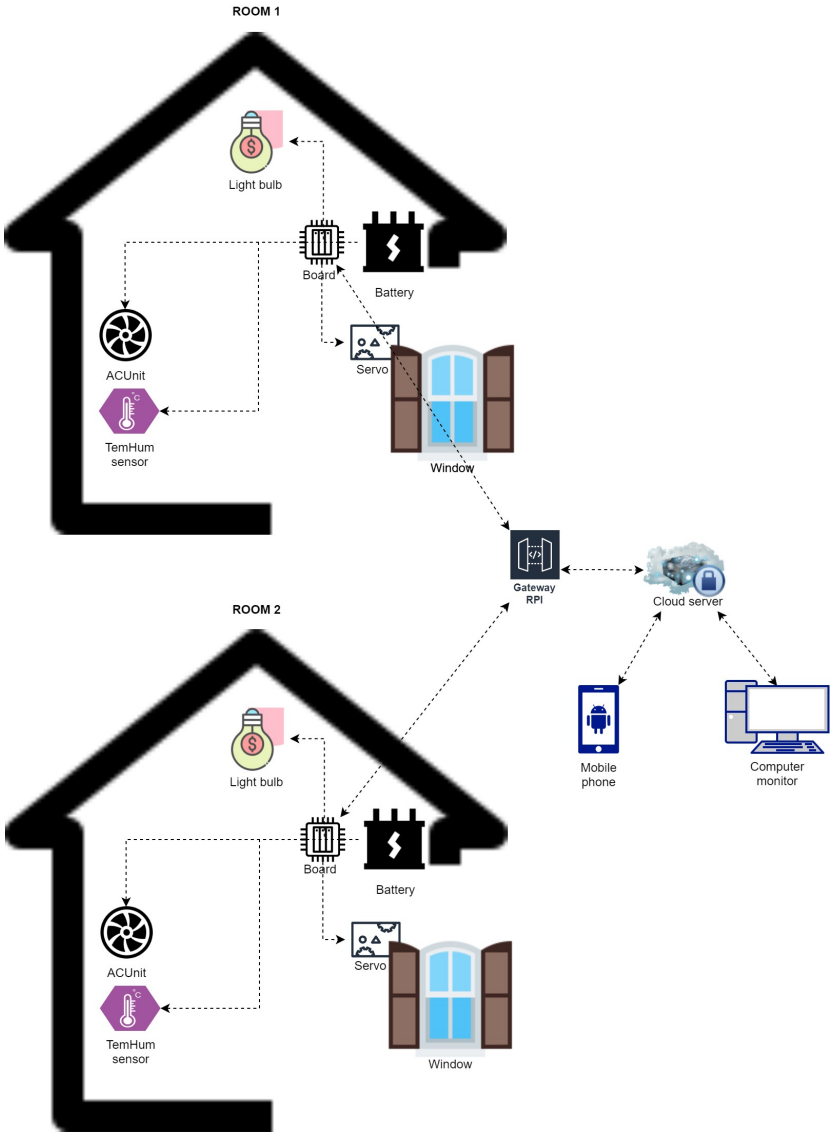


Figure 8.11: Home Automation System

requiring high power (e.g., window motors). Such considerations are beyond the scope of this study.

Figure 8.12 illustrates the power supply configuration of the system, where a single battery source supplies power to the two rooms independently. The two room components communicate individually with a central gateway. The server hosts the necessary software services for data storage, processing, and accessibility by authenticated parties. Users can access these services remotely through active devices such as mobile phones or PCs, which display the relevant information on their screens. In the event of abnormal sensor readings that exceed or fall below certain thresholds, the system may automatically trigger actions such as turning on/off the AC or opening/closing the windows. Moreover, the system should be capable of sending notifications to the user regarding any unusual room conditions. For example, John can view the displayed data on his phone or PC and choose to manually override the system's decisions by forcibly opening the windows or turning on the AC, disregarding the sensor readings.

As stated, the above system is vulnerable to several types of failures, usually generated by the system or caused by the surrounding environment. It is essential to model the failure behavior of individual components, which could be used to establish the failure behavior of all subsystems or a whole system. It is crucial to note that we do not focus on software-level functional behavior, but rather on hardware failure behavior that users could understand. To grasp the requirement for the

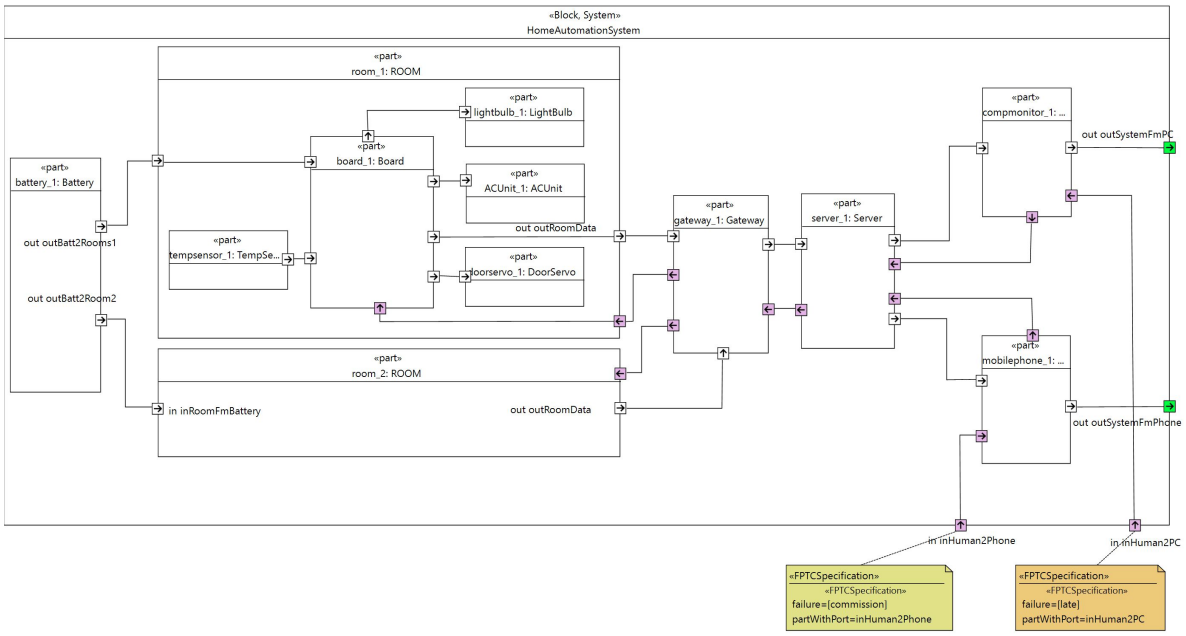


Figure 8.12: Home Automation System internal diagram

conducted analysis, let us first review the several top failure scenarios that we believe could occur at the system's output port, such as the phone or the PC.

1. **Phone/PC displaying wrong data:** This can happen at any time the data received from the server is wrong with regards to the actual data to be represented.

2. **Phone/PC is off completely and does not display any data:** This can happen either when the phone or PC does not receive any data or those entities are faulty.

Figure 8.12 depicts the system's two input ports: *inHuman2Phone* and *inHuman2PC*, corresponding to the mobile phone and PC, respectively. These ports enable us to simulate the effects of external failures on the overall system functionality. For example, the *inHuman2Phone* port can simulate a scenario where the user mistakenly turns an appliance "ON/OFF" when it is not required. This situation represents a *"commission"* failure injected externally into the system. Similarly, for the PC, we simulate a scenario where the user responds to a "LATE" system condition, indicating a *"late"* failure externally injected into the system. It is important to note that the consequences of both scenarios propagate throughout the system and elicit different responses based on the actual failure behavior of each component they encounter. The routes of failure propagation are highlighted in pink in Figure 8.12

The next step involves deriving the system components' internal failure and propagation rules. To determine the failure behavior of each component, it is necessary to understand their functional behavior. Let's consider the example of a sensor. A sensor can fail in two different ways. First, it may completely cease providing data (resulting in an ***"omission"*** at the output port). Alternatively, the sensor may experience internal failures, such as inaccuracies in the readings or data values outside the expected range (leading to a ***"valueCoarse"*** at the output port). We can establish distinct failure rules for these scenarios, as depicted in Equation 8.1 and 8.2, respectively. It is important to note that the asterisk notation denotes an unknown source of failure in cases where a component does not possess any input ports. Additionally, other components like the power battery, gateway, server, ACUnit, etc., can fail by ceasing to provide power (*omission at the outputs*). A comprehensive list of failures and detailed descriptions can be found in the table set provided in [238].

$$FLA : (*) \rightarrow outSensor2Board.\textbf{omission} \tag{8.1}$$

$$FLA : (*) \rightarrow outSensor2Board.\textbf{valueCoarse} \tag{8.2}$$

Once the failure behavior specifications of the components are finalized, the safety expert can assign basic failure probabilities to aid in quantitative analysis. Determining the failure probability of a component can be a challenging task. It is recommended to consult the device manufacturer's documentation, and industry standards, or seek advice from device experts. In safety engineering, the device failure probability is often considered to be extremely low and often expressed as failures per million (10^{-6}), particularly for individual components [231]. To maintain simplicity, we have set a default probability value of $4 \cdot 10^{-5}$ for all basic failure events. It is important to note that the Fault Tree Analysis (FTA) can also be applied at the sub-composite component level, such as the ROOM level. This allows for investigating the potential impact of failures originating from internal sub-components, as well as the effects of externally injected failures on the behavior of internal components.

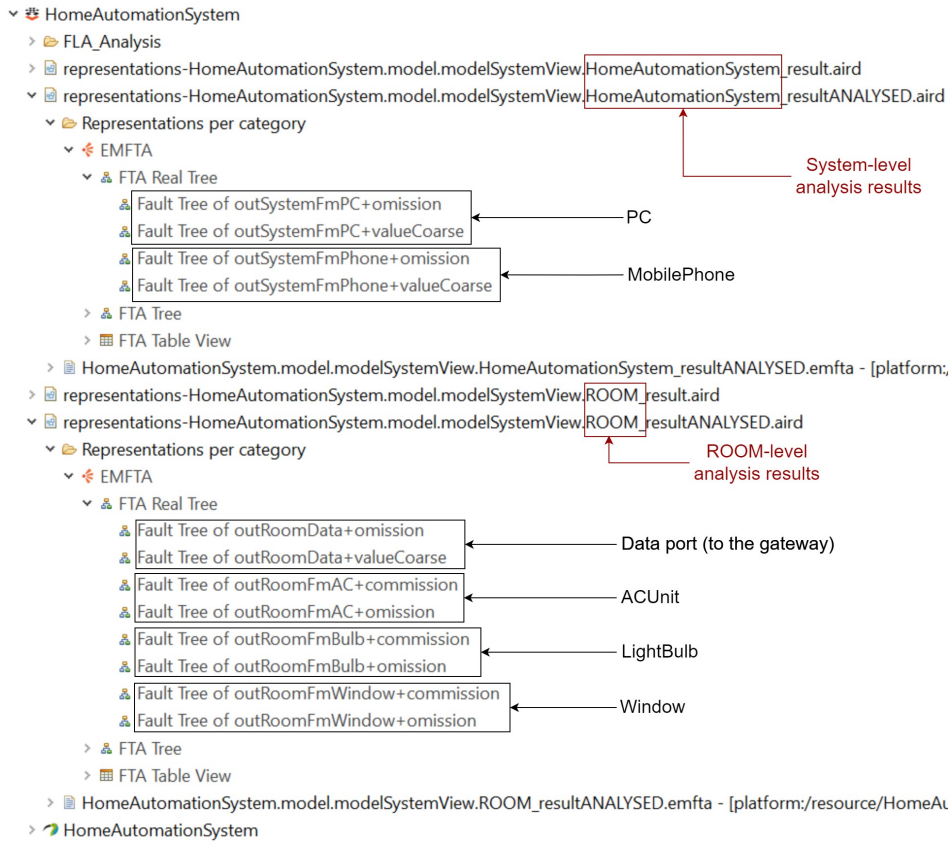


Figure 8.13: Analysis results

Upon completing the analysis, fault-tree models are generated based on the failures that have propagated to the system's output port. The analysis results are represented for both the "ROOM-level" and the "System-level" in Figure 8.13. In particular, the *"omission"* and *"valueCoarse"* failures have propagated to the system's *outSystemFmPC* and *outSystemFmPhone* output ports. Furthermore, at the ROOM-level, the *"commission"* and *"omission"* failures have propagated to the output ports of components such as ACUnit, LightBulb, and Window, whereas the *"commission"* and *"valueCoarse"* failures have propagated to the *outRoomData* port which sends data to the gateway.

An FT is generated and analyzed accordingly for each analysis result described above. The two system-level propagated failures match the two big top failure scenarios described earlier. Fig. 8.14, Fig. 8.15, and Fig. 8.16, present generated and analyzed FTs for both at the Room level as well as at the system level.

Figure 8.14 shows an analyzed fault tree in the situation where the window stops working totally. As can be observed, at the low lever, there are three basic events: *"a completely broken board"*, *"a completely broken sensor"*, and *"an external failure related to the battery, for example, a drained*

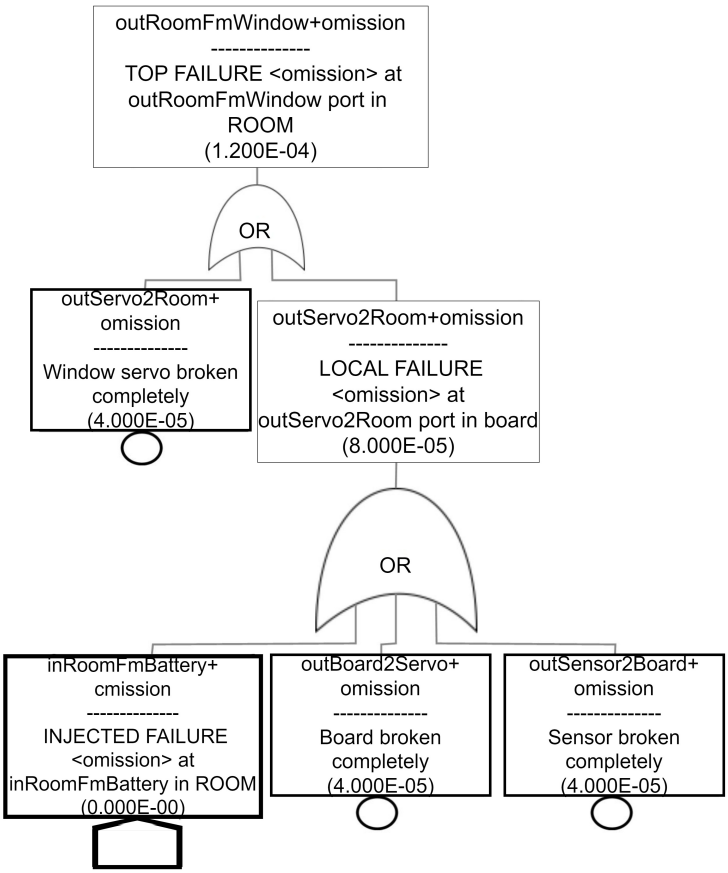


Figure 8.14: Room-level FT diagram: *Window not working (omission at the window output port)*

battery". Based on the shown fault tree in Fig. 8.14, the three basic events are fed into an "OR" gate, which means that if any of them occurs, it will flow directly through the gate. As the simulation evolves, the resulting intermediate event is OR'ed with the *"window servo broken completely"* internal failure resulting in the undesired top failure. Eventually, one of the four basic failures will directly propagate to the output port.

Overall, the top-level undesired event probability for such a scenario is estimated to be $1.2 \cdot 10^{-4}$. Because the scope of the room is substantially smaller than that of the system, the external event, in this case resulted from the injected failure from the battery port and was assigned a probability of zero. This may appear irrational, however, to obtain the probability values of such an event, the entire system's probability must first be computed and then assign the corresponding probability value to such an event. We plan to tackle this issue in the future.

In Figure 8.15, we present another example of a room-level Fault Tree (FT) that illustrates an event where the ACUnit unexpectedly switches on and off, resulting in a "commission" failure at the ACUnit's output port. The diagram includes two external events: one located at the bottom right, representing an external "valueSubtle" failure injected from the outside due to a late reaction from the PC, and another event in the middle-left depicting the user pressing the commanding button when it is not needed. Both scenarios elicit different responses from the system. It is important to note again that the two external events labeled as injected failure events are beyond the scope of the current analysis context, and thus no probability can be assigned to them at this stage.

Finally, at the system level, we discuss the fault tree depicted in Fig. 8.16. It shows the fault tree in which the "Mobile phone displays erroneous data," inferring a "valueCoarse" failure propagating at the output port. In this situation, the two rooms will have equal control over whether the data on the display is totally incorrect. Two rooms in the tree have the same sub-tree since they are from the same instance, and their failure outcomes are joined by a "AND" gate. According to the above tree,

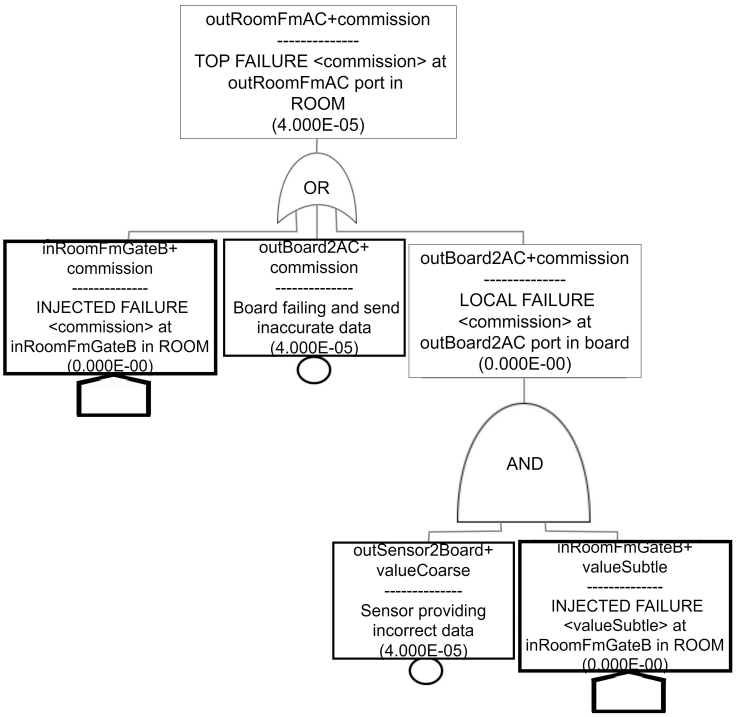


Figure 8.15: Room-level FT diagram: *ACUnit turn on and off when not expected*

erroneous sensor data in an event with a late reaction from the user PC will permit erroneous data to propagate up the tree. The event in which the "Board is failing and sent inaccurate data" will also play a role in the loop.

Maintaining coherence between the system and the safety model can be challenging when the model grows in size and complexity. Having a framework that can automate the safety analysis process by allowing the safety expert and the IoT engineer to work on the same problem from the same unique environment can potentially improve transparency while significantly reducing the time required to perform such rigorous analysis tasks. Based on the previous findings, we discuss the feasibility of establishing a collaborative analysis mechanism in which both parties collaborate to keep the system and safety model up to date, thereby improving consistency throughout the process.

8.4.2 Software design and development

The software development approach supported by CHESSIoT encompasses the functional and behavioral design aspects of the system, along with code generation, with a specific emphasis on the edge layer. These aspects are described in detail in Section 8.2. In this section, we will primarily focus on the Room level, providing models for the functional and behavioral aspects of its sub-components: the *Temperature sensor*, *ACUnit*, *Bulb*, and *WindowServo*.

To maintain continuity with our previous system-level model, which is presented in Figure 8.12, we refer to it as a point of reference.

Behavior modeling

The internal component model representation of the room is depicted in Figure 8.17. This model illustrates the structure and interactions of the sub-components within the room, offering insights into their functionalities and behaviors. As shown in the figure, communication between components is accomplished by sending and receiving payload messages over provided and required ports. A set of payloads initiated by each component is created internally, and a pin number is specified for each port of the actuating or sensing component to be connected to the board. For instance, two payloads

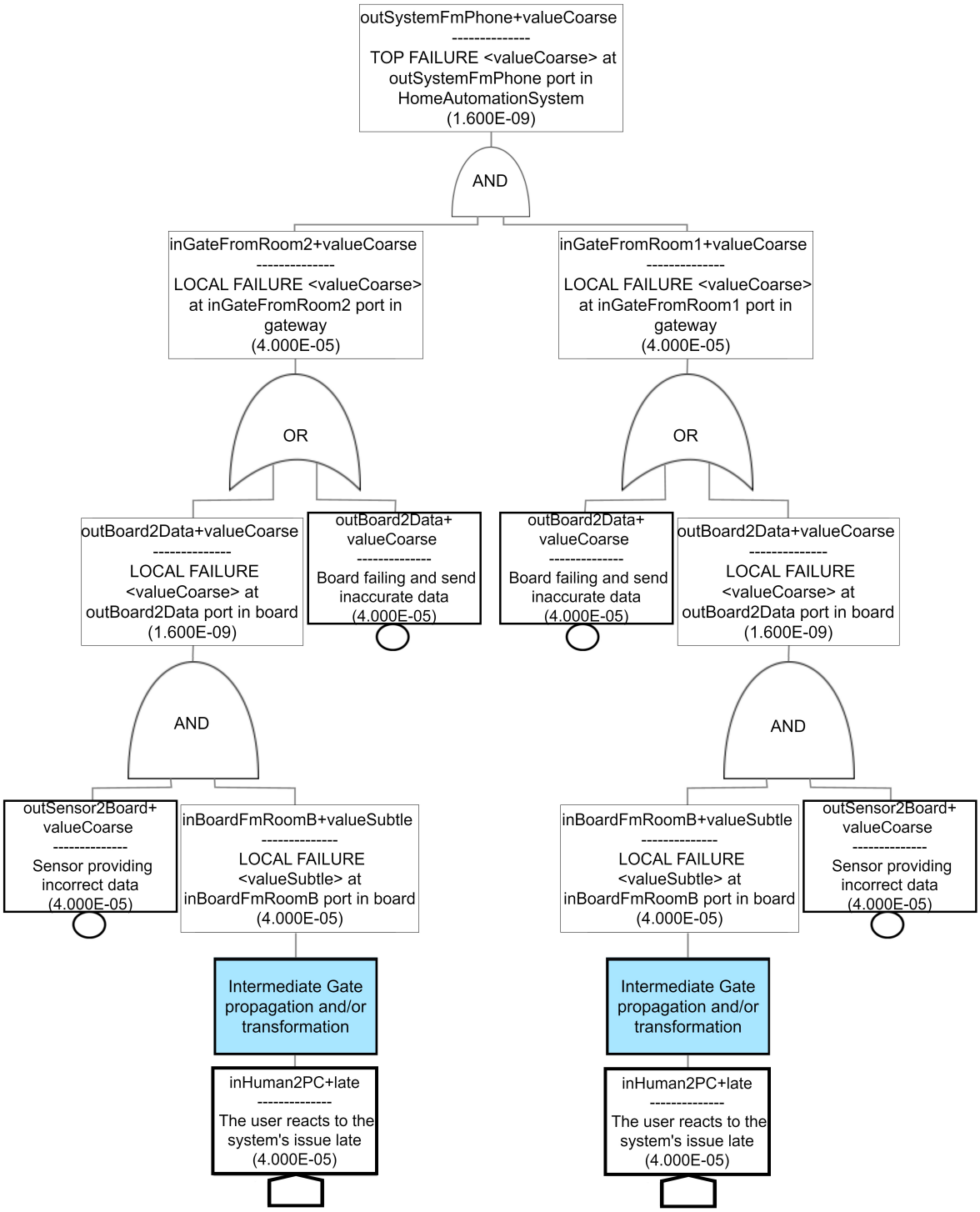


Figure 8.16: System-level FT diagram: *Mobile phone displays inaccurate data scenario*

(i.e., ON/OFF or OPEN/CLOSE) are defined for each actuating component, namely ACUnit, LightBulb, and WindowServo, to be used when communicating with the board. Furthermore, the generic actions that are fired were defined internally to set the associated pins HIGH or LOW accordingly. Furthermore, internal events are initiated for each component to determine whether there is a received actuating payload from the board via the dedicated port.

Each component is associated with its respective state machine. The working principle of actuating components is to react on a source of electric energy received to move or change the physical state of something. From that, each actuating component state machine has been assigned a single state.

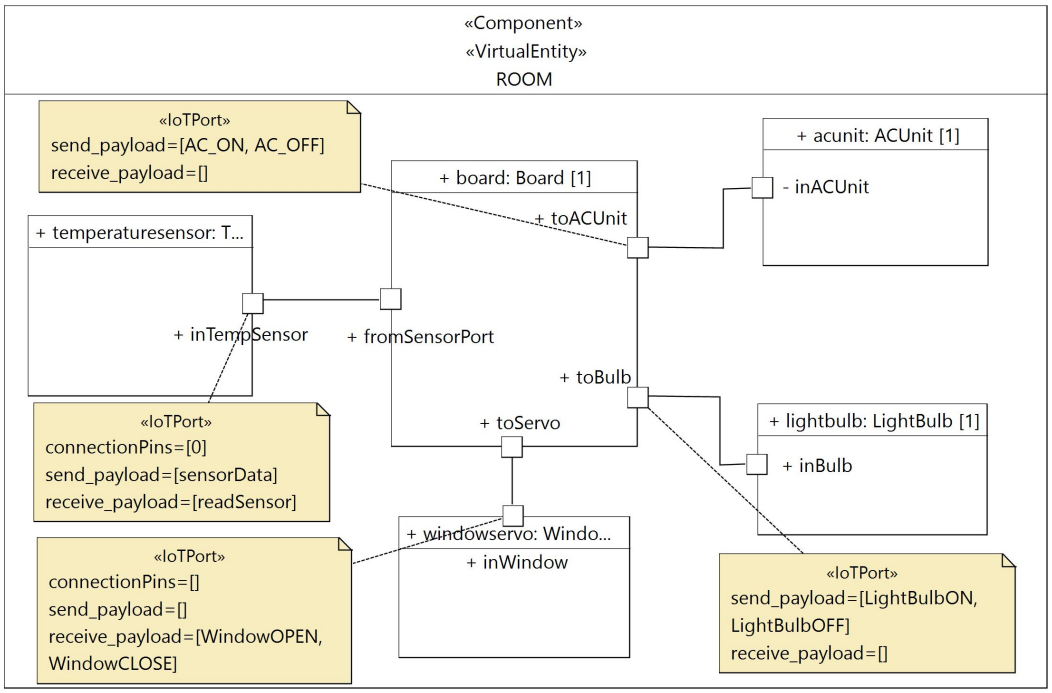


Figure 8.17: Room internal composite structure

In this case, it waits for the board's command and reacts accordingly using the previously defined actions. A sensor, on the other hand, has a state called "Sensing" in which it constantly monitors the *"readSensor"* communication payload from the board to sense the temperature and send a new payload "*sensorData*" the board through the same port (Figure 8.17).

The board serves as a central computing component in the process, coordinating all connected elements by reusing previously defined actions, payload, and guards. Figure 8.18 shows a partial caption of the inner events and guard defined within the board. As we can see, only the necessary internal events, such as checking if the sensor data has arrived to send the ON/OFF commands to the appliance (i.e *HighSensorDataReceived* and *LowerSensorDataReceived*), as well as conditional events i.e:*High_to_low* and *Low_to_high*) and transition guards *ValueLow* and *ValueHigh*) to be fulfilled accordingly when transiting from one state to the other.

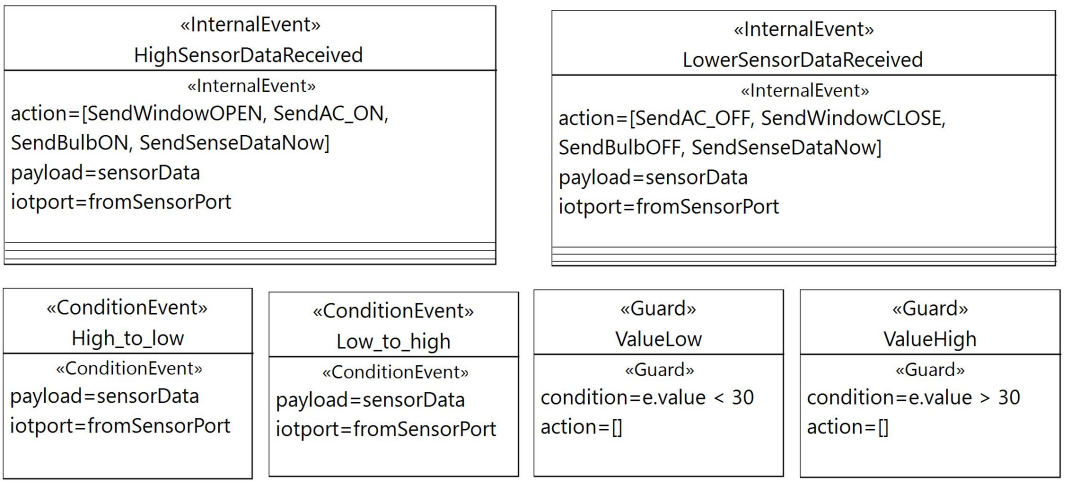


Figure 8.18: Portion of the Board event, action, guard specification

As we can see from the board state machine figure 8.19, three main states are defined, namely "*IDLE*", "*AC_OFFBulbOFFWindowClose*" as well as the "*AC_ONBulbONWindowOpen*" states are

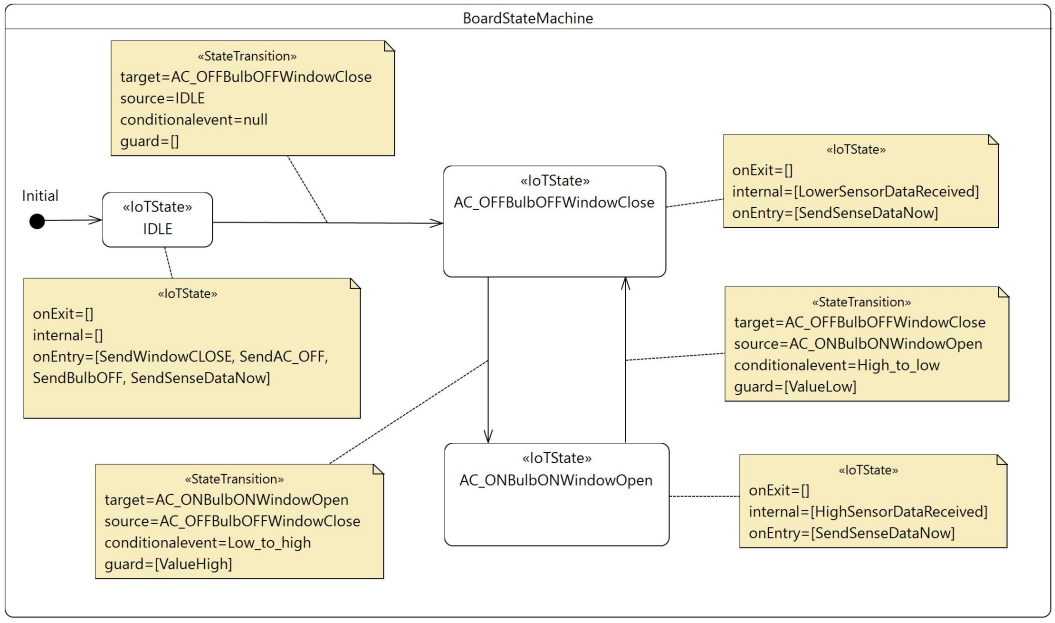


Figure 8.19: Board state machine

defined to control the basic behavior of the board. In each of the two main states, the internal event internal events such as *HighSensorDataReceived* and *LowerSensorDataReceived* are used to send trigger the ON/OFF actions accordingly refer to Figure 8.18. Coming from the "IDLE" state, the transition from the "*AC_OFFBulbOFFWindowClose*" state to the following "*AC_ONBulbONWindowOpen*" state happens when the conditional event check is confirmed (i.e: we are still getting the payload being sent at the sensor port) as well as the guard condition is fulfilled (in our case we choose to go for a temperature of 30 degree Celsius as a threshold).

Code generation

When the functional and behavior modeling is done, the CHESSIoT2ThingML transformation is launched to generate the ThingML model files ready to be compiled in the ThingML environment for generating platform-specific code. The transformation process follows the mapping presented in 8.1. Figure 8.20 depicts the structure of the generated ThingML models infrastructure. During the transformation process, each of the Room's sub-component is transformed into its unique ThingML model. Furthermore, the utility files such as license files as well as the global data types and timing messages are generated separately.

Figure 8.21 depicts the generated ThingML model of the board mapped back to the state machine diagram presented in Figure 8.19. As we can see from Figure 8.20, the ThingML model associated with the board model is generated in the parent folder of the *Room* and it imports all of its connected siblings. This gives the board the possibility to have access to the message of all the other components. For instance, at this level, the board can use its port to send and receive payloads from other components through its *required ports*. Each of the states indicated in the state machine diagram is converted into a ThingML thing's state with all its internal actions and events transformed accordingly.

Upon executing the transformation process, the code generator generates the configuration code, adhering to a component-to-connector architecture [37] that aligns with the Room's internal structure. As depicted in the bottom-left section of Figure 8.21, the configuration code instantiates all components as ThingML objects. From there, internal connections are established by linking the corresponding ports of these objects. Additionally, the properties of each Thing object are set to their original values as specified in their respective Things.

The current CHESSIoT generator supports the ThingML code generation compiled into Arduino

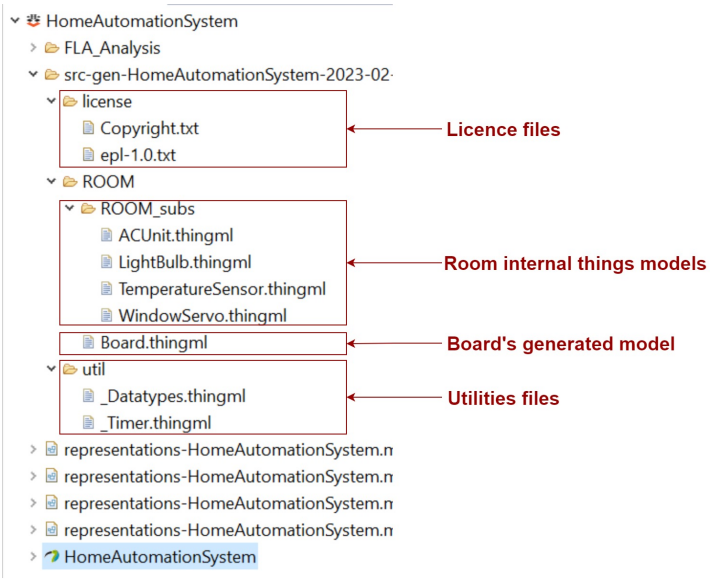


Figure 8.20: Generated ThingML models

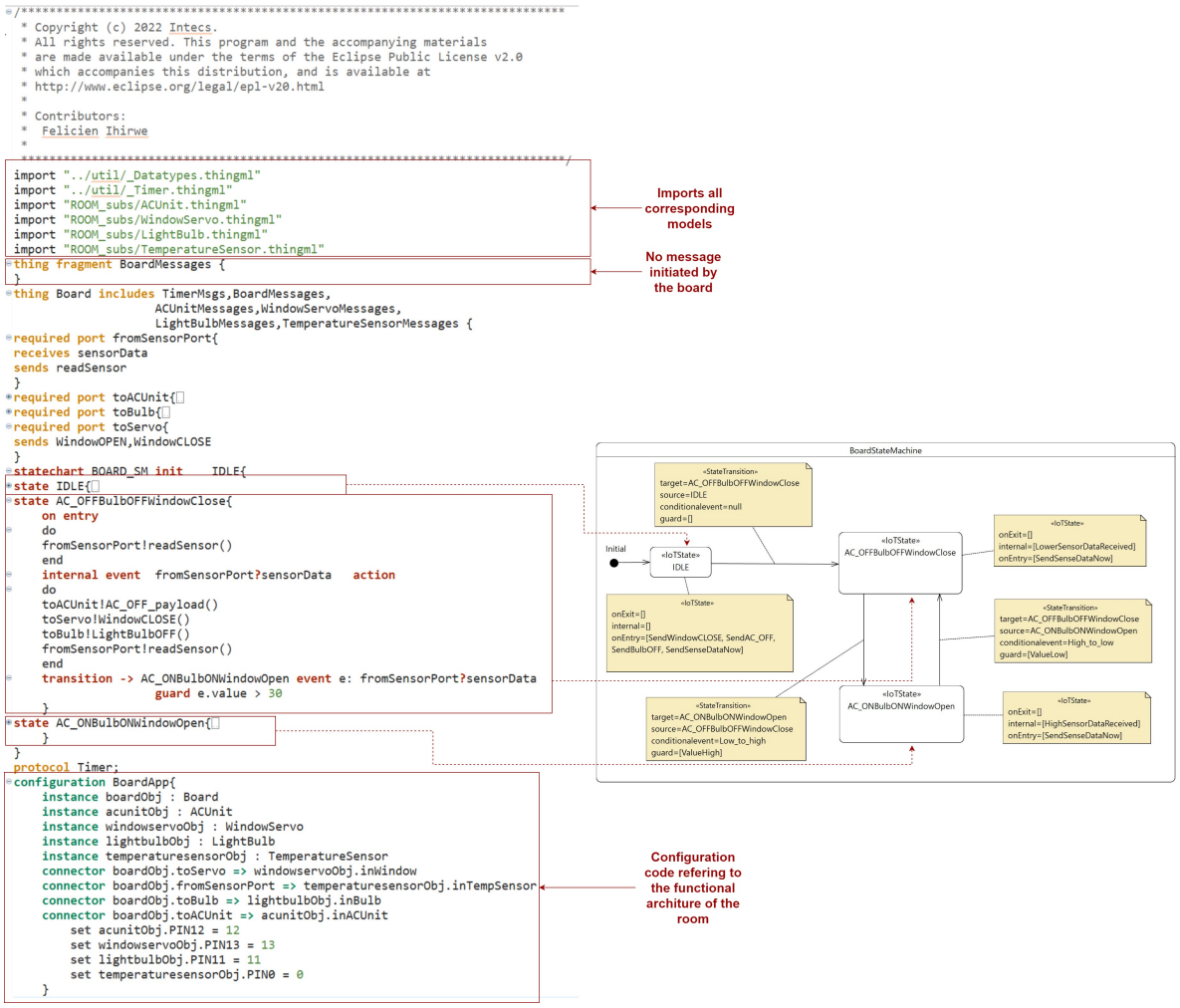


Figure 8.21: Generated Board's ThingML model mapped to the state machine diagram

code and from the generated models we could successfully generate Arduino code ready to be deployed on IoT devices. To validate the generated code, we have successfully deployed the generated

code without any single change in the same project designed in the Proteus Simulation software [7] and the code worked perfectly as expected. The full example with all the materials is online available at https://github.com/fihirwe/HomeAutomationSystem.git

8.4.3 Deployment and service provisioning

In this section, we cover the "Home Automation System" deployment designs as well as the deployment artifact generation aspects.

HAS Deployment plan design

As we described above, the HAS system involves the code running at all layers, namely the edge device, i.e., Arduino, and the mobile phone, at the Fog, i.e, a RasberryPi running the MQTT broker, as well as the on the cloud i.e-, a web server running a Node-RED dashboard instance. Figure 8.22 shows the deployment model of the system. As can be seen, all three main node layers are present, namely "DeviceNode", "FogNode", and "CloudNode" reflecting the level of computation involved other than the device layer.

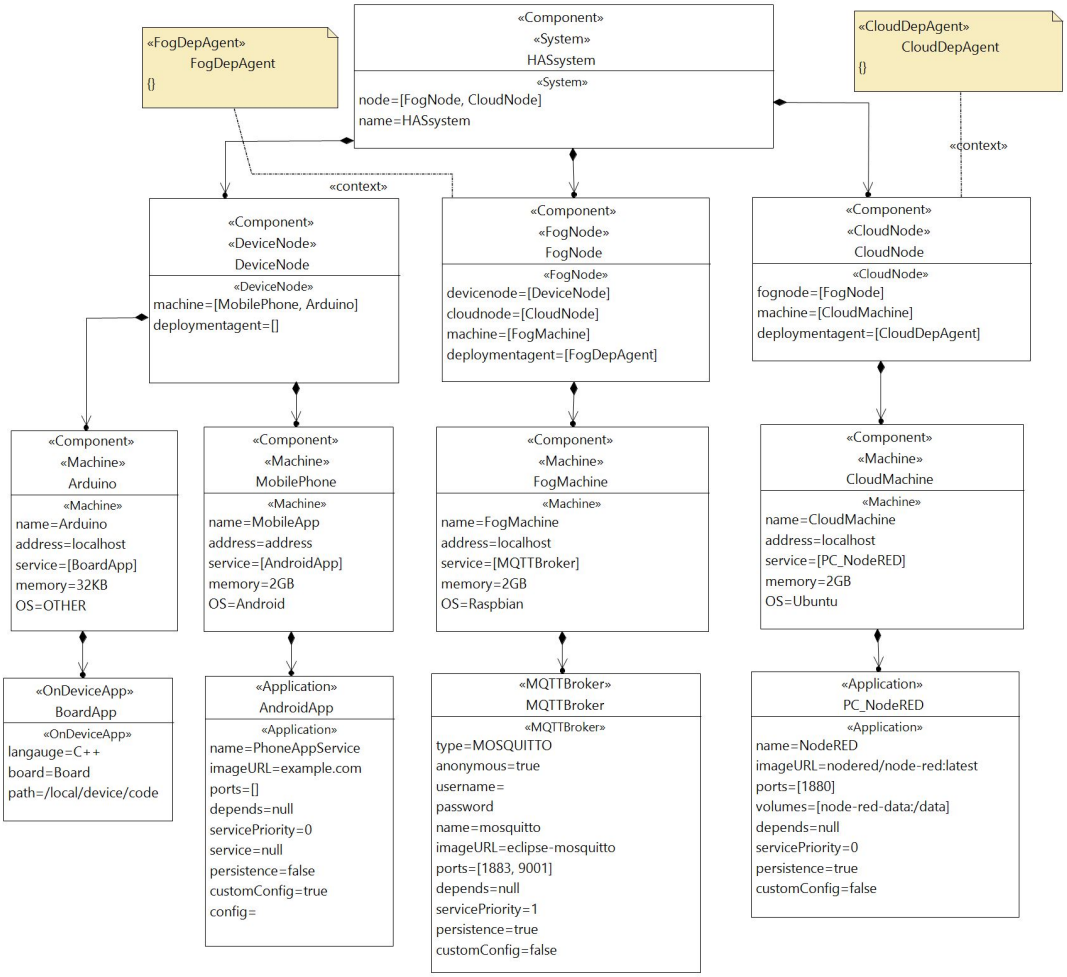


Figure 8.22: HAS system deployment plan

At the edge layer, two machines are defined namely to reflect the generated Arduino code running at the Arduino micro-controller as well as the Mobile-Phone as a machine running the Android app. The deployment at this layer is done manually and for now, no automation is provided. This is mainly

[7] https://www.labcenter.com/

due to the computing limitation presented by such chosen deployment platform for this example. At the FogNode, one machine running a Raspbian operating system was chosen to host the MQTT broker, which receives the communication from the edge layer (i.e., the Android app as well as the system code running on the Arduino code which publishes/subscribes messages to it).

As shown in Fig. 8.22, the Broker allows anonymous connections, so no username and password are needed to be created for such a server. Furthermore, the service priority property at this stage doesn't matter because we have only one service running on such a machine. This is typically used as a priority reference during the run-time management of services as a given machine. The persistence is set to true in which, during the transformation, the default Eclipse-Mosquitto broker persistence directories are chosen by default. Finally, at the cloud layer, one Ubuntu-based machine is used to host both the Node-RED dashboard instance.

During the transformation process, a docker-compose file is generated for each machine at any layer. These docker-compose files contain the necessary information for hosting the services on each machine. However, due to missing information and service incompatibility at the Device layer, the generated docker-compose file in this case is incomplete and cannot be used. For illustration purposes, the generated docker-compose file at the Fog layer is presented in Figure 8.23.

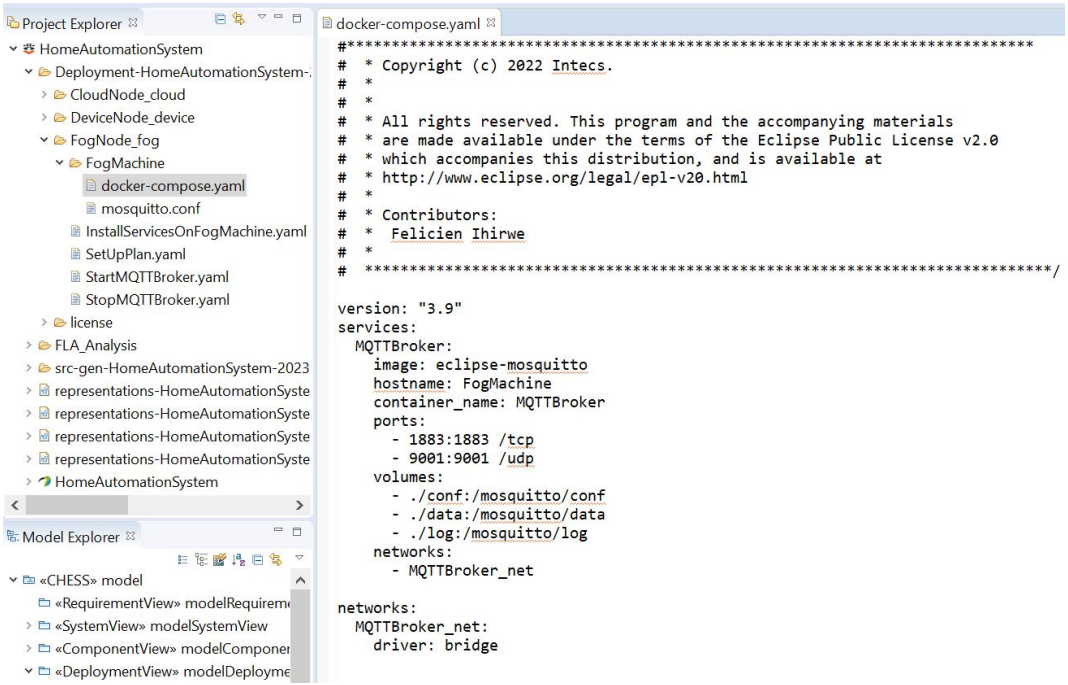


Figure 8.23: Generated deployment configuration Fog

HAS runtime service provisioning

As shown in the deployment plan model in Figure 8.22, the FogNode and the CloudNode elements are annotated with their corresponding FogDepAgent and CloudDepAgent, respectively. These annotations enable the definition of runtime deployment rules associated with the runtime management of the services deployed at each of the machines running at the node.

In Figure 8.24, we present an example of agent rules defined at the edge node. The provided agent includes four distinct deployment plans. The first plan, known as the setup plan, is responsible for installing all the necessary dependencies on the target machine. This setup plan is highly dependent on the specific target host, and the actual setup tasks will be defined accordingly.

Furthermore, the *"installServiceOnFogMachine"* plan will create and install the MQTT broker instance at the target fog machine. In addition to that, on the next plan, a *"StartMQTTBroker"* plan is defined to start and save the broker logs in the file with the name specified. By default, such a file will

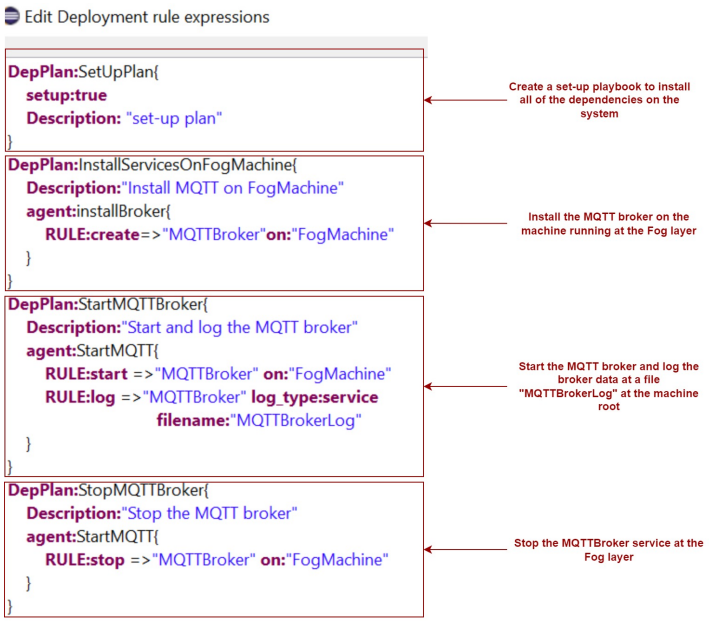


Figure 8.24: FogDepAgent rules

be located in the root folder of the machine server. The log type in this case is set to service to limit the logging to the Service log, not the machine host in which the broker is running. Note that during the transformation process, each of the *"DepPlan"* is translated into the corresponding playbook with its corresponding name. At this stage, the playbook can be used and launched separately depending on the user's need (see Fig. 8.23).

8.5 Conclusion

This chapter presented CHESSIoT, a model-driven environment for developing and deploying multi-layered IoT systems. In the chapter, we illustrated the CHESSIoT approach for software modeling IoT systems across all three primary layers. A deployment modeling approach, together with a run-time service provisioning approach was presented. Through the use of a home automation case study, we demonstrated the fully CHESSIoT tool capability for conducting both qualitative and quantitative safety analyses, model software aspect of the system and generate a set of fully functional ThingML source models, which are then used to generate platform-specific code ready for deployment on low-level IoT devices. Finally, a full deployment model was designed and generated deployment artifacts ready ti be executed on a remote docker environment. In the future, we intend to provide testing support for generated code, with the outputs potentially assisting in the recommendation of any potentially missing safety rules. Finally, we plan to enhance the qualitative safety analysis mechanism by enabling the generation of minimal cut-set FTs.

Chapter 9

Conclusion and future work

9.1 General contributions

We summarize the key contributions of this thesis as follows:

1. **Low-code engineering and its current adoption in IoT software development domain**

   In this dissertation, we have covered the current state of the art of LCE, particularly in the IoT domain, and how the model-driven approach is playing a huge role in its realization. We have mapped its similarities and differences with respect to existing trends in LCDPS as to where MDE fits in the loop. We have presented the analysis that has been performed by conceiving a taxonomy, which has been formalized as a feature diagram presenting all the features of a typical modeling platform supporting the engineering of IoT systems. We have conducted and presented the current state of the art regarding which IoT domain developers adopt cloud-based modeling technology revolutions. The considered approaches have been analyzed to assess their strengths and weaknesses concerning many characteristics, including their modeling focus, accessibility, openness, and artifact generation. We have also presented the challenges that are being faced in order to migrate the legacy platforms into cloud-based low-code ones. For instance, issues such as the extensibility of the legacy platforms, and IoT system complexity, in general, make the learning curve route even more hard and scalability concerns. We also conceived some opportunities that this initiative could bring to the community such as attracting more citizen developers, collaborative modeling, low cost of maintenance, and so on.

2. **Software product quality evaluation of Low-Code/MDE engineering platforms**

   Evaluating the quality of engineering platforms, especially in IoT, is still an open issue due to the technological revolution that the IoT domain experience to enhance our everyday livelihood. This thesis presented a model for evaluating the quality of IoT Low-code/MDE engineering platforms targeting the software product quality features. The presented model is based on and extends the ISO/IEC 25010:2011 software product quality model [14] standard targeting to help IoT practitioners in assessing and establishing the software quality requirements for engineering IoT platforms. Among others, security and maintainability features were found to be less addressed, whereas functional appropriateness, portability, and usability were found to be the most addressed. The model has been used to evaluate the software quality of 17 IoT platforms in which it was discovered that the overall quality performance of considered IoT engineering platforms (MDEs and LCDPs combined), regardless of characteristics and sub-characteristics, was about 45.5%, in which MDE accounts for 39.6%, whereas LCDPs have 51.1%. In general, the proposed model could be used mostly to evaluate the software platforms' static and dynamic properties rather than the quality of the outcome of interaction when a product is used in a specific context.

3. **Supporting model-based safety analysis of IoT systems**

   As the current technological revolution evolves around the improved quality of life of human beings, the safety of that system needs to be well studied and certified beforehand. The IoT domain is one of the emerging domains; fewer approaches have been developed to assess the safety of the system under development at its earliest stages. In this dissertation, we presented the CHESSIoT safety analysis approach for IoT systems based on the Fault-Tree Analysis technique. The presented approach relies on and extends CHESS Failure Logic Analysis (CHESS-FLA) [44], a methodology that enables the user to model the system's failure behavior, run the Failure Logic Analysis, and propagate the analysis results back onto the original model [45]. In addition to its ability to generate the system's complete FTs, the new FTA approach automatically performs qualitative analyses by eliminating unnecessary paths and redundancies in the FTs' events. Finally, the proposed approach also calculates the failure probabilities of an entire system from its constituent parts' failure event probabilities. In addition to a short comparative analysis conducted based on related works, a safety critical example was used to showcase the capability of the tool. The proposed approach can hugely help in predicting the impact of a component change or architectural change on a system in a very cheap way [81]. For instance, in case of an essential failure behavior occurring at the model system level, it will be easy to discover the source of the fault immediately and identify where the fault tolerance measures should be directed in the architecture to mitigate them.

4. **Supporting model-based development and deployment of IoT systems**

   In addition to the safety analysis contribution, the CHESSIoT engineering environment presented in this thesis gives the user the opportunity to perform the modeling, development as well as deployment tasks of IoT systems. The new approach employs a range of DSL for each task to better enhance the separation of concerns as well as the model's correctness. CHESSIoT brings a unique possibility for the user to design, develop, analyze, and deploy engineering IoT systems all from the same environment. Through CHESSIoT, a user can benefit from a multi-view development environment in which each of the supported views has its own underlined constraints that enforce its specific privileges on model entities and properties that can be manipulated. The software model containing the system's functional and behavioral aspects is transformed to ThingML [37] models, which eventually can later be transformed into platform-specific code. A deployment modeling approach together that supports the users in decomposing of IoT system deployment plan as well as managing deployed node services at all layers, namely Edge, Fog, and Cloud was presented. Finally, CHESSIoT offers a model-driven runtime service provisioning environment that allows the automatic definition of software services' life cycle based on predefined rules referred to as agents. To evaluate or approach, we have presented results from two different comparative analyses, and discussions were developed, taking into consideration modeling support as well as other supporting activities, revealing an averaged gap of 54.34%, which CHESSIoT potentially addresses.

9.2 Publications

The following contains a list of all research publications that I was involved in during the course of my PhD research.

9.2.1 Journal papers

1. **Under review:** **Felicien Ihirwe**, Katia Di Blasio, Davide Di Ruscio, Simone Gianfranceschi, and Alfonso Pierantonio. *"Supporting the model-based safety analysis for safety-critical IoT systems"* Submitted to Journal of Computer Languages (May 2022) **This paper was presented in Chapter 7**

2. **Under supervisor review:** **Felicien Ihirwe**, Davide Di Ruscio, Simone Gianfranceschi, and Alfonso Pierantonio. *"A model-driven environment for engineering multi-layered IoT systems"* Submitted to Journal of Computer Languages (May 2023) **This paper was presented in Chapter 6 and 8**

3. Murorunkwere Belle Fille, **Felicien Ihirwe**, Idrissa Kayijuka, Joseph Nzabanita, and Dominique Haughton. *"Comparison of Tree-Based Machine Learning Algorithms to Predict Reporting Behavior of Electronic Billing Machines"* Information Volume:14, Issue no. 3: 140. February 2023. https://doi.org/10.3390/info14030140 **Not presented in this thesis**

9.2.2 Conference papers

4. **Felicien Ihirwe** , Davide Di Ruscio, Simone Gianfranceschi, and Alfonso Pierantonio. *"Assessing the Quality of Low-Code and Model Driven Engineering Platforms for Engineering IoT Systems"*. In Proceedings of the 22nd IEEE International Conference on Software Quality, Reliability, and Security (QRS22). 12 Pages. November 2022. Available at SSRN: http://dx.doi.org/10.2139/ssrn.4267269. **This paper was presented in Chapter 5**

5. Alberto Debiasi, **Felicien Ihirwe** , Pierluigi Pierini, Silvia Mazzini, and Stefano Tonetta. *"Model-based Analysis Support for Dependable Complex Systems in CHESS"*. In Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development - MODELSWARD'21. ISBN 978-989-758-487-9; ISSN 2184-4348, pages 262-269. DOI: http://doi.org/10.5220/0010269702620269. February 2021 **This paper was presented in Chapter 2**

6. **Felicien Ihirwe**, Giovanni Iovino, and Davide Di Ruscio. *"Towards an MQTT5 geo-location extension for location-aware applications"*. In the 44th IEEE International Conference on Telecommunications and Signal Processing (TSP'21). July 2021. DOI: http://doi.org/10.1109/TSP52935.2021.9522590. **Not presented in this thesis**

7. **Felicien Ihirwe** , Davide Di Ruscio, Silvia Mazzini, and Alfonso Pierantonio. *"A domain-specific modeling and analysis environment for complex IoT applications"*. In the 7th Italian Conference on ICT for Smart Cities And Communities (I-CiTies'21). September 2021. https://arxiv.org/abs/2109.09244 **This is an extended abstract that introduces the approach presented in Chapter 2**

9.2.3 Workshop papers

8. **Felicien Ihirwe**, Davide Di Ruscio, Silvia Mazzini, Pierluigi Pierini, and Alfonso Pierantonio. *"Low-code engineering for internet of things: a state of research"*. In Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS '20). Association for Computing Machinery, New York, NY, USA, Article 74, 1–8. November 2020 https://doi.org/10.1145/3417990.3420208 **This paper was presented in Chapter 4**

9. **Felicien Ihirwe** , Davide Di Ruscio, Silvia Mazzini, and Alfonso Pierantonio. *"Towards a modeling and analysis environment for industrial IoT systems"*. In the International Workshop on MDE for Smart IoT Systems co-located with Software Technologies: Applications and Foundations (MESS@STAF21) conferences. June 2021. Bergen, Norway. Available https://ceur-ws.org/Vol-2999/messpaper1.pdf **This paper presents the preliminary results of the approach presented in Chapter 2**

10. **Felicien Ihirwe** , Arsene Indamutsa, Davide Di Ruscio, Silvia Mazzini, and Alfonso Pierantonio, *"Cloud-based modeling in IoT domain: a survey, open challenges, and opportunities"*. In 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C 2021), Fukuoka, Japan, 2021, pp. 73-82, doi:http://doi.org/10.1109/MODELS-C53483.2021.00018. **This paper was presented in Chapter 4**

9.2.4 Technical Reports

11. Léa Brunschwig, **Felicien Ihirwe**, Panagiotis Kourouklides, Joost Noppen *"D3.2. Lowcomotive Integrations - Interim Version"*. Technical report in the context of Training the Next Generation of Experts in Scalable Low-Code Engineering Platforms (Lowcomote). November 30th, 2020. The report available at:
https://cordis.europa.eu/project/id/813884/results

9.3 Developed tools

The full code and the instruction on how to use CHESSIoT can be found in detail in the appendix C

9.4 Future Directions

1. **Low-Code Engineering platforms capabilities on tackling complex IoT system:** While MDE is often referred to as an essential building block of low-code but with certain differences [58], however, there is still a question of which if the current generation of domain-agnostic LCDPs will keep up with the growing complexity of IoT stems. In the future, we will keep exploring quantitatively as well as qualitatively to which extent IoT low-code approaches are keeping up with the rise. We will keep exploring the limitations which such platforms are facing in terms of providing more engineering support, such as early analyses and verification, deployment, and continuous maintenance of developed software. In addition to that, we plan to explore the extent to which the generated solutions are to tackles complex tasks in comparison to the existing code-centric approaches.

2. **Quality in use product model for IoT engineering platforms:** The proposed software quality model was based on ISO/IEC 25010:2011 quality standard [14]. While the standard is composed of two main sets of quality models, namely *"quality in use model"*, and *"product quality model"*, the presented model only extends the software product model. In order to enhance the proposed quality evaluation process, we plan to also examine the possibility of evaluating *"quality in the use"* aspects for the IoT engineering platform. We plan on extending the *quality in use model* of the as well as accommodate other quality aspects beyond the software product quality model. This *"quality in the use"* model includes five characteristics, namely *effectiveness, efficiency, satisfaction, risk freedom, and context coverage* where some of them are further subdivided into nine sub-characteristics, however in "real-world" also other quality aspects matter. For example, how about the viability of the community (e.g., support and answer rate) or other aspects not covered by the model?

3. **Enhanced Fault-tree qualitative safety analysis:** The goal of the presented Fault-tree qualitative analysis approach is to provide a new representation of the existing FT that only includes the essential event representations. Although the current implementation does not fully reflect the final shape of the calculation of the minimal cut-set event sets [230], it does provide a much shorter and more correct, shorter, and readable FT that still helps and reflects the goal for the analysis. Due to the time constraint and the complexity that lies behind solving the logic function of the fault-tree model function, we did not manage to implement the probability logic solver infrastructure. In the future, we plan to implement the infrastructure for deriving minimal tree representation based on minimum cut-set events.In addition to that, we plan to integrate time-based failure logic analysis as well as the severity aspects into our approach. This is mainly to reflect the effect of which a component failure my cause on the entire system taking into account short or longer periods as well as how severe it could be. Finally, we intend to improve our system failure mode abstraction method by making it easily customizable from one domain to another as well as providing testing support to potentially assist in the recommendation of any potentially missing safety rules.

4. **Improved code generator for supporting other platforms:** The presented software development approach provides means for modeling the functional and behavioral model of the systems, and later it is transformed into a ThingML model ready to be compiled into platform-specific code. Although ThingML supports many different platform-specific languages, the provided transformation infrastructure currently supports the ThingML models, which can be compiled into Arduino platforms. In the future, we want to expand such infrastructure e.g., by including other platforms running on Java and C++.

5. **CHESSIoT services on the cloud**: With the growing interest in cloud-based engineering tools, traditional local-based solutions appear to be increasingly driven towards becoming cloud-based. Although CHESSIoT is not a cloud-based tool, it provides significant support for more

complex and critical IoT system engineering tasks. Despite this, the lack of cloud-based support causes challenges with installability, dependencies, and, in some cases, usability. To support future accessibility as well as supporting our proposed LCEP concepts presented in 2.3.2, we intend to detach all supported engineering features and deploy them separately, allowing such facilities to be consumed via dedicated API. This will also allow re-engineering of the modeling infrastructure to the cloud.

6. **Extensive empirical validation**: As CHESSIoT was developed as a tool to support the engineering of IoT systems which involves capabilities that needs to abide with certain standards and approvals from the external experts, in the future we plan to conduct an extensive empirical evaluation of the approach to check the acceptance of the proposed method by various external stakeholders in the industry. For instance, for what regards the safety analysis infrastructure, conducting this can in turns give us possible recommendation on what to follow when proposing potential fundamental principles on dealing with safety analysis for IoT systems an an evolving domain.

Appendix A

Software product quality evaluation questionnaire for IoT LCDP and MDE

Characteristic	Sub-characteristic	Questions
Functional suitability	Functional Completeness	"Does the platform support any form of modeling of the Edge layer? Does the platform support any form of modeling of the Fog layer? Does the platform support any form of modeling of the Cloud layer? Does the supporting paper mention any support for dealing with different communication protocols?"
	Functional Correctness	"Does the platform employs at least one of correctness methodologies during the development? For instance: - Component based development - Correct by construction - Model-checking - Model validation - Rule-based development - Model verification"
	Functional Appropriateness	Does correctness approaches impact in the platform code generation process?
Performance efficiency	Time-behavior	"Does the platform respond well while in usage? If it is not lively accessible, does the supporting paper mention any means for execution time efficiency?"
	Resource Utilization	"Does the platform require zero or very minimal set-up in usage? If not accessible live, does the supporting paper mention any means for dealing with resources needed?"
	Capacity	"Can the platform be used to develop apps from various IoT sub-domains? Can the platform develop a scalable IoT platform with hundreds of connected devices?"
Compatibility	Co-existence	"Can the platform be deployed and used in a shared server environment? If the platform is not lively accessible, does the supporting paper mention any mean for that?"
	Interoperability	"Does the platform support any means for exchanging data from third parties services? Does the developed application support external data exchanges?"
Reliability	Maturity	"Does the platform support both the design and code generation of engineering IoT systems? Does the platform support any kind of deployment of engineering IoT systems? Does the platform support any kind of analysis?"
	Availability	"Does the platform operational when needed? In case of a local based platform, is it downloadable and directly be used when needed?"
	Fault tolerance	"Does the platform provide any mean for fault tolerance such as self-adaptation or self-healing operation mechanisms? Does the supporting paper mention any mean for such support?"
	Recoverability	"Does the platform provide any mean for fault tolerance such as self-recovery or self-redeployment operation mechanisms? Does the supporting paper mention any mean for such support?"
Usability	Appropriateness	"Does the name of the platform reveals easily any IoT related concepts? Is the platform solery designed for IoT systems development Does the platform's concepts, elements and constructs used in development IoT specific?"
	Learnability	Does the platform provides any kind of the following supports: context-based modeling, on-the-fly suggestions, and so on
	Operability	Does the platform provide any kind of the following supports: as auto-completion for textual languages, guide-through mechanisms, multi-view modeling, palette show/hide, and palette element search?
	User error protection	Does the platform provide any means for error protection such as static analysis or on-the-fly error handling
	User interface	Does the platform provides pleasant and satisfying user interfaces?
	Accessibility	"Does the platform easily reachable in case of needs, either being locally or online? Does the supporting paper mention any mean for such support (open accessed repo for local based platforms)?"

Security	Confidentiality	"Does the platform provides any means for granting access only to authorized parties? Does the supporting paper mention any mean for such support?"
	Integrity	"Does the platform provide any means to prohibits unwanted access, modification of the platform, or data? Does the supporting paper mention any mean for such support?"
	Non-repudiation	"Does the platform enforce the logging of all hortorical activities performed during the developemnt process? Can such activities be retrieved later?"
	Accountability	"Does the platform provide any kind of sofware under development versioning? Does the supporting paper mention any mean for such support?"
	Authenticity	"Does the platform support any kind of authentication mechanisms while accessing platform resources? Does the supporting paper mention any mean for such support?"
Maintainability	Modularity	Does the platform be decoupled into differeent sub-parts for instance, micro-services?
	Reusability	"Does the platform sub-parts be reused independently? Does the supporting paper mention any mean for such support?"
	Analyzability	"Does the platform sub-parts be analysed independently? Does the supporting paper mention any mean for such support?"
	Modifiability	"Does the platform sub-parts be modified independently without introducing any flaws or deteriorating the quality of existing products? Does the supporting paper mention any mean for such support?"
	Testability	"Does the platform sub-parts be tested independently? Does the supporting paper mention any mean for such support?"
Portability	Adaptability	Can the platform be deployed effectively and efficiently adapts to different environments
	Installability	Can the platform be successfully installed and/or uninstalled in a given environment?
	Replaceability	Can the pltform be updated, replaced, and redeployed in the same environment and still performs as expected.

Appendix B

Fault-Tree generation

B.1 FLA2FT transformation rules

```
/*******************************************************************************
* Copyright (c) 2022, Intecs Solutions SpA
*
* All rights reserved. This program and the accompanying materials
* are made available under the terms of the Eclipse Public License v2.0
* which accompanies this distribution, and is available at
* https://www.eclipse.org/org/documents/epl-2.0/EPL-2.0.html
*
* SPDX-License-Identifier: EPL-2.0
*
*   Contributors:
*   Felicien Ihirwe
*   Initial API and implementation and/or initial documentation
*******************************************************************************/
pre{
  "Running transformation-------------------------".println();
  var mapping : Sequence ;
  var ftas : Sequence ;
  var parents : Sequence;
  var events : Sequence;
  var portNames : Sequence;
  var count : Integer;
  count=0;
  mapping = flamm!Port.allInstances();
  "Started the transformation-------------------------".println();
  for(port in mapping){
    if(port.owner.parent.isUndefined()and port.owner.outputPorts.contains(port)){
      for(f in port.failures){
        if(f.id.contains("noFailure") = false){
          var fta = new emfta!FTAModel;
          fta.name ="Fault Tree of "+ port.name+"+"+f.id+" \n--------------\n TOP FAILURE
                <"+f.id+"> at "+port.name+" port in "+port.owner.name;
          ftas.add(fta);
        }
      }
    }
  }
}

rule outputOfComposite2FTA
transform c : flamm!CompositeComponent
to ev : emfta!FTAModel{
  guard : c.parent.isUndefined()
  for(p in mapping){
    if(p.owner.parent.isUndefined()and p.owner.outputPorts.contains(p)){
      for(f in p.failures){
        for(ft in ftas){
          if(ft.name = "Fault Tree of "+p.name+"+"+f.id+" \n--------------\n TOP FAILURE
                  <"+f.id+"> at "+p.name+" port in "+p.owner.name){
            var e= new emfta!Event;
            e.name = p.name+"+"+f.id+" \n--------------\n TOP FAILURE <"+f.id+"> at
                "+p.name+" port in "+p.owner.name;
```

```
            e.type = emfta!EventType#Intermediate;
            var gate : new emfta!Gate;
            e.gate = gate;
            e.gate.type = emfta!GateType#OR;
            e.description=""+ftas.getCount()+"_"+f.id;
            e.gate.description=""+e.description;
            ft.events.add(e);
            for(con in p.connectedPorts){
              parents.clear();
              con.recurseFaultTree(f,con,ft,e);
            }
          }
          //take care of undeveloped events to avoid concurrent modification error
          var k1=0;
          while(k1<ft.events.size()){
            var event= ft.events.at(k1);
            if(event.type.name="Intermediate"){
              if(event.gate.events.size()<1){
                var eventSp : new emfta!Event;
                eventSp.name= "UNDEVELOPED FAILURE";
                eventSp.type = emfta!EventType#Undevelopped;
                eventSp.description="unknown_undeveloped";
                ft.events.add(eventSp);
                event.gate.events.add(eventSp);
                parents.add(eventSp);
              }
              //take care of internal propagation and transformation gate differences
              if(event.gate.events.size()=1){
                for(v1 in event.gate.events){
                  if(event.description.split("_").get(1)=v1.description.split("_").get(1)){
                    event.gate.type = emfta!GateType#OR;
                  }
                }
              }
            }k1=k1+1;
          }
        }
      }
    }
  }
  // remove the undeveloped tree
  var k=0;
  while(k<ftas.size()){
    var ft= ftas.at(k);
    if(ft.events.size()=1){
      ftas.remove(ft);
    }k=k+1;
  }
  "Ended the transformation-------------------------".println();
}
operation flamm!Port recurseFaultTree(f : flamm!Failure, con : flamm!Port,
          ft : emfta!FTAModel,e : emfta!Event) : Sequence{
  if(con.owner.parent.isDefined()){
    if(con.owner.type.name="SimpleComponent"){
      for(rul in con.owner.rules){
        for(outexp in rul.outputExpression){
          if(outexp.port = con)
          {
            for(f1 in outexp.failures){
              if(f1.id=f.id){
                var eventSp : new emfta!Event;
                eventSp.name= outexp.port.name+"+"+f1.id+" \n--------------\n
                        LOCAL FAILURE <"+f1.id+"> at "+outexp.port.name+"
                          port in "+ outexp.port.owner.name;
                eventSp.type = emfta!EventType#Intermediate;
                var gate : new emfta!Gate;
                eventSp.gate = gate;
                eventSp.gate.type = emfta!GateType#AND;
                eventSp.description=""+ftas.getCount()+"_"+f1.id;
                eventSp.gate.description=""+eventSp.description;
                ft.events.add(eventSp);
                e.gate.events.add(eventSp);
                parents.add(eventSp);
                for(inpexp in rul.inputExpression){
```

```
                for(f2 in inpexp.failures){
                  if(f2.id.contains("wildcard")=false){
                    if(f2.id.contains("noFailure")=true){
                      if(rul.inputExpression.size()=1){
                        var ev : new emfta!Event;
                        ev.name =outexp.port.name+"+"+f1.id+" \n--------------\n
                              INTERNAL FAILURE <"+f1.id+"> at "+outexp.port.name+"
                                port in "+ outexp.port.owner.name;
                        ev.type = emfta!EventType#Basic;
                        ev.description=""+ftas.getCount()+"_"+f1.id;
                        eventSp.gate.events.add(ev);
                        ft.events.add(ev);
                        parents.add(ev);
                      }
                      else{
                        //check if all input failures are noFailures
                        var tempF=true;
                        for(inpexp2 in rul.inputExpression){
                          for(f3 in inpexp2.failures){
                            if(f3.id.contains("noFailure")=false and
                                f3.id.contains("wildcard")=false){
                              tempF=false;
                              break;
                            }
                          }
                        }
                        // add a internal failure only if no one was added before
                        if(tempF=true and eventSp.gate.events.size()=0){
                          var ev : new emfta!Event;
                          ev.name =outexp.port.name+"+"+f1.id+" \n--------------\n
                                INTERNAL FAILURE <"+f1.id+">  at "+outexp.port.name+"
                                  port in "+ outexp.port.owner.name;
                          ev.type = emfta!EventType#Basic;
                          ev.description=""+ftas.getCount()+"_"+f1.id;
                          eventSp.gate.events.add(ev);
                          ft.events.add(ev);
                          parents.add(ev);
                        }
                      }

                    }

                    else{
                      var ev : new emfta!Event;
                      ev.name = inpexp.port.name+"+"+f2.id+" \n--------------\n
                            LOCAL FAILURE <"+f2.id+"> at "+inpexp.port.name+"
                              port in "+ inpexp.port.owner.name;
                      ev.type = emfta!EventType#Intermediate;
                      var gate : new emfta!Gate;
                      ev.gate = gate;
                      ev.gate.type = emfta!GateType#OR;
                      ev.description=""+ftas.getCount()+"_"+f2.id;
                      ev.gate.description=""+ev.description;
                      eventSp.gate.events.add(ev);
                      ft.events.add(ev);
                      parents.add(ev);
                      for(neWcon in inpexp.port.connectedPorts){
                        con.recurseFaultTree(f2,neWcon,ft,ev);
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
    else{
      if(con.owner.outputPorts.contains(con)){
        for(p in con.connectedPorts){
          if(not p.owner.inputPorts.contains(p) and not (p.owner=con.owner.parent)){
            p.recurseFaultTree(f,p,ft,e);
```

```
          }
        }
      }
      else if(con.owner.inputPorts.contains(con)){
        for(p in con.connectedPorts){
          if(p.owner.parent=con.owner and p.owner.inputPorts.contains(p)){
          }
          else{
            p.recurseFaultTree(f,p,ft,e);
          }
        }
      }
    }
  }
  else{
    var ev : new emfta!Event;
    ev.name =con.name+"+"+f.id+" \n--------------\n INJECTED FAILURE <"+f.id+">
        at "+ con.name+" port in "+con.owner.name;
    ev.type = emfta!EventType#External;
    ev.description=""+ftas.getCount()+"_"+f.id;
    e.gate.events.add(ev);
    ft.events.add(ev);
    parents.add(e);
  }
  return parents;
}
operation Any getCount(): Integer{
  count=count+1;
  return count;
}
```

Listing B.1: FLA2FT ETL transformation rules

B.2 FT2FT transformation:Qualitative and quantitative analysis

```
/*********************************************************************************
* Copyright (c) 2022, Intecs Solutions Spa
*
* All rights reserved. This program and the accompanying materials
* are made available under the terms of the Eclipse Public License v2.0
* which accompanies this distribution, and is available at
* https://www.eclipse.org/org/documents/epl-2.0/EPL-2.0.html
*
* SPDX-License-Identifier: EPL-2.0
*
*   Contributors:
*   Felicien Ihirwe
*   Initial API and implementation and/or initial documentation
*********************************************************************************/


pre{
  "Running Analysis -------------------------".println();
  var mapping : Sequence ;
  var ftas : Sequence ;
  var parents : Sequence;
  var events : Sequence;
  var count : Integer;
  count=0;
  mapping = emftanew!FTAModel.allInstances();
  for(ft in mapping){
    if(ft.events.size()>1){
      var fta = new emfta!FTAModel;
      fta.name = ft.name;
      ftas.add(fta);
    }
  }
}
rule topEvent2FTA
transform ftSource : emftanew!FTAModel
to evTarget : emfta!Event{
  var firstelement : Boolean = true;
```

```
38   for(ftTarget in ftas){
39     if(firstelement = true){
40       for(evSource in ftSource.events){
41         if(ftTarget.name = "Fault Tree of "+evSource.name){
42           var nevEv= new emfta!Event;
43           nevEv.name=evSource.name;
44           if(evSource.gate.events.size()>1){
45             nevEv= AssignNewType(evSource,nevEv);
46             nevEv.gate=getGate(evSource.description);
47             nevEv.probability=evSource.probability;
48             ftTarget.events.add(nevEv);
49             firstelement=false;
50             generateFollowings(evSource,nevEv,ftTarget);
51           }
52           else if(evSource.gate.events.size()=1){
53             getTheNextEvenWithOneEvent(evSource,nevEv,ftTarget,firstelement);
54           }
55         }
56       }
57     }
58     //Cleaning fault tree;
59     cleanTree(ftTarget);
60     //Calculating probabilities
61     for(event in ftTarget.events){
62       if("Fault Tree of "+event.name=ftTarget.name){
63         if (event.probability=0){
64           var eventProb=1;
65           //get the gate attached to it (in case it is a OR gate)
66           if(event.gate.type.name="OR"){
67             //check all following events and check if they are among the system's events
68             for(ev in event.gate.events){
69               //sometimes these attached event includes the original events, they are
     excluded
70               if(not(ev.name = event.name) and ftTarget.events.contains(ev)){
71                 if(ev.type.name="Intermediate"){
72                   var p=getProbability(ev,event.gate,ftTarget).asDouble();
73                   eventProb=eventProb*(1-p);
74                 }
75                 else{
76                   eventProb=eventProb*(1-ev.probability);
77                 }
78               }
79             }
80             event.probability=(1-eventProb).asDouble();
81           }
82           else if(event.gate.type.name="AND"){
83             for(ev in event.gate.events){
84               if(not(ev.name = event.name) and ftTarget.events.contains(ev)){
85                 if(ev.type.name="Intermediate"){
86                   var p=getProbability(ev,event.gate,ftTarget).asDouble();
87                   eventProb=eventProb*p;
88                 }
89                 else{
90                   eventProb=eventProb*(ev.probability);
91                 }
92               }
93             }
94             event.probability=(eventProb).asDouble();
95           }
96         }
97       }
98     }
99     cleanGates(ftTarget);
100  }
101 }
102 operation getTheNextEvenWithOneEvent(evSource: emftanew!Event, nevEv : emfta!Event,
103                 ftTarget : emfta!FTAModel,firstelement : Boolean) : Sequence{
104   for(event in evSource.gate.events){
105     if(event.type.name="Basic"){
106       if(firstelement=true){
107         nevEv= AssignNewType(evSource,nevEv);
108         nevEv.gate=getGate(evSource.description);
109         nevEv.description=evSource.description;
110         nevEv.probability=evSource.probability;
```

```
111         ftTarget.events.add(nevEv);
112         firstelement=false;
113         var localEv= new emfta!Event;
114         localEv.name=event.name;
115         localEv.type= emfta!EventType#Basic;
116         localEv.description=event.description;
117         localEv.probability=event.probability;
118         nevEv.gate.events.add(localEv);
119         ftTarget.events.add(localEv);
120         parents.add(nevEv);
121       }
122       else{
123         firstelement=false;
124         nevEv.name=event.name;
125         nevEv.type= emfta!EventType#Basic;
126         nevEv.description=event.description;
127         nevEv.probability=event.probability;
128         ftTarget.events.add(nevEv);
129         parents.add(nevEv);
130       }
131     }
132     else if(event.type.name="External"){
133       if(firstelement=true){
134         nevEv= AssignNewType(evSource,nevEv);
135         nevEv.gate=getGate(evSource.description);
136         nevEv.description=evSource.description;
137         nevEv.probability=evSource.probability;
138         ftTarget.events.add(nevEv);
139         firstelement=false;
140         var localEv= new emfta!Event;
141         localEv.name=event.name;
142         localEv.type= emfta!EventType#External;
143         localEv.description=event.description;
144         localEv.probability=event.probability;
145         nevEv.gate.events.add(localEv);
146         ftTarget.events.add(localEv);
147         parents.add(nevEv);
148       }
149       else{
150         firstelement=false;
151         nevEv.name=event.name;
152         nevEv.type= emfta!EventType#External;
153         nevEv.description=event.description;
154         nevEv.probability=event.probability;
155         ftTarget.events.add(nevEv);
156         parents.add(nevEv);
157       }
158     }
159     else if(event.type.name="Intermediate"){
160       if(event.description.split("_").get(1)=evSource.description.split("_").get(1)){
161         if(event.gate.events.size()>1){
162
163           if(firstelement=false){
164             nevEv.name=evSource.name;
165           }
166           nevEv= AssignNewType(event,nevEv);
167           nevEv.gate=getGate(event.description);
168           nevEv.description=event.description;
169           nevEv.probability=event.probability;
170           ftTarget.events.add(nevEv);
171           firstelement=false;
172           generateFollowings(event,nevEv,ftTarget);
173         }
174         else if(event.gate.events.size()=1){
175           getTheNextEvenWithOneEvent(event,nevEv,ftTarget,firstelement);
176         }
177       }
178       else{
179         //in case the two single events are not the same
180         //implement the source and check the next event properties
181         nevEv= AssignNewType(evSource,nevEv);
182         nevEv.gate=getGate(evSource.description);
183         nevEv.description=evSource.description;
184         nevEv.probability=evSource.probability;
```

```
185         ftTarget.events.add(nevEv);
186         firstelement=false;
187         if(event.gate.events.size()>1){
188           // if greater than one and different form source.. implement that
189           var localEv= new emfta!Event;
190           localEv.name=event.name;
191           localEv= AssignNewType(event,localEv);
192           localEv.gate=getGate(event.description);
193           localEv.description=event.description;
194           localEv.probability=event.probability;
195           nevEv.gate.events.add(localEv);
196           ftTarget.events.add(localEv);
197           generateFollowings(event,localEv,ftTarget);
198         }
199         else if(event.gate.events.size()=1){
200           // else implement half and look the next different of many child event
201           var localEv= new emfta!Event;
202           localEv.name=event.name;
203           nevEv.gate.events.add(localEv);
204           getTheNextEvenWithOneEvent(event,localEv,ftTarget,firstelement);
205         }
206       }
207     }
208     else if(event.type.name="Undevelopped"){
209       if(firstelement=true){
210         nevEv= AssignNewType(evSource,nevEv);
211         nevEv.gate=getGate(evSource.description);
212         nevEv.description=evSource.description;
213         nevEv.probability=evSource.probability;
214         ftTarget.events.add(nevEv);
215         firstelement=false;
216         var localEv= new emfta!Event;
217         localEv.name=event.name;
218         localEv.type= emfta!EventType#Undevelopped;
219         localEv.description=event.description;
220         localEv.probability=event.probability;
221         nevEv.gate.events.add(localEv);
222         ftTarget.events.add(localEv);
223         parents.add(nevEv);
224       }
225       else{
226         firstelement=false;
227         nevEv.name=event.name;
228         nevEv.type= emfta!EventType#Undevelopped;
229         nevEv.description=event.description;
230         nevEv.probability=event.probability;
231         ftTarget.events.add(nevEv);
232         parents.add(nevEv);
233       }
234     }
235   }
236 }
237
238 operation generateFollowings(lookEv: emftanew!Event, grobalEv : emfta!Event,
239                 ft : emfta!FTAModel) : Sequence{
240   for(event in lookEv.gate.events){
241     if(event.gate.events.size()>1){
242       var localEv= new emfta!Event;
243       localEv.name=event.name;
244       localEv= AssignNewType(event,localEv);
245       localEv.gate=getGate(event.description);
246       localEv.description=event.description;
247       localEv.probability=event.probability;
248       grobalEv.gate.events.add(localEv);
249       ft.events.add(localEv);
250       generateFollowings(event,localEv,ft);
251     }
252     else if(event.gate.events.size()=1){
253       var localEv= new emfta!Event;
254       localEv.name=event.name;
255       grobalEv.gate.events.add(localEv);
256       getTheNextEvenWithOneEvent(event,localEv,ft,false);
257     }
258   }
```

```
}
operation getGate(st : String): emfta!Gate{
  var g=new emfta!Gate;
  for (gate in emftanew!Gate.allInstances()){
    if(gate.description = st){
      if(gate.type.name="OR"){
        g.type=emfta!GateType#OR;
      }
      else{
        g.type=emfta!GateType#AND;
      }
      g.description=gate.description;
      for(oldEv in gate.events){
        var evTemp=new emfta!Event;
        evTemp.name=oldEv.name;
        evTemp=AssignNewType(oldEv,evTemp);
        evTemp.description=oldEv.description;
        g.events.add(evTemp);
      }
      g.nbOccurrences=gate.nbOccurrences;
      return g;
    }
  }
}

operation AssignNewType(oldEv: emftanew!Event,evTemp: emfta!Event):emfta!Event{
  if(oldEv.type.name="External"){
    evTemp.type=emfta!EventType#External;
  }
  else if(oldEv.type.name="Basic") {
    evTemp.type=emfta!EventType#Basic;
  }
  else {
    evTemp.type=emfta!EventType#Intermediate;
  }
  return evTemp;
}
operation getProbability(lookEv: emfta!Event, gate : emfta!Gate,
              ftTarget:emfta!FTAModel) : Any{
  var eProb=1;
  //Same process but here we need to return the lookEV probability
  if(lookEv.gate.type.name="OR"){
    for(ev in lookEv.gate.events){
      if(not(ev.name = lookEv.name) and ftTarget.events.contains(ev)){

        if(ev.type.name="Intermediate"){
          var p=getProbability(ev,lookEv.gate,ftTarget).asDouble();
          eProb=eProb*(1-p);
        }
        else{
          eProb=eProb*(1-ev.probability);
        }
      }
    }
    lookEv.probability=(1-eProb).asDouble();
    return lookEv.probability;
  }
  else if(lookEv.gate.type.name="AND"){
    for(ev in lookEv.gate.events){
      if(not(ev.name = lookEv.name) and ftTarget.events.contains(ev)){

        if(ev.type.name="Intermediate"){
          var p=getProbability(ev,lookEv.gate,ftTarget).asDouble();
          eProb=eProb*p;
        }
        else{
          eProb=eProb*(ev.probability);
        }
      }
    }
    lookEv.probability=(eProb).asDouble();
    return lookEv.probability;
  }
  return lookEv.probability;
```

```
}
operation cleanTree(ftTarget : emfta!FTAModel) : Any{
  var k=0;
  var gates= emfta!Gate.allInstances();
  var toRemove : emfta!Event;
  while(k<gates.size()){
    var gate= gates.at(k);
    var i=0;
    var eventsLocal=gate.events;
    while(i<eventsLocal.size()){
      if(not (eventsLocal.at(i).type.name="Intermediate")){
        var name=eventsLocal.at(i).name;
        if(i+1<eventsLocal.size()){
          if(name= eventsLocal.at(i+1).name){
            //remove duplicated paths
            toRemove= eventsLocal.at(i);
            ftTarget.events.remove(toRemove);
            eventsLocal.remove(eventsLocal.at(i));
            emfta!Event.allInstances().remove(eventsLocal.at(i));
            parents.remove(eventsLocal.at(i));
          }
        }
      }i=i+1;
    }k=k+1;
  }
}
operation cleanGates(ftTarget : emfta!FTAModel) : Any{
  var k=0;
  var gates= emfta!Gate.allInstances();
  var toRemove : emfta!Event;
  while(k<gates.size()){
    var gate= gates.at(k);
    if(gate.events.size()<4){
      gate.type=emfta!GateType#INTERMEDIATE;
      gate.description="propagations and/or \n transformations";
    }k=k+1;
  }
}
```

Listing B.2: FT2FT ETL transformation rules

Appendix C

Installing CHESSIoT extension on top of CHESS

The infrastructure that was implemented to test and validate our approach was implemented on top of CHESS1.0.0. Although a newer version of CHESS (1.1.0) was been recently released, we haven't yet tested with our implementations. Therefore the following steps only apply to the CHESS1.0.0 version. In this section We show you the steps to follow in order to successfully install the extension:

1. Prerequisite:
   - Jdk/Java 8 (Mandatory not 11 or above)
   - Git client (optional)
   - Window OS
2. Download CHESS1.0.0 for Window OS from CHESS1.0.0 main page and extract it
3. Clone the repository contains CHESSIoTFeatures

   `"git clone https://github.com/fihirwe/CHESSIoT-features.git"`
4. The CHESSIoT source code can be accessed through this CHESSIoTplugins git repo.
5. Launch ***CHESS.exe*** to start CHESS.
6. To install the CHESSIoT features proceed to Help-> Install New Software–> Add–> Local –> Browse your computer then pick the folder with all of the extracted content and hit Finish.
7. When loading is finished make sure to uncheck the *"Group items by category"* option and select *"CHESSIoT extension supported FTA, ThingML model generation and IoT deployment support"* features and Click next to continue
8. Click next for CHESSIoT updates on already existing installation
9. Accepts the license
10. This will take a few seconds or minutes to load and download all the dependencies.
11. After few moment remeber to **"Agree to install unsigned contents"**
12. When the process is finished, you will need to restart your CHESS tool.
13. **Voila! Now you are ready to explore CHESSIoT!!**

Bibliography

[1] Alexandru Serbanati, Carlo Maria Medaglia, and Ugo Biader Ceipidor. Building blocks of the internet of things: State of the art and beyond. *Deploying RFID-Challenges, Solutions, and Open Issues*, pages 351–366, 2011.

[2] Team CHESS, 2020. URL https://www.eclipse.org/chess/publis/CHESS_ToolsetGuide.pdf.

[3] Zulqarnain Haider, Barbara Gallina, and Enrique Zornoza Moreno. FLA2FT: Automatic generation of fault tree from ConcertoFLA results. In *2018 3rd International Conference on System Reliability and Safety (ICSRS)*, pages 176–181, 2018. doi: 10.1109/ICSRS.2018.8688825.

[4] Bashar Alshboul, Dorina C. Petriu, Bashar Alshboul, and Dorina C. Petriu. Automatic derivation of fault tree models from SysML models for safety analysis. *Journal of Software Engineering and Applications*, 11:204–222, 5 2018. ISSN 1945-3116. URL https://doi.org/10.4236/jsea.2018.115013.

[5] Rajesh Kannan Megalingam, Divya M. Kaimal, and Maneesha V. Ramesh. Efficient patient monitoring for multiple patients using wsn. In *2012 International Conference on Advances in Mobile Network, Communication and Its Applications*, pages 87–90, 2012. doi: 10.1109/MNCApps.2012.23.

[6] Petri Kettunen and Maarit Laanti. Future software organizations – agile goals and roles. *European Journal of Futures Research*, 5(1):16, 2017. ISSN 2195-2248. doi: 10.1007/s40309-017-0123-7. URL https://doi.org/10.1007/s40309-017-0123-7.

[7] Martin Bauer, Mathieu Boussard, Nicola Bui, Jourik De Loof, Carsten Magerkurth, Stefan Meissner, Andreas Nettsträter, Julinda Stefa, Matthias Thoma, and Joachim W. Walewski. *IoT Reference Architecture*, pages 163–211. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-40403-0. doi: 10.1007/978-3-642-40403-0_8. URL https://doi.org/10.1007/978-3-642-40403-0_8.

[8] Felicien Ihirwe, Davide Di Ruscio, Silvia Mazzini, Pierluigi Pierini, and Alfonso Pierantonio. Low-Code engineering for internet of things: A state of research. In *In the 23rd ACM/IEEE MODELS'20: Companion Proceedings*, MODELS '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381352. doi: 10.1145/3417990.3420208. URL https://doi.org/10.1145/3417990.3420208.

[9] Abel Gómez, Markel Iglesias-Urkia, Lorea Belategi, Xabier Mendialdua, and Jordi Cabot. Model-driven development of asynchronous message-driven architectures with AsyncAPI. *Software and Systems Modeling*, pages 1–29, 2021. URL https://doi.org/10.1007/s10270-021-00945-3.

[10] Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. A Low-Code Development Environment to Orchestrate Model Management Services. *Advances in Production Management Systems*, 2021.

[11] Massimo Tisi, Jean-Marie Mottu, Dimitrios S. Kolovos, Juan De Lara, Esther M Guerra, Davide Di Ruscio, Alfonso Pierantonio, and Manuel Wimmer. Lowcomote: Training the next generation of experts in scalable low-code engineering platforms. *1st Junior Researcher Community Event*, page 67–76, July 2019. URL https://hal.archives-ouvertes.fr/hal-02363416.

[12] Aymen J Salman, Mohammed Al-Jawad, and Wisam Al Tameemi. Domain-Specific Languages for IoT: Challenges and Opportunities. *IOP Conference Series: Materials Science and Engineering*, 1067(1):012133, 2021. ISSN 1757-8981. doi: 10.1088/1757-899x/1067/1/012133.

[13] F. Ihirwe, A. Indamutsa, D. Ruscio, S. Mazzini, and A. Pierantonio. Cloud-based modeling in IoT domain: a survey, open challenges and opportunities. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 73–82, Los Alamitos, CA, USA, oct 2021. IEEE Computer Society. doi: 10.1109/MODELS-C53483.2021.00018.

[14] ISO ISO. ISO/IEC 25010:2011, Systems and software engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. *ISO*, 34:2910, 2011. URL https://www.iso.org/standard/35733.html. Last accessed March 2022.

[15] John Hutchinson, Jon Whittle, and Mark Rouncefield. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming*, 89:144–161, 2014. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2013.03.017. URL http://www.sciencedirect.com/science/article/pii/S0167642313000786. Special issue on Success Stories in Model Driven Engineering.

[16] Alberto Debiasi, Felicien Ihirwe, Pierluigi Pierini, Silvia Mazzini, and Stefano Tonetta. Model-based analysis support for dependable complex systems in CHESS. In *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,*, pages 262–269. INSTICC, SciTePress, 2021. ISBN 978-989-758-487-9. doi: 10.5220/0010269702620269. URL https://doi.org/10.5220/0010269702620269.

[17] Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. MontiThings: Model-driven development and deployment of reliable IoT applications. *Journal of Systems and Software*, 183:111087, 2022. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2021.111087. URL https://doi.org/10.1016/j.jss.2021.111087.

[18] Azham Hussain and Emmanuel Mkpojiogu. An application of the ISO/IEC 25010 standard in the quality-in-use assessment of an online health awareness system. *Jurnal Teknologi*, 77:9–13, 11 2015. doi: 10.11113/jt.v77.6107. URL https://doi.org/10.11113/jt.v77.6107.

[19] Eddas Bertrand-Martinez, Phelipe Dias Feio, Vagner de Brito Nascimento, Fabio Kon, and Antônio Abelém. Classification and evaluation of IoT brokers: A methodology. *International Journal of Network Management*, 31(3), may 2021. ISSN 1099-1190. doi: 10.1002/nem.2115. URL https://doi.org/10.1002/nem.2115.

[20] Johan J.C. Tambotoh, Sani M. Isa, Ford Lumban Gaol, Benfano Soewito, and Harco Leslie Hendric Spits Warnars. Software quality model for internet of things governance. In *2016 International Conference on Data and Software Engineering (ICoDSE)*, pages 1–6, 2016. doi: 10.1109/ICODSE.2016.7936138. URL https://doi.org/10.1109/ICODSE.2016.7936138.

[21] Md. Saifur Rahman and Hassan Reza. Systematic mapping study of non-functional requirements in big data system. In *2020 IEEE International Conference on Electro Information Technology (EIT)*, pages 025–031, 2020. doi: 10.1109/EIT48999.2020.9208288. URL http://dx.doi.org/10.1109/EIT48999.2020.9208288.

[22] Julien Siebert, Lisa Jöckel, Jens Heidrich, Koji Nakamichi, Kyoko Ohashi, Isao Namba, Rieko Yamamoto, and Mikio Aoyama. Towards guidelines for assessing qualities of machine learning systems. In *13th International Conference on Quality of Information and Communications Technology - , QUATIC 2020, Faro, Portugal, September 9-11, 2020*, volume 1266, pages 17–31, 2020. doi: 10.1007/978-3-030-58793-2_2. URL https://doi.org/10.1007/978-3-030-58793-2_2.

[23] Almeida Martins Luana, Afonso Júnior Paulo, Pimenta Freire André, and Costa Heitor. *IET Software*, 14:572–581(9), December 2020. ISSN 1751-8806. URL https://doi.org/10.1049/iet-sen.2020.0037.

[24] Jhonatan Bernardes Boarim and Ana Regina Cavalcanti da Rocha. CRM systems quality evaluation. In *XX Brazilian Symposium on Software Quality*, SBQS '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450395533. doi: 10.1145/3493244.3493273. URL https://doi.org/10.1145/3493244.3493273.

[25] Janine Koepp, Miriam Viviane Baron, Paulo Ricardo Hernandes Martins, Cristine Brandenburg, and et al. The quality of mobile apps used for the identification of pressure ulcers in adults: Systematic survey and review of apps in app stores. *JMIR Mhealth Uhealth*, 8(6):e14266, Jun 2020. ISSN 2291-5222. doi: 10.2196/14266. URL https://doi.org/10.2196/14266.

[26] Lenin Erazo-Garzón, Priscila Cedillo, Gustavo Rossi, and José Moyano. A domain-specific language for modeling IoT system architectures that support monitoring. *IEEE Access*, 10: 61639–61665, 2022. doi: 10.1109/ACCESS.2022.3181166.

[27] Felicien Ihirwe, Davide Di Ruscio, Simone Gianfranceschi, and Alfonso Pierantonio. Assessing the quality of Low-Code and MDE platforms for engineering IoT systems. *22nd IEEE International Conference on Software Quality, Reliability, and Security (QRS22)*, 2022. doi: http://dx.doi.org/10.2139/ssrn.4267269.

[28] Alexander Power and Gerald Kotonya. Providing fault tolerance via complex event processing and machine learning for IoT systems. In *Proceedings of the 9th International Conference on the Internet of Things*, IoT 2019, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450372077. doi: 10.1145/3365871.3365872. URL https://doi.org/10.1145/3365871.3365872.

[29] Antero Taivalsaari and Tommi Mikkonen. A roadmap to the programmable world: Software challenges in the IoT era. *IEEE Software*, 34(1):72–80, 2017. doi: 10.1109/MS.2017.26. URL https://doi.org/10.1109/MS.2017.26.

[30] Akram Amin Abdellatif and Florian Holzapfel. Model based safety analysis (MBSA) tool for avionics systems evaluation. In *2020 AIAA/IEEE 39th Digital Avionics Systems Conference (DASC)*, pages 1–5, 2020. URL https://doi.org/10.1109/DASC50938.2020.9256578.

[31] A. Joshi, S.P. Miller, M. Whalen, and M.P.E. Heimdahl. A proposal for model-based safety analysis. In *24th Digital Avionics Systems Conference*, volume 2, pages 13 pp. Vol. 2–, 2005. URL https://doi.org/10.1109/DASC.2005.1563469.

[32] Gaëlle Girard, Ivan Baeriswyl, Jonathan James Hendriks, Roland Scherwey, Christian Müller, Philipp Hönig, and Rüdiger Lunde. Model based safety analysis using sysml with automatic generation of FTA and FMEA artifacts. In *Proceedings of the 30th European Safety and Reliability Conference and the 15th Probabilistic Safety Assessment and Management Conference (Esrel 2020 PSAM 15), 1-5 November 2020, Venice, Italy*, number CONFERENCE. 1-5 November 2020, 2020.

[33] Richard F Paige, Louis M Rose, Xiaocheng Ge, Dimitrios S Kolovos, and Phillip J Brooke. FPTC: Automated Safety Analysis for Domain-Specific Languages. In Michel R V Chaudron, editor, *Models in Software Engineering*, pages 229–242, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-01648-6.

[34] Federico Ciccozzi, Ivica Crnkovic, Davide Di Ruscio, Ivano Malavolta, Patrizio Pelliccione, and Romina Spalazzese. Model-driven engineering for mission-critical IoT systems. *IEEE Software*, 34(1):46–53, 2017. doi: 10.1109/MS.2017.1. URL https://doi.org/10.1109/MS.2017.1.

[35] Davide Di Ruscio, Romina Eramo, Alfonso Pierantonio, Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio. *Model Transformations*, pages 91–136. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-30982-3. doi: 10.1007/978-3-642-30982-3_4. URL https://doi.org/10.1007/978-3-642-30982-3_4.

[36] Federico Ciccozzi and Romina Spalazzese. MDE4IoT: Supporting the internet of things with model-driven engineering. In Costin Badica, Amal El Fallah Seghrouchni, Aurélie Beynier, David Camacho, Cédric Herpson, Koen Hindriks, and Paulo Novais, editors, *Intelligent Distributed Computing X*, pages 67–76, Cham, 2017. Springer International Publishing. ISBN 978-3-319-48829-5. doi: 10.1007/978-3-319-48829-5_7. URL http://dx.doi.org/10.1007/978-3-319-48829-5_7.

[37] Nicolas Harrand, Franck Fleurey, Brice Morin, and Knut Eilif Husa. 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*, MODELS '16, page 125–135, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343213. doi: 10.1145/2976767.2976812. URL https://doi.org/10.1145/2976767.2976812.

[38] Richard Nicholson, Timothy Ward, Derek Baum, Xu Tao, Davide Conzon, and Enrico Ferrera. Dynamic fog computing platform for event-driven deployment and orchestration of distributed internet of things applications. In *2019 Third World Conference on Smart Trends in Systems Security and Sustainablity (WorldS4)*, pages 239–246, 2019. doi: 10.1109/WorldS4.2019.8903975.

[39] Davide Conzon, Mohammad Rifat Ahmmad Rashid, Xu Tao, Angel Soriano, Richard Nicholson, and Enrico Ferrera. BRAIN-IoT: Model-based framework for dependable sensing and actuation in intelligent decentralized IoT systems. In *2019 4th International Conference on Computing, Communications and Security (ICCCS)*, pages 1–8, 2019. doi: 10.1109/CCCS.2019.8888136. URL http://dx.doi.org/10.1109/CCCS.2019.8888136.

[40] José A. Barriga, Pedro J. Clemente, Encarna Sosa-Sanchez, and Alvaro E. Prieto. SimulateIoT: Domain specific language to design, code generation and execute IoT simulation environments. *IEEE Access*, 9:92531–92552, 2021. doi: 10.1109/ACCESS.2021.3092528. URL http://dx.doi.org/10.1109/ACCESS.2021.3092528.

[41] Liudong Xing and Suprasad V. Amari. *Fault Tree Analysis*, pages 595–620. Springer London, London, 2008. ISBN 978-1-84800-131-2. doi: 10.1007/978-1-84800-131-2_38. URL https://doi.org/10.1007/978-1-84800-131-2_38.

[42] Felicien Ihirwe, Davide Di Ruscio, Silvia Mazzini, and Alfonso Pierantonio. Towards a modeling and analysis environment for industrial IoT systems. *STAF Workshops*, abs/2105.14136: 90–104, 2021. URL https://ceur-ws.org/Vol-2999/messpaper1.pdf.

[43] Felicien Ihirwe, Davide Di Ruscio, Silvia Mazzini, and Alfonso Pierantonio. A domain-specific modeling and analysis environment for complex IoT applications. *CoRR*, abs/2109.09244, 2021. URL https://arxiv.org/abs/2109.09244.

[44] Barbara Gallina, Muhammad Atif Javed, Faiz Ul Muram, and Sasikumar Punnekkat. A model-driven dependability analysis method for component-based architectures. In *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, pages 233–240, 2012. doi: 10.1109/SEAA.2012.35.

[45] Barbara Gallina, Edin Sefer, and Atle Refsdal. Towards safety risk assessment of socio-technical systems via failure logic analysis. In *ISSRE Workshops*, pages 287–292, 2014.

[46] A.S. Cheliyan and S.K. Bhattacharyya. Fuzzy fault tree analysis of oil and gas leakage in subsea production systems. *Journal of Ocean Engineering and Science*, 3(1):38–48, 2018. ISSN 2468-0133. doi: https://doi.org/10.1016/j.joes.2017.11.005. URL https://doi.org/10.1016/j.joes.2017.11.005.

[47] Stefan Markulik, Marek Šolc, Jozef Petrík, Michaela Balážiková, Peter Blaško, Juraj Kliment, and Martin Bezák. Application of FTA analysis for calculation of the probability of the failure of the pressure leaching process. *Applied Sciences*, 11(15), 2021. ISSN 2076-3417. doi: 10.3390/app11156731. URL https://www.mdpi.com/2076-3417/11/15/6731.

[48] Refaul Ferdous, Faisal Khan, Brian Veitch, and Paul R. Amyotte. Methodology for computer aided fuzzy fault tree analysis. *Process Safety and Environmental Protection*, 87(4):217–226, 2009. ISSN 0957-5820. doi: https://doi.org/10.1016/j.psep.2009.04.004. URL https://www.sciencedirect.com/science/article/pii/S0957582009000421.

[49] A. Bassi, M. Bauer, M. Fiedler, T. Kramp, R. van Kranenburg, S. Lange, S. Meissner, and Eds. *Enabling things to talk: Designing IoT solutions with the IoT Architectural Reference Model*. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-40402-3. doi: 10.1007/978-3-642-40403-0.

[50] Bruno Costa, Paulo F. Pires, and Flávia C. Delicato. Modeling IoT applications with SysML4IoT. In *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 157–164, 2016. doi: 10.1109/SEAA.2016.19.

[51] Ajay Krishna Muroor Nadumane. *Models and Verification for Composition and Reconfiguration of Web of Things Applications*. Theses, Université Grenoble Alpes [2020-....], December 2020. URL https://theses.hal.science/tel-03188299.

[52] Jonathan Ostroff, Susan Gerhart, Dan Craigen, Ted Ralston, Nancy G. Leveson, Jonathan Bowen, and Victoria Stavridou. *Real-Time and Safety-Critical Systems*, pages 359–528. Springer London, London, 1999. ISBN 978-1-4471-3431-2. doi: 10.1007/978-1-4471-3431-2_6. URL https://doi.org/10.1007/978-1-4471-3431-2_6.

[53] John C. Knight. Safety critical systems: Challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 547–550, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 158113472X. doi: 10.1145/581339.581406. URL https://doi.org/10.1145/581339.581406.

[54] Jonathan Bowen. The ethics of safety-critical systems. *Commun. ACM*, 43(4):91–97, apr 2000. ISSN 0001-0782. doi: 10.1145/332051.332078. URL https://doi.org/10.1145/332051.332078.

[55] Jaehyung An, Alexey Mikhaylov, and Keunwoo Kim. Machine learning approach in heterogeneous group of algorithms for transport safety-critical system. *Applied Sciences*, 10(8), 2020.

ISSN 2076-3417. doi: 10.3390/app10082670. URL https://www.mdpi.com/2076-3417/10/8/2670.

[56] W.R. Dunn. Designing safety-critical computer systems. *Computer*, 36(11):40–46, 2003. doi: 10.1109/MC.2003.1244533.

[57] Raj kamal Kaur, Babita Pandey, and Lalit Kumar Singh. Dependability analysis of safety critical systems: Issues and challenges. *Annals of Nuclear Energy*, 120:127–154, 2018. ISSN 0306-4549. doi: https://doi.org/10.1016/j.anucene.2018.05.027. URL https://www.sciencedirect.com/science/article/pii/S030645491730213X.

[58] Davide Di Ruscio, Dimitris Kolovos, Juan de Lara, Alfonso Pierantonio, Massimo Tisi, and Manuel Wimmer. Low-Code development and model-driven engineering: Two sides of the same coin? *Software and Systems Modeling*, 21:437–446, 2022. ISSN 1619-1374. doi: 10.1007/s10270-021-00970-2. URL https://doi.org/10.1007/s10270-021-00970-2.

[59] Luca Berardinelli, Alexandra Mazak, Oliver Alt, and Manuel Wimmer. *Model-Driven Systems Engineering: Principles and Application in the CPPS Domain*, pages 261–299. 05 2017. ISBN 978-3-319-56344-2.

[60] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. 2012. ISBN 9781608458820. doi: 10.2200/s00441ed1v01y201208swe001.

[61] Paulo F. Pires, Bruno Costa, Flávia C. Delicato, Wei Li, and Albert Y. Zomaya. Design and analysis of IoT applications: A model driven approach. *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing*, pages 392–399, 2016. doi: 10.1109/DASC-PICom-DataCom-CyberSciTec.2016.81. URL http://dx.doi.org/10.1109/DASC-PICom-DataCom-CyberSciTec.2016.81.

[62] A. Bucchiarone, F. Ciccozzi, L. Lambers, A. Pierantonio, M. Tichy, M. Tisi, A. Wortmann, and V. Zaytsev. What is the future of modeling? *IEEE Software*, 38(02):119–127, mar 2021. ISSN 1937-4194. doi: 10.1109/MS.2020.3041522.

[63] Jiwei Huang, Songyuan Li, Ying Chen, and Junliang Chen. Performance modeling and analysis for iot services. *International Journal of Web and Grid Services*, 14:146, 01 2018. doi: 10.1504/IJWGS.2018.090742.

[64] Antonio Bucchiarone, Jordi Cabot, Richard F. Paige, and Alfonso Pierantonio. Grand challenges in model-driven engineering: An analysis of the state of the research. *Softw. Syst. Model.*, 19(1):5–13, jan 2020. ISSN 1619-1366. doi: 10.1007/s10270-019-00773-6.

[65] Davide Di Ruscio, Mirco Franzago, Ivano Malavolta, and Henry Muccini. Envisioning the future of collaborative model-driven software engineering. *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017*, (May):219–221, 2017. doi: 10.1109/ICSE-C.2017.143.

[66] S. Chen, H. Xu, D. Liu, B. Hu, and H. Wang. A vision of iot: Applications, challenges, and opportunities with china perspective. *IEEE Internet of Things Journal*, 1(4):349–359, 2014.

[67] Ábel Hegedüs, Gábor Bergmann, Csaba Debreceni, Ákos Horváth, Péter Lunk, Ákos Menyhért, István Papp, Dániel Varró, Tomas Vileiniskis, and István Ráth. Incquery server for teamwork cloud: Scalable query evaluation over collaborative model repositories. *21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS-Companion 2018*, pages 27–31, 2018. doi: 10.1145/3270112.3270125.

[68] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M.Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. Technical report, University of Trento, Berlin, Heidelberg, 2002.

[69] Dang Tu Nguyen, Chengyu Song, Zhiyun Qian, Srikanth V. Krishnamurthy, Edward J. M. Colbert, and Patrick McDaniel. IotSan: Fortifying the safety of IoT systems. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, pages 191–203, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450360807. doi: 10.1145/3281411.3281440. URL https://doi.org/10.1145/3281411.3281440.

[70] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, dec 2005. ISSN 0360-0300. doi: 10.1145/1118890.1118892. URL https://doi.org/10.1145/1118890.1118892.

[71] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010. ISBN 0321712943. URL https://dl.acm.org/doi/10.5555/1809745.

[72] Henry Muccini and Mohammad Sharaf. CAPS: Architecture description of situational aware cyber physical systems. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 211–220, 2017. doi: 10.1109/ICSA.2017.21.

[73] Nicolas Ferry and et al. Development and operation of trustworthy smart iot systems: The ENACT framework. In *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pages 121–138, 2020. ISBN 978-3-030-39306-9.

[74] Imad Berrouyne, Mehdi Adda, Jean-Marie Mottu, Jean-Claude Royer, and Massimo Tisi. CyprIoT: framework for modeling and controlling network-based IoT applications. In *In 34th ACM/SIGAPP Symposium on Applied Computing*, pages 832–841, 2019.

[75] Atefeh Torkaman and M.A.Seyyedi. Analyzing IoT referencearchitecture models. *International Journal of Computer Science and Software Engineering (IJCSSE)*, 5, 2016. ISSN 2409-4285.

[76] ISO & ICE. Study report on IoT reference architectures/frameworks, "August 2014". URL http://docplayer.net/16351625-Study-report-on-iot-reference-architectures-frameworks.html. July 2020.

[77] Paul Fremantle. A reference architecture for the internet of things, 10 2015. URL https://wso2.com/wso2_resources/wso2_whiteapproach_a-reference-architecture-for-the-internet-of-things.pdf.

[78] ESA Requirements and Standards Division. *Space Product Assurance - Fault Tree Analysis - Adoption Notice ECSS/IEC 61025*. ESA Publ. Division, 2008. URL https://ecss.nl/standards/active-standards/.

[79] ESA Requirements and Standards Division. *Failure modes, effects (and criticality) analysis (FMEA/FMECA)*. ESA Publ. Division, 2009. URL https://ecss.nl/standards/active-standards/.

[80] Jacopo Parri, Samuele Sampietro, and Enrico Vicario. FaultFlow: a tool supporting an mde approach for timed failure logic analysis. In *2021 17th European Dependable Computing Conference (EDCC)*, pages 25–32, 2021. doi: 10.1109/EDCC53658.2021.00011.

[81] Malcolm Wallace. Modular architectural representation and analysis of fault propagation and transformation. *Electronic Notes in Theoretical Computer Science*, 141(3):53–71, 2005. ISSN 1571-0661. doi: https://doi.org/10.1016/j.entcs.2005.02.051. URL http://www.sciencedirect.com/science/article/pii/S1571066105051650. FESCA 2005.

[82] Leonardo Montecchi and Barbara Gallina. SafeConcert: A metamodel for a concerted safety modeling of socio-technical systems. In Marco Bozzano and Yiannis Papadopoulos, editors, *Model-Based Safety and Assessment*, pages 129–144, Cham, 2017. Springer International Publishing. ISBN 978-3-319-64119-5.

[83] Diomidis H Stamatis. *Failure mode and effect analysis: FMEA from theory to execution*. Quality Press, 2003.

[84] Xiangyu Han and Jun Zhang. A combined analysis method of FMEA and FTA for improving the safety analysis quality of safety-critical software. In *2013 IEEE International Conference on Granular Computing (GrC)*, pages 353–356, 2013. URL https://doi.org/10.1109/GrC.2013.6740435.

[85] Mohsen Marjani, Fariza Nasaruddin, Abdullah Gani, Ahmad Karim, Ibrahim Abaker Targio Hashem, Aisha Siddiqa, and Ibrar Yaqoob. Big IoT Data Analytics: Architecture, Opportunities, and Open Research Challenges. *IEEE Access*, 5:5247–5261, 2017. ISSN 21693536. doi: 10.1109/ACCESS.2017.2689040.

[86] Laith Farhan, Sinan T. Shukur, Ali E. Alissa, Mohmad Alrweg, Umar Raza, and Rupak Kharel. A survey on the challenges and opportunities of the Internet of Things (IoT). *Proceedings of the International Conference on Sensing Technology, ICST*, 2017-Decem(December):1–5, 2017. ISSN 21568073. doi: 10.1109/ICSensT.2017.8304465.

[87] Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. Supporting the understanding and comparison of low-code development platforms. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 171–178, 2020. doi: 10.1109/SEAA51224.2020.00036. URL https://doi.org/10.1109/SEAA51224.2020.00036.

[88] Jordi Cabot. Positioning of the low-code movement within the field of model-driven engineering. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381352. doi: 10.1145/3417990.3420210. URL https://doi.org/10.1145/3417990.3420210.

[89] Node-RED. Node-RED: Low-code programming for event-driven applications. https://nodered.org/, 2020. URL https://nodered.org/. Last accessed May 2020.

[90] AtmosphereIoT. Fast time to first data. https://atmosphereiot.com/, 2020. URL https://atmosphereiot.com/. Last accessed May 2020.

[91] Darnell Kenebrew. The difference between a software developer and a software engineer, Jan 2023. URL https://www.computerscience.org/resources/software-developer-vs-software-engineer/.

[92] RapidMiner Team. Amplify the impact of your people, expertise data, Oct 2022. URL https://rapidminer.com/.

[93] Shalin Hai-Jew. Running a 'deep learning'artificial neural network in rapidminer studio. *C2C Digital Magazine*, 1(10):17, 2019.

[94] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Kilian Thiel, and Bernd Wiswedel. Knime - the konstanz information miner: Version 2.0 and beyond. *SIGKDD Explor. Newsl.*, 11(1):26–31, nov 2009. ISSN 1931-0145. doi: 10.1145/1656274.1656280. URL https://doi.org/10.1145/1656274.1656280.

[95] Panagiotis Kourouklidis, Dimitris Kolovos, Joost Noppen, and Nicholas Matragkas. A model-driven engineering approach for monitoring machine learning models. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 160–164, 2021. doi: 10.1109/MODELS-C53483.2021.00028.

[96] Lissette Almonte, Esther Guerra, Iván Cantador, and Juan De Lara. *Building Recommenders for Modelling Languages with Droid*. Association for Computing Machinery, New York, NY, USA, 2023. ISBN 9781450394758. URL https://doi.org/10.1145/3551349.3559521.

[97] Lissette Almonte, Sara Pérez-Soler, Esther Guerra, Iván Cantador, and Juan de Lara. Automating the synthesis of recommender systems for modelling languages. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2021, page 22–35, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450391115. doi: 10.1145/3486608.3486905. URL https://doi.org/10.1145/3486608.3486905.

[98] Claudio Di Sipio, Juri Rocco, Davide Di Ruscio, and Phuong Nguyen. Lev4rec: A low-code environment to support the development of recommender systems, 04 2022.

[99] Antonio Cicchetti, Federico Ciccozzi, Silvia Mazzini, Stefano Puri, Marco Panunzio, Alessandro Zovi, and Tullio Vardanega. CHESS: a model-driven engineering tool environment for aiding the development of complex industrial systems. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 362–365, 2012. doi: 10.1145/2351676.2351748.

[100] W. Godard and Geoffrey Nelissen. Model-based design and schedulability analysis for avionic applications on multicore platforms. *Ada User Journal*, 37(3):157–163, 09 2016. ISSN 1381-6551.

[101] L. Bressan, A. L. de Oliveira, L. Montecchi, and B. Gallina. A systematic process for applying the CHESS methodology in the creation of certifiable evidence. In *EDCC*, pages 49–56, 2018.

[102] Lorenzo Pace, Mauro Pasquinelli, Diego Gerbaz, Joachim Fuchs, Valter Basso, S. Mazzini, Laura Baracchi, Stefano Puri, Marco Lassalle, and Juhani Viitaniemi. Model-based approach for the verification enhancement across the lifecycle of a space system. In *INCOSE CIISE2014*, 10 2014.

[103] Silvia Mazzini. The CONCERTO project: An open source methodology for designing, deploying, and operating reliable and safe cps systems. *Ada User Journal*, 36:264–267, December 2015.

[104] B. Gallina, E. Sefer, and A. Refsdal. Towards safety risk assessment of socio-technical systems via failure logic analysis. In *ISSRE Workshops*, pages 287–292, 2014.

[105] Mahmood Shafiee, Evenye Enjema, and Athanasios Kolios. An integrated FTA-FMEA model for risk analysis of engineering systems: A case study of subsea blowout preventers. *Applied Sciences*, 9(6):1192, Mar 2019. ISSN 2076-3417. doi: 10.3390/app9061192. URL http://dx.doi.org/10.3390/app9061192.

[106] Laura Baracchi Silvia Mazzini, John Favaro. A model-based approach across the IoT lifecycle for scalable and distributed smart applications. pages 149–154, 2015. doi: 10.1109/ITSC.2015.33.

[107] Marco Panunzio and Tullio Vardanega. A component-based process with separation of concerns for the development of embedded real-time software systems. *Journal of Systems and Software*, 96:105–121, 2014. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2014.05.076. URL http://www.sciencedirect.com/science/article/pii/S0164121214001381.

[108] A. Bondavalli, I. Mura, S. Chiaradonna, R. Filippini, S. Poli, and F. Sandrini. DEEM: a tool for the dependability modeling and evaluation of multiple phased systems. In *DSN 2000*, pages 231–236, 2000. doi: 10.1109/ICDSN.2000.857541.

[109] Leonardo Montecchi, Paolo Lollini, and Andrea Bondavalli. A reusable modular toolchain for automated dependability evaluation. In *VALUETOOLS*, pages 298–303. ICST, 2013. ISBN 9781936968480. doi: 10.4108/icst.valuetools.2013.254395. URL https://doi.org/10.4108/icst.valuetools.2013.254395.

[110] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv Symbolic Model Checker. In Armin Biere and Roderick Bloem, editors, *CAV*, pages 334–342, Cham, 2014. Springer. ISBN 978-3-319-08867-9. doi: 10.1007/978-3-319-08867-9_22. URL https://doi.org/10.1007/978-3-319-08867-9_22.

[111] Alessandro Cimatti, Michele Dorigatti, and Stefano Tonetta. OCRA: A tool for checking the refinement of temporal contracts. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 702–705, 2013. doi: 10.1109/ASE.2013.6693137.

[112] Marco Bozzano, Alessandro Cimatti, Marco Gario, David Jones, and Cristian Mattarei. Model-based safety assessment of a triple modular generator with xsap. *Formal Aspects of Computing*, 33, 04 2021. doi: 10.1007/s00165-021-00532-9.

[113] Marco Bozzano, Alessandro Cimatti, Cristian Mattarei, and Stefano Tonetta. Formal safety assessment via contract-based design. In *ATVA*, pages 81–97. Springer, 2014.

[114] Benjamin Bittner, Marco Bozzano, Roberto Cavada, Alessandro Cimatti, Marco Gario, Alberto Griggio, Cristian Mattarei, Andrea Micheli, and Gianni Zampedri. The xSAP Safety Analysis Platform. In Marsha Chechik and Jean-François Raskin, editors, *TACAS*, pages 533–539, Berlin, Heidelberg, 2016. Springer. ISBN 978-3-662-49674-9. doi: 10.1007/978-3-662-49674-9_31. URL https://doi.org/10.1007/978-3-662-49674-9_31.

[115] M. Bozzano and A. Villafiorita. Safety critical systems. In *Encyclopedia of Software Engineering*. CRC Press (Taylor & Francis Group), USA, 1st edition, 2013. ISBN 1439803315. doi: https://doi.org/10.5555/1951720.

[116] T. Courtney, S. Gaonkar, K. Keefe, E. W. D. Rozier, and W. H. Sanders. Möbius 2.3: An extensible tool for dependability, security, and performance evaluation of large and complex system models. In *DSN*, pages 353–358, 2009. doi: 10.1109/DSN.2009.5270318.

[117] P. Popov. Models of reliability of fault-tolerant software under cyber-attacks. In *ISSRE*, pages 228–239, 2017.

[118] Eclipse Foundation. Qvto, . URL https://wiki.eclipse.org/QVTo.

[119] Eclipse Foundation. Generate anything from any emf model, . URL https://www.eclipse.org/acceleo/.

[120] Stéphane Bonnet, Jean-Luc Voirin, Véronique Normand, and Daniel Exertier. Implementing the MBSE cultural change: Organization, coaching and lessons learned. *INCOSE International Symposium*, 25(1):508–523, 2015. doi: 10.1002/j.2334-5837.2015.00078.x.

[121] Marco Bozzano, Harold Bruintjes, Alessandro Cimatti, Joost-Pieter Katoen, Thomas Noll, and Stefano Tonetta. COMPASS 3.0. In *TACAS*, pages 379–385. Springer, 2019. doi: 10.1007/978-3-030-17462-0_25. URL https://doi.org/10.1007/978-3-030-17462-0_25.

[122] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. Wiley-IEEE Press, 2 edition, April 2015. ISBN 978-1-118-85912-4.

[123] Jia Ming Cao and Tao Wu. Multi-domain modeling simulation and application based on MapleSim. In *Mechatronics and Intelligent Materials III*, volume 706, pages 1894–1897. Trans Tech Publications Ltd, 7 2013. doi: 10.4028/www.scientific.net/AMR.706-708.1894.

[124] Mendix. Mendix: IoT application development with a low-code platform. https://www.mendix.com/building-iot-applications/, 2020. Last accessed May 2020.

[125] Paul Vincent, Kimihiko Iijima, Mark Driver, Jason Wong, and Yefim Natis. Magic quadrant for enterprise Low-Code application platforms. https://www.gartner.com/, 2019. Last accessed June 2020.

[126] Salesforce. Salesforce news, features and certifications. Online, 2023. URL https://www.salesforceben.com. Accessed February 2023.

[127] ThingWorx ThingWorx. ThingWorx: Industrial IoT software: IIot platform, Feb 2023. URL https://www.ptc.com/en/products/thingworx.

[128] Microsoft Corporation. Business application platform | microsoft power platform, Feb 2023. URL https://powerplatform.microsoft.com/en-us/.

[129] Amazon. Aws IoT core: Easily and securely connect devices to the cloud, Feb 2023. URL https://aws.amazon.com/iot-core/.

[130] IBM Watson IoT platform, Sep 2015. URL https://internetofthings.ibmcloud.com/.

[131] Simplifier. Simplifier: Enterprise apps made simple. https://www.simplifier.io/en/, 2020. Last accessed May 2020.

[132] Google Cloud. Cloud IoT core | google cloud. URL https://cloud.google.com/iot-core.

[133] A. Salihbegovic, T. Eterovic, E. Kaljic, and S. Ribic. Design of a domain specific language and ide for internet of things applications. In *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 996–1001, 2015. doi: 10.1109/MIPRO.2015.7160420.

[134] Fabrizio F. Borelli, Gabriela O. Biondi, and Carlos A. Kamienski. BIoTA: A buildout IoT application language. *IEEE Access*, 8:126443–126459, 2020. doi: 10.1109/ACCESS.2020.3003694.

[135] Jussi Kiljander, Janne Takalo-Mattila, Matti Etelapera, Juha-Pekka Soininen, and Kari Keinanen. Enabling end-users to configure smart environments. In *2011 IEEE/IPSJ International Symposium on Applications and the Internet*, pages 303–308, 2011. doi: 10.1109/SAINT.2011.58.

[136] Jagni Dasa Horta Bezerra and Cidcley Teixeira de Souza. A model-based approach to generate reactive and customizable user interfaces for the web of things. In *Proceedings of the 25th Brazillian Symposium on Multimedia and the Web*, WebMedia '19, page 57–60, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367639. doi: 10.1145/3323503.3360631.

[137] Flavio Corradini, Arianna Fedeli, Fabrizio Fornari, Andrea Polini, and Barbara Re. FloWare: An Approach for IoT Support and Application Development. In Adriano Augusto, Asif Gill, Selmin Nurcan, Iris Reinhartz-Berger, Rainer Schmidt, and Jelena Zdravkovic, editors, *Enterprise, Business-Process and Information Systems Modeling*, pages 350–365, Cham, 2021. Springer International Publishing. ISBN 978-3-030-79186-5.

[138] Guillermo Cueva-Fernandez, Jordán Pascual Espada, Vicente García-Díaz, Cristian González García, and Nestor Garcia-Fernandez. Vitruvius: An expert system for vehicle sensor tracking and managing application generation. *Journal of Network and Computer Applications*, 42:178–188, 2014. ISSN 1084-8045. doi: https://doi.org/10.1016/j.jnca.2014.02.013. URL https://doi.org/10.1016/j.jnca.2014.02.013.

[139] Cristian González García, B. Cristina Pelayo G-Bustelo, Jordán Pascual Espada, and Guillermo Cueva-Fernandez. MIDGAR: Generation of heterogeneous objects interconnecting applications. a domain specific language proposal for internet of things scenarios. *Computer Networks*, 64:143–158, may 2014. ISSN 1389-1286. doi: https://doi.org/10.1016/j.comnet.2014.02.010. URL https://doi.org/10.1016/j.comnet.2014.02.010.

[140] Wajid Rafique, Xuan Zhao, Shui Yu, Ibrar Yaqoob, Muhammad Imran, and Wanchun Dou. An application development framework for internet-of-things service orchestration. *IEEE Internet of Things Journal*, 7(5):4543–4556, 2020. doi: 10.1109/JIOT.2020.2971013. URL http://dx.doi.org/10.1109/JIOT.2020.2971013.

[141] Manfred Sneps-Sneppe and Dmitry Namiot. On web-based domain-specific language for internet of things. In *2015 7th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*, pages 287–292, 2015. doi: 10.1109/ICUMT.2015.7382444.

[142] Robert Kleinfeld, Stephan Steglich, Lukasz Radziwonowicz, and Charalampos Doukas. Glue.Things: A mashup platform for wiring the internet of things with the internet of services. In *Proceedings of the 5th International Workshop on Web of Things*, WoT '14, page 16–21, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330664. doi: 10.1145/2684432.2684436.

[143] Amir Taherkordi and Frank Eliassen. Scalable modeling of cloud-based IoT services for smart cities. In *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, pages 1–6, 2016. doi: 10.1109/PERCOMW.2016.7457098.

[144] Yannis Valsamakis and Anthony Savidis. Personal Applications in the Internet of Things Through Visual End-User Programming. In Claudia Linnhoff-Popien, Ralf Schneider, and Michael Zaddach, editors, *Digital Marketplaces Unleashed*, pages 809–821, Berlin, Heidelberg, 2018. Springer Berlin Heidelberg. ISBN 978-3-662-49275-8. doi: 10.1007/978-3-662-49275-8_71. URL https://doi.org/10.1007/978-3-662-49275-8_71.

[145] Yi Xu and Abdelsalam Helal. Scalable cloud–sensor architecture for the internet of things. *IEEE Internet of Things Journal*, 3(3):285–298, 2016. doi: 10.1109/JIOT.2015.2455555.

[146] Simon Mayer, Ruben Verborgh, Matthias Kovatsch, and Friedemann Mattern. Smart configuration of smart environments. *IEEE Transactions on Automation Science and Engineering*, 13 (3):1247–1255, 2016. doi: 10.1109/TASE.2016.2533321.

[147] Badr El Khalyly, Mouad Banane, Allae Erraissi, and Abdessamad Belangour. InteroEvery: Microservice based interoperable system. In *2020 International Conference on Decision Aid Sciences and Application (DASA)*, pages 320–325, 2020. doi: 10.1109/DASA51403.2020.9317159. URL http://dx.doi.org/10.1109/DASA51403.2020.9317159.

[148] M. Hussein, S. Li, and A. Radermacher. Model-driven development of adaptive IoT systems. In *2017 MODELS Satellite Event*, volume 2019, pages 17–23, Austin, United States, September 2017. CEUR-WS. doi: 10.1109/DASC-PICom-DataCom-CyberSciTec.2016.81. URL https://hal-cea.archives-ouvertes.fr/cea-01843007.

[149] Mahmoud Hussein, Shuai Li, and Ansgar Radermacher. Model-driven development of adaptive IoT systems. *4th International Workshop on Interplay of Model-Driven Engineering and Component-Based Software EngineeringAt: Austin, Texas, USA*, 2017. URL https://www.researchgate.net/publication/319328820.

[150] Kleanthis Thramboulidis and Foivos Christoulakis. UML4IoT-A UML-based approach to exploit IoT in cyber-physical manufacturing systems. *Comput. Ind.*, 82(C):259–272, oct 2016. ISSN 0166-3615. doi: 10.1016/j.compind.2016.05.010. URL https://doi.org/10.1016/j.compind.2016.05.010.

[151] Open Mobile Alliance. Lightweight machine to machine technical specification. *Technical Specification OMA-TS-LightweightM2M-V1*, 2013.

[152] Armin Moin, Stephan Rössler, and Stephan Günnemann. ThingML+: Augmenting model-driven software engineering for the internet of things with machine learning. *MDE4IoT - MODELS 2018*, 2018.

[153] Claudio M. de Farias and Italo C. Brito et al. COMFIT: A development environment for the internet of things. *Future Generation Computer Systems*, 75:128–144, 2017. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2016.06.031.

[154] Mohammad Sharaf, Mai Abusair, Rami Eleiwi, Yara Shana'a, Ithar Saleh, and Henry Muccini. Modeling and code generation framework for IoT. In Pau Fonseca i Casas, Maria-Ribera Sancho, and Edel Sherratt, editors, *System Analysis and Modeling. Languages, Methods, and Tools for Industry 4.0*, pages 99–115, Cham, 2019. Springer International Publishing. ISBN 978-3-030-30690-8. doi: https://doi.org/10.1007/978-3-030-30690-8_6.

[155] S. Dhouib, A. Cuccuru, F. Le Fèvre, S. Li, B. Maggi, I. Paez, A. Rademarcher, N. Rapin, J. Tatibouet, P. Tessier, S. Tucci, and S. Gerard. Papyrus for IoT – a modeling solution for IoT. ., 2016.

[156] Papyrus Moka team. Papyrus moka, 2019. URL https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution.

[157] Ferry Pramudianto, Carlos Alberto Kamienski, Eduardo Souto, Fabrizio Borelli, Lucas L. Gomes, Djamel Sadok, and Matthias Jarke. IoT Link: An internet of things prototyping toolkit. In *2014 IEEE 11th Intl Conf on Ubiquitous Intelligence and Computing and 2014 IEEE 11th Intl Conf on Autonomic and Trusted Computing and 2014 IEEE 14th Intl Conf on Scalable Computing and Communications and Its Associated Workshops*, pages 1–9, 2014. doi: 10.1109/UIC-ATC-ScalCom.2014.95. URL http://dx.doi.org/10.1109/UIC-ATC-ScalCom.2014.95.

[158] Flavio Corradini, Arianna Fedeli, Fabrizio Fornari, Andrea Polini, Barbara Re, and Luca Ruschioni. X-IoT: a model-driven approach to support IoT application portability across IoT platforms. *Computing*, 01 2023. doi: 10.1007/s00607-023-01155-z.

[159] Xuan Thang Nguyen, Huu Tam Tran, Harun Baraki, and Kurt Geihs. FRASAD: A framework for model-driven IoT application development. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pages 387–392, 2015. doi: 10.1109/WF-IoT.2015.7389085.

[160] Thiago Nepomuceno, Tiago Carneiro, Paulo Henrique Maia, Muhammad Adnan, Thalyson Nepomuceno, , and Alexander Martin. AutoIoT: a framework based on user-driven MDE for generating IoT applications. *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20)*, page 719–728, 2020. doi: https://doi.org/10.1145/3341105.3373873. URL https://doi.org/10.1145/3341105.3373873.

[161] Dimitris Soukaras, Pankesh Pately, Hui Songz, and Sanjay Chaudhary. IoTSuite: A toolsuite for prototyping internet of things applications. *The 4th Workshop on on Computing and Networking for Internet of Things (ComNet-IoT 2015)*, 2020. doi: https://doi.org/10.1145/3341105.3373873.

[162] Nguyen Xuan Thang, Michael Zapf, and Kurt Geihs. Model driven development for data-centric sensor network applications. In *Proceedings of the 9th International Conference on Advances in Mobile Computing and Multimedia*, MoMM '11, page 194–197, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450307857. doi: 10.1145/2095697.2095733. URL https://doi.org/10.1145/2095697.2095733.

[163] Arvid Butting, Arne Haber, Lars Hermerschmidt, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic language extension mechanisms for the montiarc architecture description language. In Anthony Anjorin and Huáscar Espinoza, editors, *Modelling Foundations and Applications*, pages 53–70, Cham, 2017. Springer International Publishing. ISBN 978-3-319-61482-3. doi: https://doi.org/10.1007/978-3-319-61482-3_4.

[164] Francisco Durán, Ajay Krishna, Michel Le Pallec, Radu Mateescu, and Gwen Salaün. Models and analysis for user-driven reconfiguration of rule-based IoT applications. *Internet of Things*, 19:100515, 2022. ISSN 2542-6605. doi: https://doi.org/10.1016/j.iot.2022.100515.

[165] Ivan Alfonso, Kelly Garcés, Harold Castro, and Jordi Cabot. A model-based infrastructure for the specification and runtime execution of self-adaptive IoT architectures. *Computing*, pages 1–24, 02 2023. doi: 10.1007/s00607-022-01145-7.

[166] Behailu Negash, Tomi Westerlund, Amir M Rahmani, Pasi Liljeberg, and Hannu Tenhunen. DoS-IL: A Domain Specific Internet of Things Language for Resource Constrained Devices. *Procedia Computer Science*, 109:416–423, 2017. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2017.05.411.

[167] Fei Li, Michael Vögler, Markus Claeßens, and Schahram Dustdar. Towards automated IoT application deployment by a cloud-based approach. In *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*, pages 61–68, 2013. doi: 10.1109/SOCA.2013.12.

[168] Nicolas Ferry, Phu Nguyen, Hui Song, Pierre-Emmanuel Novac, Stéphane Lavirotte, Jean-Yves Tigli, and Arnor Solberg. GeneSIS: Continuous orchestration and deployment of smart IoT systems. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 870–875, 2019. doi: 10.1109/COMPSAC.2019.00127.

[169] ISOGRAPH. Fault tree analysis in reliability workbench, 2022. URL https://www.isograph.com/. Last accessed May 2022.

[170] Jacopo Parri, Fulvio Patara, Samuele Sampietro, and Enrico Vicario. A framework for model-driven engineering of resilient software-controlled systems. *Computing*, 103, 04 2021. doi: 10.1007/s00607-020-00841-6.

[171] Faida Mhenni, Nga Nguyen, and Jean-Yves Choley. Automatic fault tree generation from SysML system models. In *2014 IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, pages 715–720, 2014. doi: 10.1109/AIM.2014.6878163.

[172] Nataliya Yakymets, Hadi Jaber, and Agnes Lanusse. Model-based system engineering for fault tree generation and analysis. In *Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,*, pages 210–214. INSTICC, SciTePress, 2013. ISBN 978-989-8565-42-6. doi: 10.5220/0004346902100214.

[173] Tatiana Prosvirnova. *AltaRica 3.0: a Model-Based approach for Safety Analyses*. Theses, Ecole Polytechnique, November 2014. URL https://pastel.archives-ouvertes.fr/tel-01119730.

[174] Hamed Fazlollahtabar and Seyed Niaki. Fault tree analysis for reliability evaluation of an advanced complex manufacturing system. *Journal of Advanced Manufacturing Systems*, 17: 107–118, 03 2018. doi: 10.1142/S0219686718500075.

[175] Kester Clegg, Mole Li, David Stamp, Alan Grigg, and John McDermid. Integrating existing safety analyses into SysML. In Yiannis Papadopoulos, Koorosh Aslansefat, Panagiotis Katsaros, and Marco Bozzano, editors, *Model-Based Safety and Assessment*, pages 63–77, Cham, 2019. Springer International Publishing. ISBN 978-3-030-32872-6.

[176] Kester Clegg, Mole Li, David Stamp, Alan Grigg, and John McDermid. A SysML profile for fault trees—linking safety models to system design. In Alexander Romanovsky, Elena Troubitsyna, and Friedemann Bitsch, editors, *Computer Safety, Reliability, and Security*, pages 85–93, Cham, 2019. Springer International Publishing. ISBN 978-3-030-26601-1.

[177] Jianwen Xiang and Kazuo Yanoo. Automatic static fault tree analysis from system models. In *2010 IEEE 16th Pacific Rim International Symposium on Dependable Computing*, pages 241–242, 2010. doi: 10.1109/PRDC.2010.35.

[178] Moomen Chaari, Wolfgang Ecker, Thomas Kruse, Cristiano Novello, and Bogdan-Andrei Tabacaru. Transformation of failure propagation models into fault trees for safety evaluation purposes. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, pages 226–229, 2016. doi: 10.1109/DSN-W.2016.18.

[179] Stefanos Katsavounis, Nikolaos Patsianis, E. Konstantinidis, and Pantelis Botsaris. Reliability analysis on crucial subsystems of a wind turbine through FTA approach. 09 2014. doi: 10.13140/2.1.2524.3849.

[180] Ivanovitch Silva, Rafael Leandro, Daniel Macedo, and Luiz Affonso Guedes. A dependability evaluation tool for the internet of things. *Computers & Electrical Engineering*, 39(7):2005–2018, 2013. ISSN 0045-7906. doi: https://doi.org/10.1016/j.compeleceng.2013.04.021. URL https://www.sciencedirect.com/science/article/pii/S0045790613001171.

[181] Yingyi Chen, Zhumi Zhen, Huihui Yu, and Jing Xu. Application of fault tree analysis and fuzzy neural networks to fault diagnosis in the internet of things (IoT) for aquaculture. *Sensors*, 17(1), 2017. ISSN 1424-8220. doi: 10.3390/s17010153. URL https://www.mdpi.com/1424-8220/17/1/153.

[182] Liudong Xing, Massarrah Tannous, Vinod M. Vokkarane, Honggang Wang, and Jun Guo. Reliability modeling of mesh storage area networks for internet of things. *IEEE Internet of Things Journal*, 4(6):2047–2057, 2017. doi: 10.1109/JIOT.2017.2749375.

[183] Ovidiu-Andrei Schipor, Radu-Daniel Vatavu, and Jean Vanderdonckt. Euphoria: A scalable, event-driven architecture for designing interactions across heterogeneous devices in smart environments. *Information and Software Technology*, 109:43–59, 2019. ISSN 0950-5849. URL https://doi.org/10.1016/j.infsof.2019.01.006.

[184] Gökhan Kahraman and Semih Bilgen. A framework for qualitative assessment of domain-specific languages. *Softw. Syst. Model.*, 14(4):1505–1526, oct 2015. ISSN 1619-1366. doi: 10.1007/s10270-013-0387-8. URL https://doi.org/10.1007/s10270-013-0387-8.

[185] Christian Moreira, Juan Cobos-Q, Wilson Valdez Solis, Cristina Sánchez-Zhunio, and Irene Priscila Cedillo Orellana. Evaluating the usability in domain-specific languages. In *Information Systems Development: Crossing Boundaries between Development and Operations (DevOps) in Information Systems (ISD2021 Proceedings), Valencia, Spain, September 8-10, 2021*, 2021. URL https://aisel.aisnet.org/isd2014/proceedings2021/hci/3.

[186] Mohamed Darwish and Essam Shehab. Framework for engineering design systems architectures evaluation and selection: Case study. *Procedia CIRP*, 60:128–132, 2017. ISSN 2212-8271. URL https://doi.org/10.1016/j.procir.2017.01.058. Complex Systems Engineering and Development Proceedings of the 27th CIRP Design Conference Cranfield University, UK 10th – 12th May 2017.

[187] Miguel Goulão, Vasco Amaral, and Marjan Mernik. Quality in model-driven engineering: A tertiary study. 24(3):601–633, sep 2016. ISSN 0963-9314. doi: 10.1007/s11219-016-9324-8. URL https://doi.org/10.1007/s11219-016-9324-8.

[188] Diego X. Jara Juárez and Priscila Cedillo. Security of mobile cloud computing: A systematic mapping study. In *2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM)*, pages 1–6, 2017. doi: 10.1109/ETCM.2017.8247486. URL https://doi.org/10.1109/ETCM.2017.8247486.

[189] John Estdale and Elli Georgiadou. Applying the ISO/IEC 25010 quality models to software product. In Xabier Larrucea, Izaskun Santamaria, Rory V. O'Connor, and Richard Messnarz, editors, *Systems, Software and Services Process Improvement*, pages 492–503, Cham, 2018. Springer International Publishing. ISBN 978-3-319-97925-0. URL https://doi.org/10.1007/978-3-319-97925-0_42.

[190] Jose Luis González, Roberto García, Josep Maria Brunetti, Rosa Gil, and Juan Manuel Gimeno. SWET-QUM: A quality in use extension model for semantic web exploration tools. In *Proceedings of the 13th International Conference on Interacción Persona-Ordenador*, INTERACCION '12, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450313148. doi: 10.1145/2379636.2379651. URL https://doi.org/10.1145/2379636.2379651.

[191] Siamak Farshidi, Slinger Jansen, and Sven Fortuin. Model-driven development platform selection: Four industry case studies. *Softw. Syst. Model.*, 20(5):1525–1551, oct 2021. ISSN 1619-1366. doi: 10.1007/s10270-020-00855-w. URL https://doi.org/10.1007/s10270-020-00855-w.

[192] CONNECT Advisory Forum (CAF). Internet of things – the next revolution "a strategic reflection about an european approach to internet of things. Technical report, European Commission, 2014.

[193] Satya Nand. Environmental energy harvesting techniques to power standalone iot-equipped sensor and its application in 5g communication. *Emerging Science Journal*, 4:116–126, 11 2021. doi: 10.28991/esj-2021-SP1-08.

[194] Amir H. Moin. Domain specific modeling (DSM) as a service for the internet of things & services. In *Internet of Things. User-Centric IoT*. Springer, Cham, 2015. doi: 10.1007/978-3-319-19656-5_47.

[195] Francesco Basciani, Juri Rocco, Davide Di Ruscio, Amleto Di Salle, Ludovico Iovino, and Alfonso Pierantonio. MDEForge: An extensible web-based modeling platform. volume 1242, 09 2014.

[196] Léa Brunschwig, Esther Guerra, and Juan de Lara. Towards access control for collaborative modelling apps. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381352. doi: 10.1145/3417990.3420201.

[197] Adriana Caione, Alessandro Fiore, Luca Mainetti, Luigi Manco, and Roberto Vergallo. Chapter 13 - WoX: Model-Driven Development of Web of Things Applications. In Quan Z Sheng, Yongrui Qin, Lina Yao, and Boualem Benatallah, editors, *Managing the Web of Things*, pages 357–387. Morgan Kaufmann, Boston, 2017. ISBN 978-0-12-809764-9. doi: https://doi.org/10.1016/B978-0-12-809764-9.00017-2.

[198] Darko Androcec and Neven Vrcek. Thing as a service interoperability: Review and framework proposal. In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 309–316, 2016. doi: 10.1109/FiCloud.2016.51.

[199] Giancarlo Fortino, Claudio Savaglio, Giandomenico Spezzano, and MengChu Zhou. Internet of things as system of systems: A review of methodologies, frameworks, platforms, and tools. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 51(1):223–236, 2021. doi: 10.1109/TSMC.2020.3042898.

[200] Andreas Wortmann, Benoit Combemale, and Olivier Barais. A systematic mapping study on modeling for industry 4.0. In *Proceedings of the ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems*, MODELS '17, page 281–291. IEEE Press, 2017. ISBN 9781538634929. doi: 10.1109/MODELS.2017.14.

[201] Sergio Teixeira, Bruno Alves Agrizzi, José Gonçalves Pereira Filho, Silvana Rossetto, and Roquemar de Lima Baldam. Modeling and automatic code generation for wireless sensor network applications using model-driven or business process approaches: A systematic mapping study. *Journal of Systems and Software*, 132:50–71, 2017. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2017.06.024.

[202] Partha Pratim Ray. A Survey on Visual Programming Languages in Internet of Things. *Scientific Programming*, 2017:1231430, 2017. ISSN 1058-9244. doi: 10.1155/2017/1231430.

[203] Christian Prehofer and Ilias Gerostathopoulos. Chapter 3 - Modeling RESTful Web of Things Services: Concepts and Tools. In Quan Z Sheng, Yongrui Qin, Lina Yao, and Boualem Benatallah, editors, *Managing the Web of Things*, pages 73–104. Morgan Kaufmann, Boston, 2017. ISBN 978-0-12-809764-9. doi: https://doi.org/10.1016/B978-0-12-809764-9.00004-4.

[204] Jorge Biolchini, Paula Gomes Mian, Ana Candida Cruz Natali, and Guilherme Horta Travassos. Systematic review in software engineering. *System Engineering and Computer Science Department COPPE/UFRJ, Technical Report ES*, 679(05):45, 2005.

[205] Marco Brambilla, Eric Umuhoza, and Roberto Acerbis. Model-driven development of user interfaces for IoT systems via domain-specific components and patterns. *Journal of Internet Services and Applications*, 8(1):14, 2017. ISSN 1869-0238. doi: 10.1186/s13174-017-0064-1.

[206] Mahda Noura, Mohammed Atiquzzaman, and Martin Gaedke. Interoperability in Internet of Things: Taxonomies and Open Challenges. *Mobile Networks and Applications*, 24(3):796–809, 2019. ISSN 15728153. doi: 10.1007/s11036-018-1089-9.

[207] Mauro Conti, Ali Dehghantanha, Katrin Franke, and Steve Watson. Internet of Things security and forensics: Challenges and opportunities. *Future Generation Computer Systems*, 78:544–546, 2018. ISSN 0167739X. doi: 10.1016/j.future.2017.07.060. URL http://dx.doi.org/10.1016/j.future.2017.07.060.

[208] Eclipse Foundation. 2020 annual eclipse foundation community report, 2020. URL https://www.eclipse.org/org/foundation/reports/annual_report.php. Last accessed July 2021.

[209] Melanie Bats and Stephane Begaudeau. Sirius web: 100 URL https://www.eclipsecon.org/2020/sessions/sirius-web-100-open-source-cloud-modeling-platform. Last accessed July 2021.

[210] André Restivo, Hugo Sereno Ferreira, João Pedro Dias, and Margarida Silva. Visually-defined real-time orchestration of IoT systems. 2020.

[211] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor C.M. Leung. Developing IoT applications in the fog: A distributed dataflow approach. In *2015 5th International Conference on the Internet of Things (IOT)*, pages 155–162, 2015. doi: 10.1109/IOT.2015.7356560.

[212] João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. Empowering visual internet-of-things mashups with self-healing capabilities. *arXiv preprint arXiv:2103.07395*, 2021.

[213] Yun Mi Antorini and Albert M Muñiz. The Benefits and Challenges of Collaborating with User Communities. *Research-Technology Management*, 56(3):21–28, 2013. doi: 10.5437/08956308X5603931. URL https://doi.org/10.5437/08956308X5603931.

[214] Olaf David, Wes Lloyd, Ken Rojas, Mazdak Arabi, Frank Geter, James Ascough, Tim Green, G. Leavesley, and Jack Carlson. Model-as-a-service (MaaS) using the Cloud Services Innovation Platform (CSIP). *Proceedings - 7th International Congress on Environmental Modelling and Software: Bold Visions for Environmental Modeling, iEMSs 2014*, 1:243–250, 2014.

[215] Faezeh Khorram, Jean-Marie Mottu, and Gerson Sunyé. Challenges & opportunities in Low-Code testing. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381352. doi: 10.1145/3417990.3420204. URL https://doi.org/10.1145/3417990.3420204.

[216] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., USA, 2nd edition, 2002. ISBN 0201729156. doi: 10.5555/559784. URL https://dl.acm.org/doi/10.5555/559784.

[217] Maryoly Ortega and et al. Construction of a systemic quality model for evaluating a software product. *Softw. Qual. J.*, 11(3), 2003. URL https://doi.org/10.1023/A:1025166710988.

[218] Oleksandr Gordieiev, Vyacheslav Kharchenko, Nataliia Fominykh, and Vladimir Sklyar. Evolution of software quality models in context of the standard ISO 25010. In *Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. June 30 – July 4, 2014, Brunów, Poland*, pages 223–232, Cham, 2014. ISBN 978-3-319-07013-1. doi: 10.1007/978-3-319-07013-1_21. URL https://doi.org/10.1007/978-3-319-07013-1_21.

[219] Marc-Alexis Côté, Witold Suryn, and Elli Georgiadou. In search for a widely applicable and accepted software quality model for software quality engineering. *Software Quality Journal*, 15(4):401–416, dec 2007. ISSN 0963-9314. doi: 10.1007/s11219-007-9029-0. URL https://doi.org/10.1007/s11219-007-9029-0.

[220] Felicien Ihirwe, Davide Di Ruscio, Simone Gianfranceschi, and Alfonso Pierantonio. Software product quality evaluation questionnaire for IoT LCDP&MDE, June 2022. URL https://doi.org/10.5281/zenodo.6631200.

[221] Bruno A. Mozzaquatro, Ricardo Jardim-Goncalves, and Carlos Agostinho. Towards a reference ontology for security in the internet of things. In *2015 IEEE International Workshop on Measurements & Networking (M&N)*, pages 1–6, 2015. doi: 10.1109/IWMN.2015.7322984. URL https://doi.org/10.1109/IWMN.2015.7322984.

[222] Eric Bouwers, Jose Pedro Correia, Arie van Deursen, and Joost Visser. Quantifying the analyzability of software architectures. In *2011 Ninth Working IEEE/IFIP Conference on Software Architecture*, pages 83–92, 2011. doi: 10.1109/WICSA.2011.20. URL https://doi.org/10.1109/WICSA.2011.20.

[223] Fahed Alkhabbas, Romina Spalazzese, Maura Cerioli, Maurizio Leotta, and Gianna Reggio. On the deployment of IoT systems: An industrial survey. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 17–24, 2020. doi: 10.1109/ICSA-C50368.2020.00012.

[224] Gourav Shah. *Ansible Playbook Essentials*. Packt Publishing Ltd, 2015.

[225] M. Hussein, S. Li, and A. Radermacher. Model-driven development of adaptive IoT systems. In *2017 MODELS Satellite Event*, volume 2019, pages 17–23, Austin, United States, September 2017. CEUR-WS. URL https://hal-cea.archives-ouvertes.fr/cea-01843007.

[226] Banks Andrew and Gupta Rahul. Mqtt version 3.1.1 plus errata 01 [online], December 2015. URL http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html. Last Accessed: September 2020.

[227] Commission Electrotechnique Internationale. *IEC 61025:2006 - Fault tree analysis (FTA)*. IEC Standards Online, 2006. URL https://webstore.iec.ch/publication/4311.

[228] Barbara Gallina and Sasikumar Punnekkat. FI4FA: A formalism for incompletion, inconsistency, interference and impermanence failures' analysis. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 493–500, 2011. doi: 10.1109/SEAA.2011.80.

[229] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The epsilon transformation language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations*, pages 46–60, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-69927-9. doi: https://doi.org/10.1007/978-3-540-69927-9_4.

[230] He Ren, Xi Chen, and Yong Chen. Chapter 6 - fault tree analysis for composite structural damage. In He Ren, Xi Chen, and Yong Chen, editors, *Reliability Based Aircraft Maintenance Optimization and Applications*, Aerospace Engineering, pages 115–131. Academic Press, 2017. ISBN 978-0-12-812668-4. doi: https://doi.org/10.1016/B978-0-12-812668-4.00006-X. URL https://www.sciencedirect.com/science/article/pii/B978012812668400006X.

[231] Fatemeh Afsharnia. Failure rate analysis. In Aidy Ali, editor, *Failure Analysis and Prevention*, chapter 7. IntechOpen, Rijeka, 2017. doi: 10.5772/intechopen.71849. URL https://doi.org/10.5772/intechopen.71849.

[232] P. Varady, Z. Benyo, and B. Benyo. An open architecture patient monitoring system using standard technologies. *IEEE Transactions on Information Technology in Biomedicine*, 6(1): 95–98, 2002. doi: 10.1109/4233.992168.

[233] Philipp Hönig, Rüdiger Lunde, and Florian Holzapfel. Model based safety analysis with smartiflow. *Information*, 8(1), 2017. ISSN 2078-2489. doi: 10.3390/info8010007. URL https://www.mdpi.com/2078-2489/8/1/7.

[234] Arthur Henrique de Andrade Melani and Gilberto Francisco Martha de Souza. Obtaining fault trees through SysML diagrams: A MBSE approach for reliability analysis. In *2020 Annual Reliability and Maintainability Symposium (RAMS)*, pages 1–5, 2020. doi: 10.1109/RAMS48030.2020.9153658.

[235] Brice Morin, Nicolas Harrand, and Franck Fleurey. Model-based software engineering to tame the IoT jungle. *IEEE Software*, 34(1):30–36, 2017. doi: 10.1109/MS.2017.11.

[236] Gözde Karataş, Ferit Can, Gamze Doğan, Cemile Konca, and Akhan Akbulut. Multi-tenant architectures in the cloud: A systematic mapping study. In *2017 International Artificial Intelligence and Data Processing Symposium (IDAP)*, pages 1–4, 2017. doi: 10.1109/IDAP.2017.8090268.

[237] Katharina Görlach and Frank Leymann. Dynamic service provisioning for the cloud. In *2012 IEEE Ninth International Conference on Services Computing*, pages 555–561, 2012. doi: 10.1109/SCC.2012.30.

[238] Felicien Ihirwe. Home automation system failure logic behavior rules, January 2023.

Funding

“This project has received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 813884”

Jean Felicien Ihirwe, Pisa, March 2023