CHESSIoT: a model-driven approach for engineering multi-layered IoT systems

Felicien Ihirwe*[a,b], Davide Di Ruscio[a], Simone Gianfranceschi[b], Alfonso Pierantonio[a]

[a]*Department of Information Engineering Computer Science and Mathematics, University of L'Aquila , L'Aquila, Italy*
[b]*Innovation Technology Services Lab, Intecs Solutions S.p.A, Pisa Italy*

Abstract

Context: The current technology revolution, which places the highest value on people's welfare, is frequently seen as being mainly supported by Internet of Things (IoT) technologies. IoT is regarded as a powerful multi-layered network of systems that integrates several heterogeneous, independently networked (sub-)systems working together to achieve a shared purpose.
Objective: In this article, we present CHESSIoT, a model-driven engineering environment that integrates high-level visual design languages, software development, safety analysis, and deployment approaches for engineering multi-layered IoT systems. With CHESSIoT, users may conduct different engineering tasks on system and software models under development to enable earlier decision-making and take prospective measures, all supported by a unique environment.
Methodology: This is achieved through multi-staged designs, most notably the physical, functional, and deployment architectures. The physical model specification is used to perform both qualitative and quantitative safety analysis by employing logical Fault-Trees models (FTs). The functional model specifies the system's functional behavior and is later used to generate platform-specific code that can be deployed on low-level IoT device nodes. Additionally, the framework supports modeling the system's deployment plan and run-time service provisioning, which would ultimately be transformed into deployment configuration artifacts ready for execution on remote servers.
Results: To showcase the effectiveness of our proposed approach, as well as the capability of the supporting tool, a multi-layered Home Automation system (HAS) scenario has been developed covering all its design, development, analysis, and deployment aspects. Last but not least, a comparative analysis of CHESSIoT to other existing platforms has been conducted along with other assessment methods, and the findings are considered promising.

Keywords: Model-driven engineering, Internet of Things, System design, Safety analysis, Code generation, System deployment

1. INTRODUCTION

Internet of Things, Machine Learning, and Cloud Computing are expected to drive the next wave of the industrial revolution [1]. IoT is regarded as a powerful network of systems architecture that combines several heterogeneous, independently networked (sub-)systems working together to achieve a common goal that the independently operating systems cannot realize. A typical IoT system consists of multiple layers: the *edge layer* generally comprises devices and sensors that collect data from the physical world and communicate it to the next layer; the *fog layer* is in charge of transmitting data acquired by the edge layer to the *cloud layer*, which is a centralized repository where data from all devices is stored and analyzed. This layer also includes data storage, analysis, and management services. Each layer of IoT systems is crucial to make them operate efficiently and offer users valuable insights and automation capabilities [2].

IoT systems generate massive amounts of data from sensors and devices, which increases development costs and time as system complexity grows [3]. These systems exhibit significant heterogeneity in all aspects and are often deployed in harsh environments, which can lead to errors and failure that may result in human harm. As reactive systems, IoT devices constantly interact with their surroundings, making IoT applications highly dynamic and prone to unexpected behavior [4]. Identifying unexpected behavior while ensuring essential functionality can be challenging, especially in a dynamic system. To alleviate these challenges, developers can use integrated development environments (IDEs) based on domain-specific high-level languages, which can abstract many of the intricacies and specifics of hardware, software, communication media, and protocols [5]. Such an approach would reduce heterogeneity and technological hurdles and promote advanced automated software engineering tools that enable faster and more secure software development. Additionally, these tools should provide means for reducing

*Corresponding author
Email addresses: jeanfelicien.ihirwe@graduate.univaq.it (Felicien Ihirwe*), davide.diruscio@univaq.it (Davide Di Ruscio), simone.gianfranceschi@intecs.it (Simone Gianfranceschi), alfonso.pierantonio@univaq.it (Alfonso Pierantonio)

user effort and maintaining system correctness during development to ensure system robustness. Several successful platforms such as IBMWatson IoT, Eclipse IoT, and AWS IoT are available to address these challenges, but their usability complexity poses further challenges.

Model-driven engineering (MDE) has demonstrated significant benefits in automating the software development process by promoting the use of models as first-class citizens, allowing subsystems to be designed, developed, and analyzed independently before integration into a fully functional system [6]. In addition, domain-specific languages (DSLs) tailored to specific application domains are used in MDE to define domain models and enable domain experts to define system behavior based on their expertise, improving productivity and communication with developers [7]. In the IoT domain, MDE can be viewed as a methodology that focuses on defining IoT devices and data processing relying on them. MDE has automated several development processes, and platforms like MDE4IoT, ThingML, IoTML/BrainIoT, SimulateIoT, and Montithings demonstrate its potential as a realistic alternative for developing scalable IoT systems [8, 9, 10, 11, 12, 13]. However, finding a platform that integrates design, development, analysis, and deployment infrastructures is critical to engineering IoT systems entirely. This article uses the term "engineering" to refer to the process integrating "development, analysis, and deployment" when realizing IoT systems.

This paper presents CHESSIoT, a model-driven framework for engineering multi-layered IoT systems. CHESSIoT permits users to design, develop, analyze, and deploy engineering IoT systems 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 CHESSIoT environment is built on top of the CHESS toolchain [14] to provide a fully decoupled extension for supporting the design, development, analysis, and deployment, targeting multi-layered IoT systems. 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. To achieve that, the designer relies on a CHESSIoT DSL in which the meta-modeling syntax has been specified as an extension to both UML and SysML languages.

CHESSIoT DSL comprises three primary DSLs, i.e., the *System-level*, *Software*, and *Deployment* DSLs. *System-level* DSL focuses on the system-level architecture of the system across all layers of a typical IoT system by enabling early safety analysis. In CHESSIoT safety analysis, the individual components are annotated with their failure behavior following error propagation and transformation rules [15]. An automated Failure Logic Analysis (FLA) can be performed when the model is complete. During the analysis, each behavior is systematically analyzed to determine the top-level system failures. Furthermore, it is possible to fully generate the system's Fault Trees (FTs) and perform qualitative and quantitative Fault Tree Analysis (FTA) based on the component's failure probabilities.

Following a component-based design methodology, the *Software DSL* environment offers means for the user to compose all the *software components of the system*, their *internal decomposition*, and their *functional behaviors* through the use of state machines. In doing so, internal payloads, events, actions, and guards are associated with states and their transitions to realize the desired component's behavioral goal. When the model is complete, a CHESSIoT2ThingML model transformation can be applied to generate a series of fully functional ThingML [9] source models. ThingML is amongst the most popular MDE tools for the IoT domain.

A typical IoT system's components can be deployed at any layer, namely edge, fog, and cloud. Thus, the CHESSIoT *Deployment DSL* offers means for modeling the system deployment plans and its runtime service provisioning. The deployment model connects the software to the actual system nodes in which the software program will be executed. The model decomposes the interdependency between nodes, machines, and services deployed to it. When the model is complete, a model-to-text transformation can be launched, generating a .yaml configuration file ready to be executed on a docker server.

Runtime service provisioning refers to allocating and configuring resources, such as computing power, storage, and network connectivity, to make the modeled system work [16]. In runtime service provisioning, resources are dynamically allocated based on the needs of the program or application at any given time. CHESSIoT offers a model-driven runtime service provisioning environment that automatically configures 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, which is later transformed into Ansible playbook scripts [17] that can be run manually on a remote machine based on the status of the deployed configuration.

we summarise the contribution of this paper as follows:

- We present CHESSIoT, a domain-specific language for modeling the system, functional, behavior, and deployment architectures of a multi-layered IoT system.
- To assess CHESSIoT's potential contributions, we provide and discuss a comparative analysis of the CHESSIoT tool with existing approaches.
- We show in detail the CHESSIoT safety analysis approach in terms of its support for qualitative and quantitative Fault-Tree Analyses.
- We describe the development and deployment approach and the generation of supported artifacts at different development stages.
- Finally, We present a Home Automation system use-case in which we used the tool to design, develop, conduct

safety analysis, and support its deployment.

Structure of the paper: Section 2 presents an overview of existing model-based approaches for engineering IoT systems. Section 3 presents the comparative analysis of existing approaches for engineering IoT systems covering the motivation behind our proposed approach in relation to existing approaches. Section 4 presents the proposed approach, which includes the CHESSIoT DSL 4.1, the supported model-based safety analysis 4.2, the development 4.3, and the deployment 4.4 approaches. Section 5 presents a Home Automation System running example to showcase the capability of the supporting tool in covering all three engineering tasks. Section 6 presents a brief that discusses the results as well as highlights the main limitation of our approach whereas Section 7 concludes the paper as well as the future work.

2. OVERVIEW OF EXISTING MODEL-BASED APPROACHES FOR ENGINEERING IOT SYSTEMS

Model-driven engineering (MDE) is a software development methodology that emphasizes the creation and utilization of domain models throughout the system development process. MDE has proven effective in managing complex problems in software engineering by relying on abstraction. By defining models with concepts that are independent of underlying implementation technology and more closely related to the problem domain of interest, MDE can boost productivity and speed up time to market [18]. In the context of IoT, MDE focuses on defining the behaviors of IoT devices and the data they process rather than the software that runs on them. MDE has contributed to automating various development processes in IoT by capturing the system's requirements, architecture, and design in models that can be transformed into the required implementation artifacts using code generators [9, 19, 13]. However, developing IoT code generators that can handle large models accommodating a diverse set of client requirements remains challenging due to the high heterogeneity in IoT hardware devices, data sources, protocols, and deployment levels [20].

This section provides an overview of existing model-based approaches covering IoT engineering aspects, including modeling, development, analysis, and deployment. In Section 2.1, we present approaches focusing on IoT software modeling and code generation. Section 2.2 presents existing research that focuses on fault-tree analysis for safety analysis of IoT systems. Finally, we cover related approaches that focus on deployment modeling and possible support for runtime deployment of IoT systems in Section 2.3.

2.1. Model-based development of IoT systems

In this section, we focus on the MDE approach to software modeling and development that generates code ready for deployment on IoT devices.

Ciccozzi, F. et al. [8] introduced MDE4IoT, an MDE platform that combines different UML DSLs to support the design, development, and runtime management of IoT systems by providing means for modeling and self-adaptation of Emergent Configurations (ECs) of connected systems. MDE4IoT uses model-to-model and model-to-text transformations to generate platform-specific code from state machines. The platform also supports run-time monitoring and self-adaptations through re-allocations and re-generation mechanisms based on the system's runtime feedback.

Costa B. et al. [21] developed SysML4IoT, a tool for Model-Based Systems Engineering in the context of IoT application development, with a focus on the design phase. This tool builds upon the IoT-A domain reference model [1], established by a European research body, and the ISO/IEC/IEEE 15288 standard [2], to enrich system models with Systems Engineering concepts. SysML4IoT adopts a multi-disciplinary approach to IoT application design by utilizing views and viewpoints to cater to different stakeholders involved in the process. In [22], SysML4IoT was extended to assist IoT application engineers in accurately modeling IoT applications and verifying their quality of service (QoS) properties. A model-to-text translator was developed to convert the model and QoS properties specified on it to be executed by NuSMV [23], a mature model checker that enables the entry of a system model consisting of many communicating Finite State Machines (FSM) and automatically checks its properties expressed as Computational Tree Logic (CTL) or Linear Temporal Logic (LTL) formulas.

Thramboulidis K. et al. [24] introduced UML4IoT, an MDE platform for industrial automation systems, which supports the automation of the generation process of IoT-compliant layers required for the cyber-physical component to be integrated into the modern IoT manufacturing environment. It achieves this by transforming mechatronic components into Industrial Automation Things (IAT) through model-to-model transformation. UML4IoT utilizes the Open Mobile Alliance (OMA) Lightweight Machine to Machine (LWM2M) application protocol, which runs on top of the Constrained Application Protocol (CoAP) communication protocol, to expose the IoT interface as simple smart objects [25]. The platform 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 unavailable.

Harrand N. et al. [9] presented ThingML, an IoT engineering platform that combines well-proven textual software-modeling constructs aligned with UML, such as statecharts and components, with an imperative platform-independent action language for developing IoT applications. In ThingML, a thing is defined by a set of properties, functions, messages, ports, and state machines, and 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 via message exchanges. ThingML also provides an advanced multi-platform code generation framework that supports multiple target

[1] https://www.iot-a.eu/public/
[2] https://standards.ieee.org/ieee/15288/5673/

programming languages such as C/C++, Java, Arduino, JavaScript, Python, and Go.

In their paper, Nicholson et al. [10] introduced IoTML, an integrated modeling tool developed in the context of the BRAIN-IoT project [11] to facilitate the rapid prototyping of intelligent cooperative IoT systems based on shared models. IoTML is implemented as a Papyrus profile within the BRAIN-IoT modeling environment, which consists of three macro-blocks: the BRAIN-IoT Modeling Framework, the Marketplace, and the Federation of BRAIN-IoT Fabrics. Models created using IoTML are transformed into XML format and uploaded to the BRAIN-IoT marketplace for run-time system deployment and dynamic remote edge/cloud reconfiguration.

Meanwhile, Claudio et al. [26] presented COMFIT, a Cloud and Model-based IDE for the Internet of Things, specifically designed to target wireless sensor network (WSN) applications. The COMFIT modeling environment is built on top of Papyrus and offers a simple multi-view interface for modeling the system's requirements, structural and behavioral aspects. The tool allows for the creation of wireless nodes and communication links in the structural view, while model activities and behaviors are represented as functional units that can be linked together based on the desired execution sequence. Additionally, COMFIT provides a model-checking infrastructure that follows the OCL rules specified in the meta-model.

Muccini H. et al. [27] presented CAPS, an architecture-driven modeling framework for developing situational aware cyber-physical systems. CAPS employs a multi-view architectural approach that combines software component design and interactions, hardware specification for situational awareness, and the physical environment where hardware equipment is deployed. In [28], the authors introduced CAPSml, an extension to CAPS that supports platform-specific code generation using ThingML [9].

Dhouib S. et al. [29] introduced Papyrus4IoT, a modeling tool developed under the Smart, Safe, and Security Software Development and Execution Platform (S3P) project. Papyrus4IoT provides an environment for designing and deploying complex IoT systems following an IoT-A reference architecture [2]. The designer can define process specification definition, functional and operational platforms, and deployment, which involves allocating the system's functional blocks to the device processing units. The authors proposed using development-time models to supervise a running IoT system to reflect the Models@Runtime monitoring approach. This approach helps detect overall system critical states and make decisions on adapting the running system.

Salihbegovic A. et al. [5] introduced DSL-4-IoT, a high-level visual programming language-based tool designed to simplify the complexity and heterogeneity of IoT systems. With the help of the editor, the application designer can configure the system structure and select devices, sensors, and actuators from built-in library modules. Once the design is complete, the user can export the data into a JSON array configuration file that contains information about the position of all items, relationships between items and groups, and the value of all configured fields associated with items and data types. The configuration files can then be transferred manually to the respective OpenHAB runtime directory or automatically downloaded using a simple web service for execution.

Erazo-Garzón L. et al. [30] presented Monitor-IoT, a graphical designer based on the Obeo Designer Community and Eclipse Sirius tools. Monitor-IoT supports developers in modeling IoT multi-layer monitoring architectures with a high level of abstraction, expressiveness, and flexibility. The tool enables 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 interesting concepts, it does not support the generation of any code, monitoring scripts, or data flows that can be executed on an actual IoT system.

J. A. Barriga et al. [12] presented the SimimulateIoT tool, which enables users to design complex IoT simulation environments and deploy them without writing code. The approach relies on a domain metamodel, a Eugenia-generated graphical concrete syntax, and a series of model-to-text transformations to generate cross-layer code from sensors, actuators, fog nodes, and cloud nodes. During the simulation phase, a set of Object Constraint Language (OCL) [31] constraints can be defined to validate the model's correctness in compliance with the SimulateIoT metamodel. When the generation process is finished, the tool may deploy the generated artifacts as microservices and Docker containers, with which the generated elements are coupled using a publish-subscribe communication protocol.

2.2. *Model-based safety analysis of IoT systems*

In this section, we review existing approaches that enable model-based analysis using the Fault-Tree Analysis (FTA) approach. While our focus is on the IoT domain, we also consider approaches with a broader scope, as there is a lack of IoT-specific research supporting safety analysis through the FTA methodology.

One widely used tool in both industry and academia for FTA is the ISOGRAPH tool [32]. The ISOGRAPH Reliability workbench is a powerful visual modeling and analysis environment used in the system engineering domain. ISOGRAPH provides various reliability analysis features such as failure rate and maintainability prediction, Failure Mode Effects & Criticality Analysis (FMECA), Reliability Allocation, Reliability Block Diagram, and Fault Tree, Event Tree, and Markov analysis. However, in ISOGRAPH the Fault Trees are manually constructed based on the system failure requirements provided by safety experts.

Silva I. et al. [33] introduced a dependability evaluation tool for IoT applications that considers hardware and permanent link faults. This tool enables the modeling of the system network architecture and the definition of network failure condition

events (*nfc*) that are later used to generate the FT. The *nfc* formalism follows logical association rules for addition and multiplication to reflect "OR" and "AND" gates, respectively. The tool supports both qualitative and quantitative analysis by generating minimal cut-sets. While this tool supports automatic generation and analysis of FTs, it differs from our approach in terms of system failure behavior formalism and does not support any mechanism related to failure transformation, propagation, and injection.

Chen Y. et al. [34] proposed a fault diagnosis method based on a combination of FTA and fuzzy neural networks for aquaculture IoT systems. In their approach, the FT is manually constructed for each system component, and the "IF-THEN" rules are extracted from the FT for the fuzzy neural network to learn the relationship between fault symptoms (failures) and faults. While this method uses FTA for safety analysis, it differs from our approach in that the FT generation is manual, and no quantitative analysis is supported.

Xing L. et al. [35] introduced an approach to model the failure behavior of mesh storage area networks (SANs) using a dynamic fault tree (DFT) or a network graph of imperfect links. The reliability of the mesh SAN is evaluated using a binary decision diagram-based method. The results provide insights into the general behavior of mesh SAN systems, providing guidelines for the reliable design and operation of SANs. However, like the ISOGRAPH tool, the FT construction is done manually from the system failure requirements provided by safety experts.

Alfred et al. [36] proposed Relational Reference Attribute Grammars for modeling and analyzing IoT systems. Reference Attribute Grammars (RAGs) support declarative analysis over abstract syntax trees and are used for building compilers and other language-based tools. The device-dependency analysis methodology computes what devices must be available and connected for a specific event, such as turning on a light when a door opens. The analysis computes the control flow in composition scripts as well as the transitive closure of all dependency expressions and projects the expanded dependency expression down to a set of sets of devices. The final dependency tree is calculated throughout the calculation of the device-to-device transitive closure.

In their work, Parri J. et al. [37] introduced JARVIS (Just-in-time ARtificial intelligence for the evaluation of Industrial Signals), which is a model-driven tool designed to facilitate the integration of physical IoT devices, enterprise-scale software agents, data analytics, and human operators. The tool uses agents to develop and integrate intelligent data agents capable of detecting failure events following a set of failure modes, and a FaultTreeAnalyzer agent to perform Fault Tree Analysis on detected failure events. Although this approach performs a qualitative analysis, it does not support the quantitative one, and its FT generation approach relies on the practical data model constructed by the deployed agent, whereas our approach relies on FLA.

Various approaches for the automatic generation of FTs from SysML models have been proposed; however, they do not explicitly target the IoT domain. For example, the authors of [38] proposed an approach for generating FTs from SysML models, which relies on information provided in activity and IBD diagrams and the FMEA table. Although the current tool generates a single FT picture representing the system failure paths, no FT models are generated. In [39, 40], the authors presented an MDE environment for performing preliminary safety analysis from SysML models. They 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.

In [41], the authors presented a framework that integrates the formal method approach for facilitating the automatic FT generation within an MDE workflow. The approach annotates SysML model elements with 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, the model is transformed into an AltaRica model [42] representation to perform qualitative and quantitative analysis.

2.3. Model-based deployment of IoT systems

In this section, we review various approaches that focus on model-based deployment, runtime management, and automation of IoT systems across all layers. While some approaches may overlap with the categories mentioned above, we have included them here because of their significant deployment support.

MontiThings, an integrated modeling language for IoT applications and their deployment, was proposed by Kirchhof J. C. et al. [13]. MontiThings provides a model-driven toolchain for generating executable IoT containers, automated deployment planning, deployment suggestions, and monitoring of the generated container, with the ability to suggest deployment goal changes based on deployment planning feedback. This approach is targeted mainly at the edge layer.

Duran et al. [43] proposed a technique for reconfiguring running IoT applications using coordinated rules acting on devices. The approach compares two versions of an application (before and after reconfiguration) to ensure that several functional and quantitative properties are satisfied. This information can be used by the user to decide whether to trigger the deployment of the new application. The approach supports the composition of rules using advanced Event-Condition-Action (ECA) rules, such as sequential execution, the choice between rules, concurrent execution, or repetition. Property-based verification is used to analyze whether the proposed reconfiguration preserves the application's consistency.

Alfonso et al. [44] proposed a Domain-Specific Language (DSL) that covers the static and dynamic aspects of IoT deployment. The DSL covers modeling primitives for the four layers of an IoT system (IoT devices, edge, fog, and cloud nodes), including the deployment and grouping of container-based applications. The tool also supports a sublanguage for expressing adaptation rules to ensure QoS at runtime. A proof of concept generator for deploying the

modeled IoT system on a K3S-based infrastructure (Kubernetes distribution built for IoT and edge computing) is provided.

DoS-IL, a textual domain scripting language for resource-constrained IoT devices, was introduced by Negash B. et al.[45]. It allows changing the system's behavior after deployment through a lightweight script written in the DoS-IL language and stored in a gateway at the fog layer, supporting easy maintenance and modification after deployment without physical access to the end node. The gateway hosts an interpreter to execute DoS-IL scripts that devices in the perception layer can access, while the Device Object Model (DOM) on the target node exposes the available resources for the DoS-IL script to manipulate.

Topology and Orchestration Specification for Cloud Applications (TOSCA) was presented by Vögler F. Li, M. [46]. TOSCA aims to improve the reusability of service management processes and automate IoT application deployment in heterogeneous environments. It specifies a meta-model for describing the structure and management of IT services, providing a formal way to describe the internal topology of application components and the deployment process of IoT applications. TOSCA can model common IoT components, such as gateways and drivers, as well as platform-specific properties necessary for layer-specific deployment, using various XML-like textual notations to ease deployment on heterogeneous devices and platforms.

Ferry et al. [47] proposed GENESIS, a cloud-based domain-specific modeling language that facilitates continuous orchestration and deployment of Smart IoT systems on edge and cloud infrastructures. The component-based approach used in GENESIS allows for the separation of concerns and reusability, making the deployment models an assembly of components. The GENESIS execution engines support three types of deployable artifacts: ThingML model [9], Node-RED container [19], and any black-box deployable artifact (e.g., an executable jar). The created deployment model is then passed to the GENESIS deployment execution engine, which is responsible for deploying the software components, ensuring communication between them, supplying the required cloud resources, and monitoring the deployment's status.

3. COMPARATIVE ANALYSIS OF EXISTING APPROACHES FOR ENGINEERING IOT SYSTEMS

In the previous section, we provided an overview of existing model-based 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 of 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 3.1 presents an overview of the analyzed approaches, whereas Section 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 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.

3.1. Selected platforms

Table 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 Section 2. 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.

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. 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

Table 1: Selected approaches for the comparative analysis

Tool name	Title	Year	Type
MontiThings [13]	MontiThings: Model-driven development and deployment of reliable IoT applications	2021	Journal
ThingML[9]	ThingML: A language and code generation framework for heterogeneous targets	2016	Conference
MDE4IoT[8]	MDE4IoT: Supporting the Internet of Things with model-driven engineering	2017	Conference
SysML4IoT[21]	Modeling IoT applications with SysML4IoT	2016	Conference
Monitor-IoT[30]	A domain-specific language for modeling IoT system architectures that support monitoring	2022	Journal
Simulate-IoT[12]	Simulate-IoT: Domain-specific language to design, code generation and execute IoT simulation environments	2021	Journal
DSL-4-IoT[5]	Design of a domain-specific language and IDE for Internet of Things applications	2015	Conference
UML4IoT[24]	UML4IoT-A UML-based approach to exploit IoT in Cyber-Physical manufacturing systems	2016	Journal
IoT-ML/ BRAINIoT[10]	BRAIN-IoT: Model-based framework for dependable sensing and actuation in intelligent decentralized IoT systems	2019	Conference
CAPS[27]	CAPS: Architecture description of situational aware Cyber-Physical systems	2017	Conference
Node-RED[19]	Node-RED: Low-code programming for event-driven applications	2016	Open tool
Silva I. et al. [33]	A dependability evaluation tool for the Internet of Things	2013	Journal

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 the Table 2 are highlighted below relative to CHESSIoT supported modeling features that will be presented with details in the next section:

1. From Table 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[9], 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 [13], CHESSIoT 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 [49], 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 [8], SysML4IoT [21], and CAPS [27] 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 [19] 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 2, the majority of

Table 2: Comparative table on supporting different IoT modeling features

Tool	*Graphical user interface*	*Multi-view 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[13]	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[9]	No	No	Yes	Yes	Yes	No	Yes	No	No	No	No	No	No	No	Textual Component and Connector architectures, asynchronous messaging
MDE4IoT[8]	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[21, 22, 48]	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[30]	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[12]	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[5]	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[24]	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[10, 11]	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[27, 28]	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[19]	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. [33]	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

approaches support modeling at the edge layer elements, namely components such as sensors/actuators, computing boards, and so on [8, 48, 13, 30, 19]. Except for IoT-ML [10], 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 [8, 48] as well as runtime error handling capabilities [13, 30] at the edge. While technically allowing predefined functional node behaviors to be defined, Node-RED [19] 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 [50]. 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 eleven 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 [30, 19, 12, 33], 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 [11] and SimulateIoT [12] 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 [19] and MonitorIoT [30] 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.

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 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 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 3 shows that three of the considered approaches, namely [24, 5, 33], 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 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 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 [30], DSL-4-IoT[5], and Node-RED[19] - 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 [18]. 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

Table 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[13]	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[9]	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[8]	Yes	Yes	No	No	No	No	*Case study*	Support design, development, and run-time management of IoT systems
SysML4IoT[21, 22, 48]	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[30]	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[3].
Simulate-IoT[12]	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[5]	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[24]	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[10, 11]	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[27, 28]	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[19]	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. [33]	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.

models into ThingML's models. More information will be provided in Section 4.3.

3. Developers of IoT systems often assume that their devices or systems will always succeed, but this is not the case [51]. 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 [33]

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 [52]. Many safety analysis approaches presented in Section 2.2 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 [22] supports reliability analysis of IoT systems through verification of the system's QoS properties, while the BRAINIoT platform [11] uses IoT-ML [10] 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 [14] on which the platform is built. For example, in previous research [53, 54], 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.

4. PROPOSED APPROACH

This section introduces a new engineering methodology for IoT and its accompanying tool, CHESSIoT. Our approach is designed to provide IoT developers with a modeling environment to engineer multi-layered IoT systems. With CHESSIoT, users can take advantage of a multi-view development environment, each with its constraints that enforce specific privileges on model entities and properties that can be manipulated. Different aspects of software can be modeled independently and then interlinked to meet specific engineering requirements. Users can perform various engineering activities on CHESSIoT models, such as generating IoT device code, performing analyses, and deploying applications. Figure 1 provides a high-level illustration of the proposed approach supported by the CHESSIoT tool. In particular, the approach relies on three domain-specific languages for system, software, and deployment specifications. These DSLs are aligned with different modeling views as well as the engineering task that they correspond to. A more detailed explanation of CHESSIoT DSL is given in Section 4.1. Depending on the user's needs and the stage of the development process, a specific view-compliant model corresponding to a specific metamodel could be developed.

In the *System view*, an IoT engineer creates a model of the entire IoT system, including its functional components, sub-components, and connections. This model can be given to a safety expert who will add failure logic behavior and basic component failure rates for analysis. Qualitative and quantitative analysis can be done through model-to-model transformations, including Fault Trees Analysis. Section 4.2 provides more detailed information on this topic.

Users can create a functional model in the *Component View* that includes the system's key software components, sub-functions, and relationships. Additionally, the Behavior Model allows each main sub-function of the system to have its own state machine, which defines events, actions, and guards associated with states and their transitions to achieve the desired behavior. Once the model is complete, the CHESSIoT2ThingML model transformation is initiated, generating a series of functional ThingML source models that can be used to create platform-specific code ready for deployment on low-level IoT devices. The same CHESSIoT software model can be expanded with other extra-functional properties and benefit from analysis support. For example, as demonstrated in previous work [53], CHESS can perform early real-time schedulability analysis on CHESSIoT models. More information on these aspects can be found in Section 4.3.

In the Deployment view, users can create a plan for deploying an IoT system and set rules for managing services during run-time. This plan breaks down the relationships between different layers, machines, and the services deployed on them. Additionally, a DevOps engineer can create a model for automatically configuring software services at run-time. Once the model is complete, it is transformed into a .yaml configuration and Ansible playbook scripts, which can be executed on a docker server. For more information on these topics, please refer to Section 4.4.

4.1. CHESSIoT DSL

The CHESSIoT modeling environment has been built on top of the Eclipse Papyrus[4] in terms of extensions of UML/SysML. The three profiles that make up the CHESSIoT DSL are explained in detail below.

4.1.1. 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

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

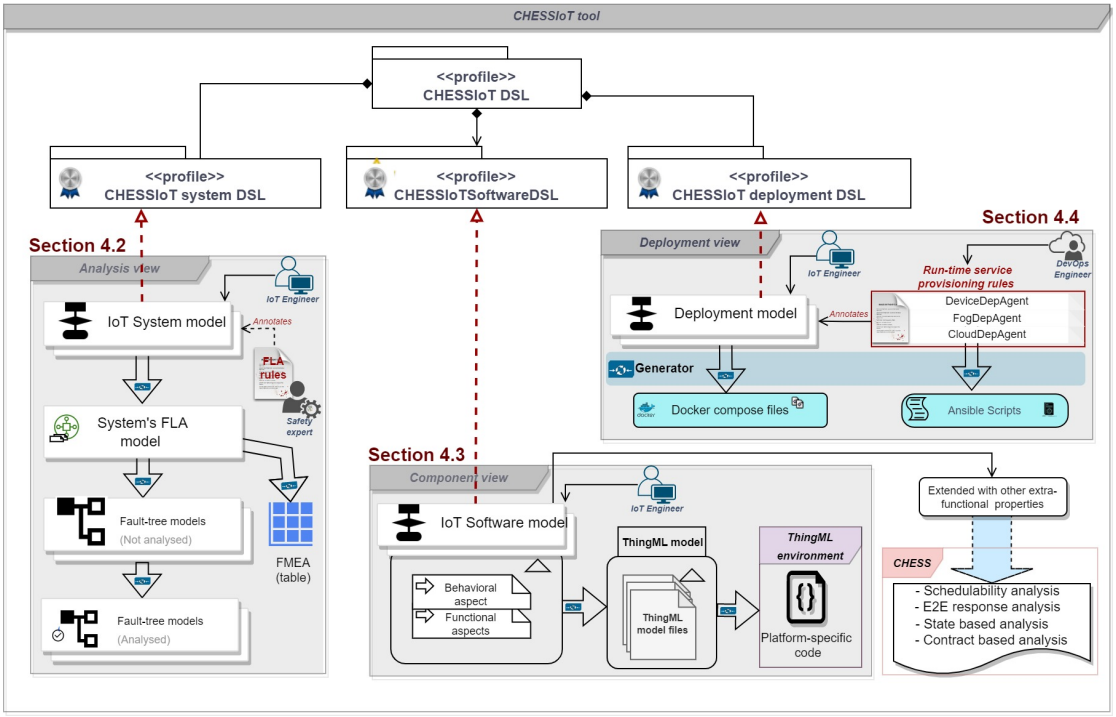


Figure 1: Overview of the CHESSIoT approach approach

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. 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.

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.

4.1.2. 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 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.

Functional aspects: *VirtualElement* represents an *IoTelement* in the virtual world. As mentioned in Section 4.1.1, this could be classified as any IoT device, an element that could be of interest at the edge. As shown in Fig. 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

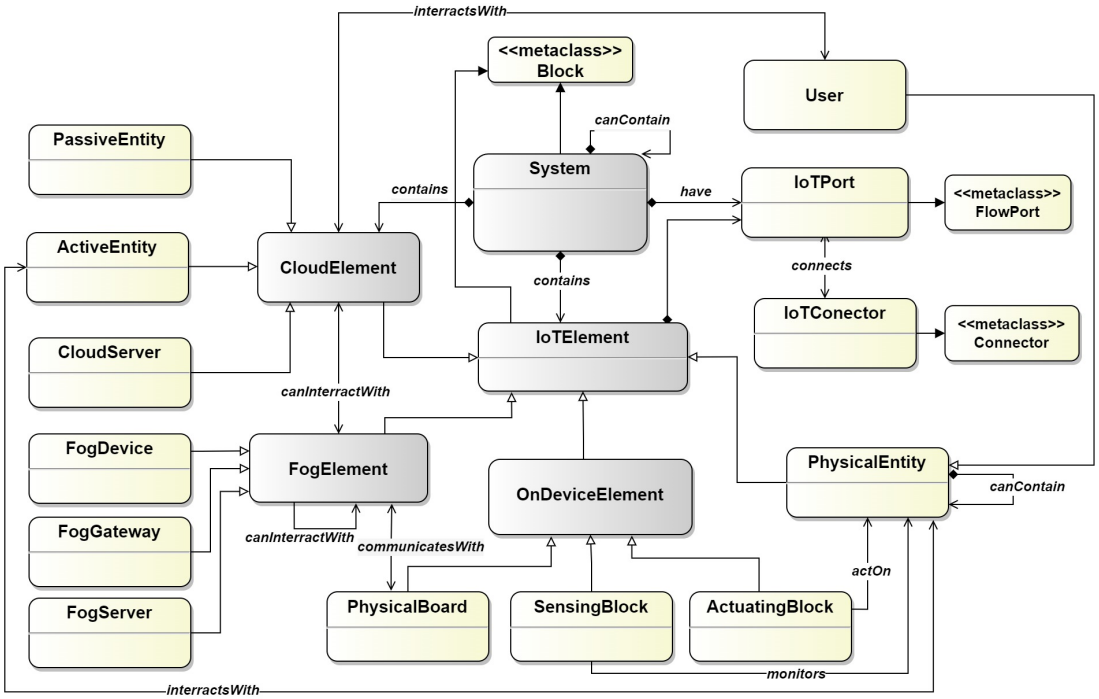


Figure 2: CHESSIoT System-level metamodel

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.

4.1.3. Deployment DSL

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. 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 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,

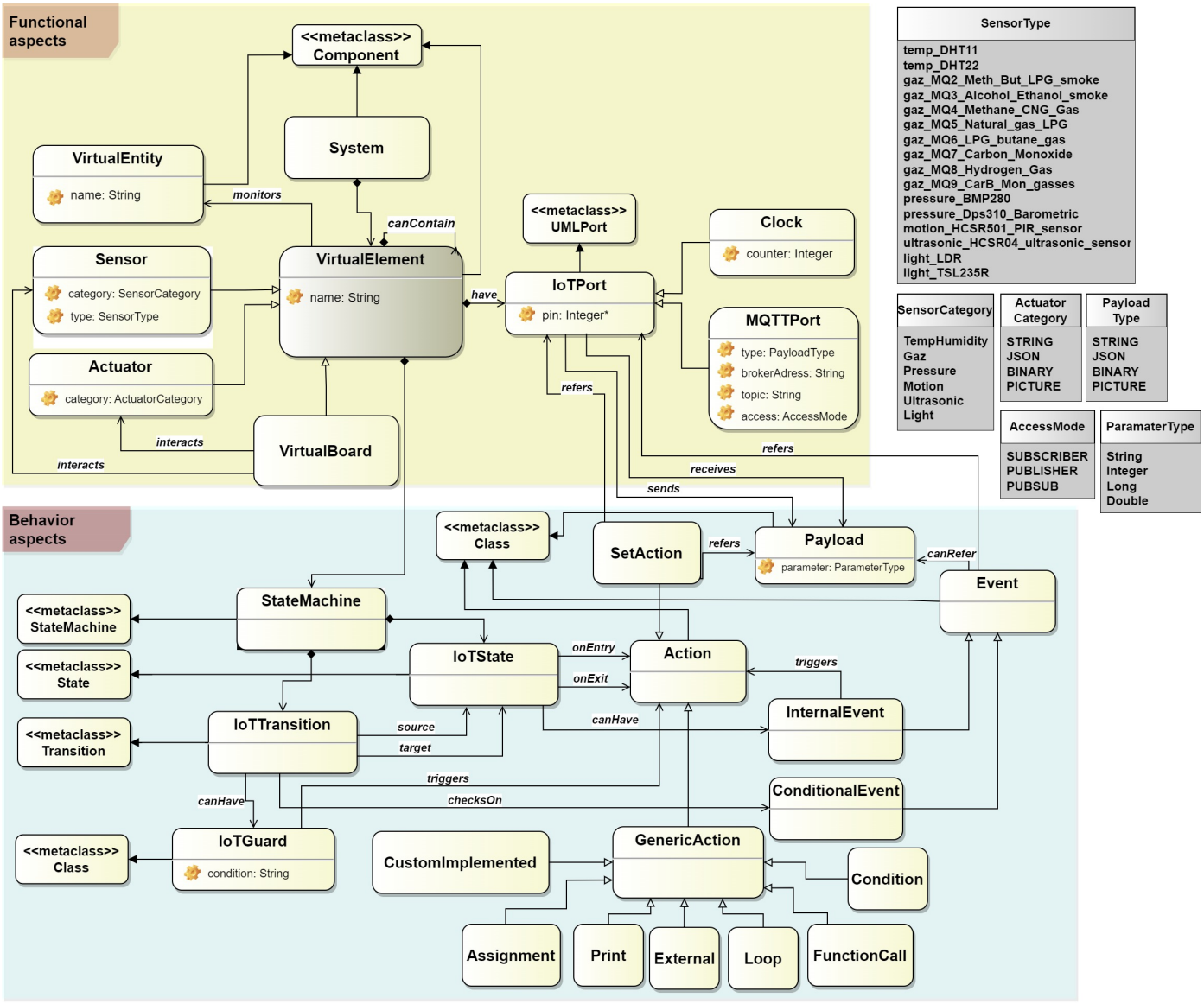


Figure 3: CHESSIoT Software metamodel

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[5], 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 Section 4.4.

4.2. Model-based safety analysis

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 [55].

As illustrated in Figure 5, the safety analysis process typically commences with the *IoT system engineer* creating a

[5] https://nodered.org/

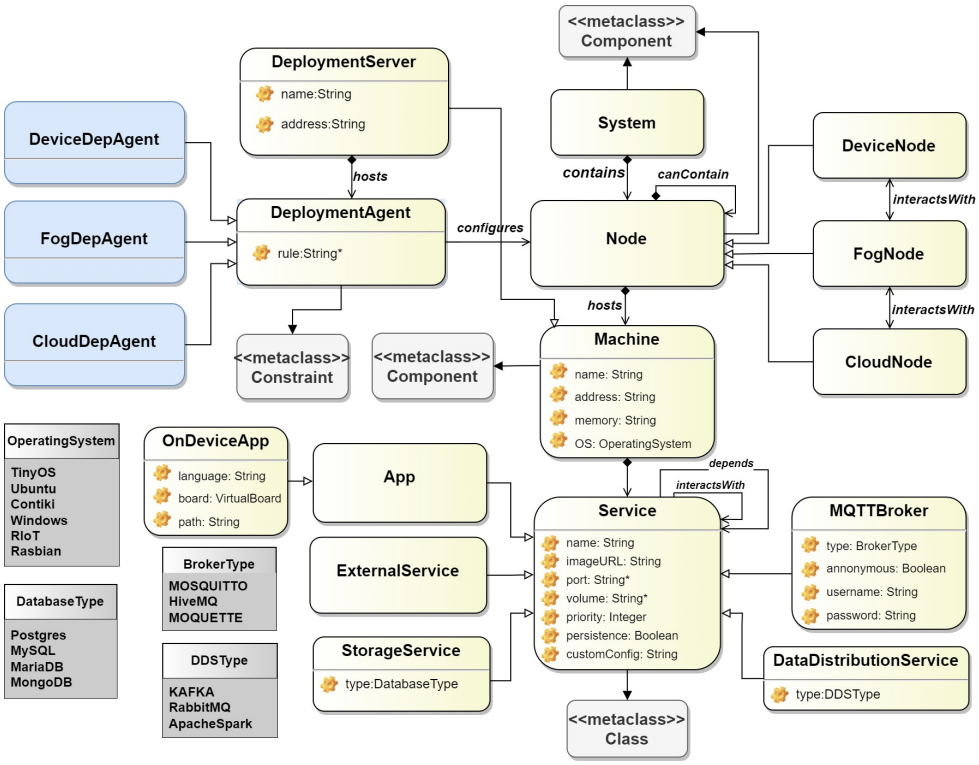


Figure 4: CHESSIoT Deployment metamodel

model based on the gathered *system functional requirements* ① *in Figure 5*. 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 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.

Once the system model is complete (see the *CHESSIoT model* ② in Fig. 5), 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.

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 is capable of avoiding 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) [15]. To support the analysis, the user must establish error propagation or transformation rules for each possible input failure or internal failure of the component. The equation in 1 presents a generic structure in which the FLA rules are specified.

$$FLA: p_{in1}.failure_{in1}, .., p_{inN}.failure_{inN} \rightarrow p_{(out1)}.failure_{out1}, ..., p_{outM}.failure_{(outM)}; \quad (1)$$

WHERE: $p_{(in1)}$ to $p_{(inN)}$ and $p_{(out1)}$ to $p_{(outM)}$ to be the input and output ports of a simple component respectively. Furthermore, $failure_{(in1)}$ to $failure_{(inM)}$ and $failure_{(out1)}$ to $failure_{(outM)}$ to be failure mode associated with the input and output ports respectively. Table 4 describes different failure modes and their description.

4.2.1. CHESSIoT to FLA model transformation

The *Annotated CHESSIoT model* ⑥, produced by the system expert as previously explained, gets automatically transformed

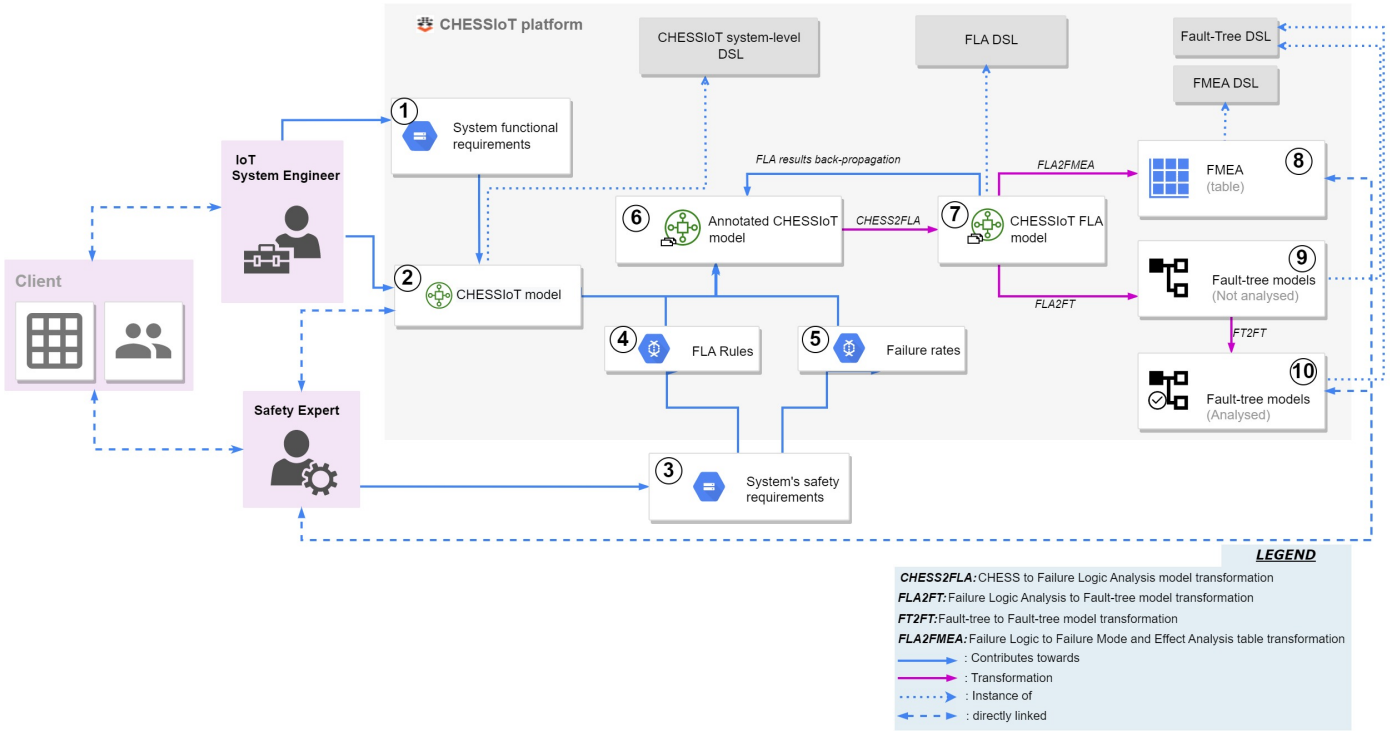


Figure 5: The proposed safety analysis process

Table 4: Failure types

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 out of range in an undetectable way
`Omission`	No output is provided
`Commission`	An output is provided when not expected

by means of the *CHESS2FLA* model transformation to produce the *CHESSIoT FLA model* (7). During CHESS2FLA transformation a composite component is systematically decomposed down to fundamental sub-functions. Then, each lower-level fundamental sub-function is systematically evaluated to identify all its potential failure modes. In addition to that, the behavior of the entire system is automatically determined only from the composition of its elements, including the top-level system's failure probabilities.

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.

The presented approach extends the CHESS-FLA by allowing the description of the failure behavior of a sub-component in the absence of an input port (for example, an IoT sensor). For instance, Sensors may produce erroneous values for numerous reasons, including wrong calibration, harsh environmental conditions, and degradation over time[13]. In such cases of internal failure of a first-class element, no need to have an input port as the failure was initiated internally from the component. Furthermore, when the failure logic definition is done, the user can go ahead with the definition of the probability occurrence of the basic component's failure events, which is afterward employed throughout the analysis.

4.2.2. *Generation of fault-tree models*

The system FTs are generated through a series of model-to-model transformations developed with the Epsilon Transformation Language (ETL) [56]. The process starts by instantiating many FT objects equal to the number of failures propagating to the system's output port(s). At this stage, each error propagating to the system's output port(s) is represented in its corresponding FT. Note that when a "noFailure" condition is propagated to the output, it is disregarded, indicating that the system could mitigate such an event.

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 AND gate is used to indicate a failure transformation from one type to the other as it goes from an input to an output port of a component, while

an OR gate depicts a failure propagation situation in which the same input failure propagated to the output port without a change in type. This is also the case when one or more outputs fail from distinct components and are passed to the input of the following component. An example rule shown in equation 2, a simple component inner failure transformation is presented, where more than two input failure expressions from distinct input ports (i.e $p_{(in1)}$ to $p_{(inN)}$) are transformed to a single failure at the output port ($p_{(out)}$).

$$p_{(in1)}.failure_1, ..., p_{(inN)}.failure_N \rightarrow p_{(out)}.failure_{(out)}; \quad (2)$$

Taking the rule in 2 as an example, Figure 6 illustrates the transformation mapping mechanism. As demonstrated, 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. Furthermore, the logical "AND" gate was used because all of the input failure expressions had to occur in order to fulfill the failure on the output port.

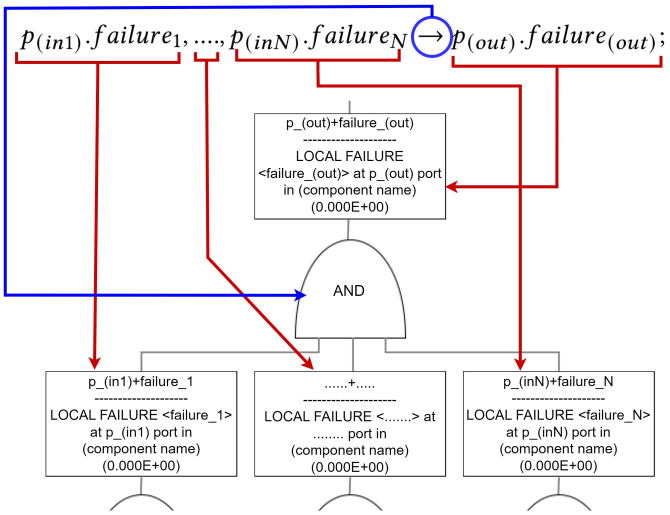


Figure 6: Expression 2 corresponding tree

As a general rule, intermediate events are created and populated into the FT based on the type of failure expression and the structure of the component to which they are allocated. The FT population is a recursive transformation process in which the transformation has access to ports from a component, ports to rules, rules to expressions, and expressions back to the components. At this point, the only stopping case is when the transformation encounters an internal or injected failure case. When the transformation hits a component with no failure behavior set, an underdeveloped event is assigned.

4.2.3. Qualitative analysis

The FT qualitative analysis is conducted using an *FT2FT* model-to-model transformation (ie: transformation from (9) to (10)) in which new FT representations only include the essential representations. During the qualitative analysis process, actions such as removing internal component failure propagation and removing external component-to-component failure propagation representations are performed. In addition, the process also involves the removal of root cause event redundancies. For instance, a failure can be initiated from a single source and passes through different propagation paths until it reaches the output port(s). If, in all paths, no transformation occurred, 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.

For example, taking the generated FT branch shown in Figure 7, 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. The going doing to a single branch of our interest, from event ***2*** with *"omission"* at the input port to event ***1*** with *"commission"* at the output port indicates internal failure transformation. So in such case, event ***1*** and its following gate are kept permanently while event ***2*** is kept temporally for future analysis. Going down to event ***3*** with *"omission"* to ***2*** with *"omission"* which is a component to component failure propagation, so event 3 will be permanently removed.

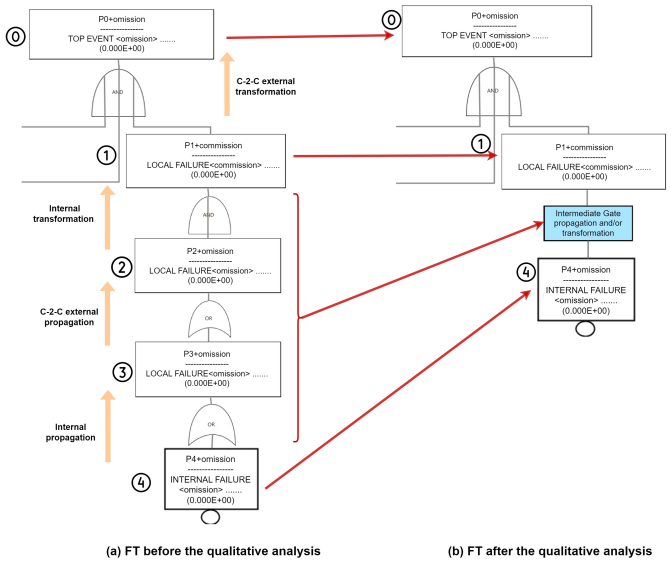


Figure 7: Qualitative transformation example (a) before, (b) after

Next, we remain with event ***4*** with *"omission"* to ***2*** with *"omission"* which is propagation as well, so because event ***4*** is a basic event so event ***2*** will be removed instead. Finally, the whole omitted part of the tree will be substituted by the feed-forward intermediate gate to enhance the readability of the FT. The final version of the FT is provided in Figure 7(b). The internal failure leading to an *"omission"* at the output port of a given component had transformed into an *"commission"* at some point in the system, which when combined with the other two failure sources had to cause a top failure event with an "omission" failure type.

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 [57]), it does provide a much shorter and more readable FT that still reflects the goal for the analysis.

4.2.4. Quantitative analysis

The quantitative probabilistic analysis is meant to calculate the system-level (top event) failure rate automatically. In the proposed approach, the user can 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 and the safety experts. The probability calculation follows a widely used formula for conducting a logical output of an "AND" or an "OR" gates in the FT [58, 52, 59]. The output of an "AND" gate means that the output event will only happen when a combination of independent events occurs simultaneously. On the other hand, the output of an "OR" gate implies that the output event will occur if any 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. 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 in Equation 3:

$$P_{out} = \begin{cases} \prod_{in=1}^{N} P_{in} & \text{for an } AND \text{ gate} \\ 1 - \prod_{in=1}^{N} (1 - P_{in}) & \text{for an } OR \text{ gate} \end{cases} \quad (3)$$

4.3. Model-based design and development

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 shows the support of the complete design and development phase.

The software development process initiates with the user creating functional and behavioral models that conform to the software metamodel shown in Fig. 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

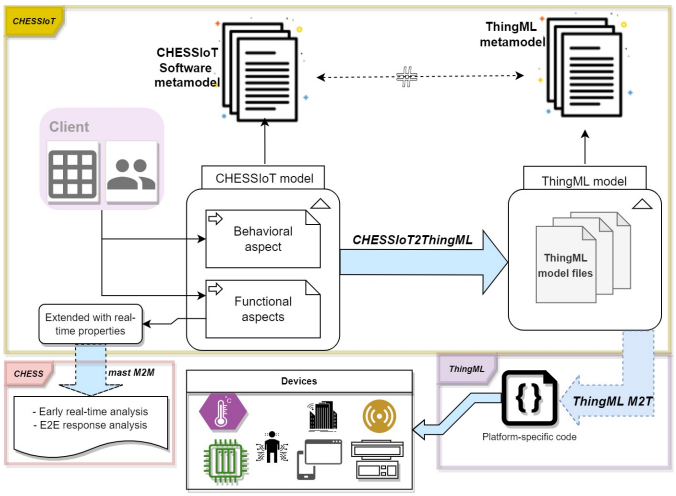


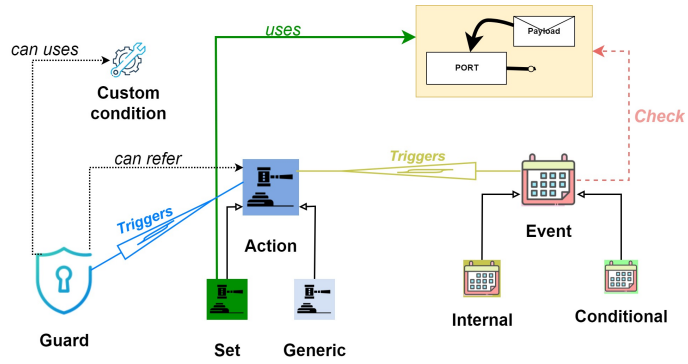
Figure 8: Software development process

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 [53]. However, it does provide specific details regarding the design strategy and code generation approach supported by the CHESSIoT tool.

4.3.1. 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 sub-components, and their interconnections. This process employs *component structure diagrams* that adhere to a component-to-connector design methodology [9]. 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. Figure 9 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 9a, 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



(a) Event & action relationship

(b) State and state transition

Figure 9: Behavioral modeling semantics

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 9a depicts the basic activities that need to be fulfilled from one state to the other.

Figure 9b 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. 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.

4.3.2. The CHESSIoT to ThingML transformation

The CHESSIoT2ThingML transformation process is done through model text transformations written in Acceleo[6]. 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) [60].

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 [9]. 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 [9]. 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 [60], Micro-aerial vehicle platform as well as the Arduino Yùn IoT-based projects [9].

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 10, 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 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 5 depict the CHESSIoT2ThingML transformation mappings implemented by the developed CHESSIoT to ThingML code generator.

IoTPorts are used to support the communication between two or more components by exposing or requiring the interfaces

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

```
[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 10: 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 5: CHESSIoT2ThingML transformation mapping

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 11 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 11: 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 12: 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 12, 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.

4.4. 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[7] and Kubernetes[8] 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 [61]. 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 [62].

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 13. 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. 4.4.1, we present the approach for designing the deployment plan. Second, in Sec. 4.4.2, we delve into the design approach for service provisioning. Finally, in Sec. 4.4.3, we describe the approach for generating deployment artifacts.

4.4.1. 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 13, commences with the user defining the system's deployment model. This model,

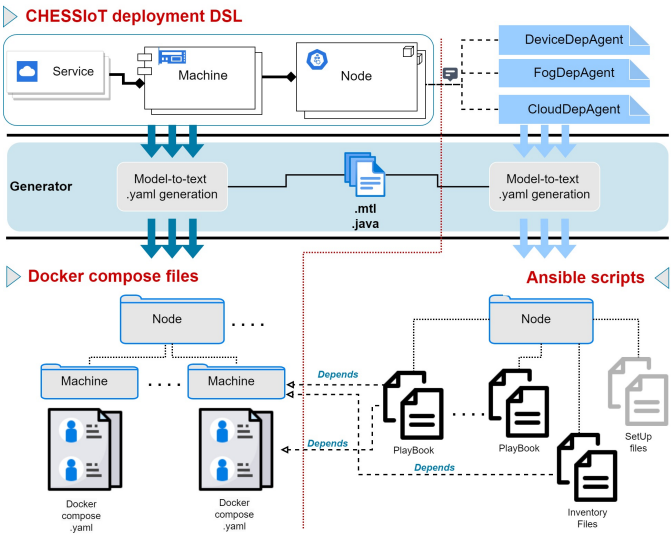


Figure 13: Deployment design process

which aligns with the metamodel discussed in Sec. 4.1.3, primarily focuses on the interconnections between computing nodes, machines, and the IoT services they host. 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. 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 [9] 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

[7] https://www.docker.com/
[8] https://kubernetes.io/
[9] https://zookeeper.apache.org/

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.

4.4.2. 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?"*

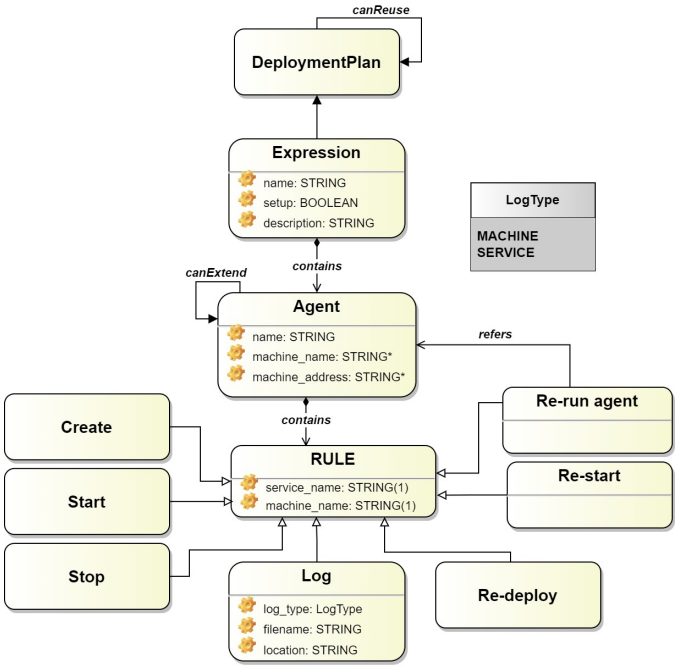


Figure 14: Service provisioning metamodel

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

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 6: CHESSIoT2Ansible transformation mapping

can be translated into any target configuration language for the desired environment. The abstract syntax for the service provisioning language is illustrated in Figure 14.

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 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 1: Run-tine Service provisioning definition example

4.4.3. *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. 4.1.3 and the concepts in Sec. 4.4.1, each node is transformed into a series of docker-compose files targeting each of the machines.

A Docker-Compose file[10], 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).

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[11] 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 [17]. 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 13, 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 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 6 is duly followed.

5. 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 5.1, we present the safety analysis of the system.

[10] https://www.docker.com/

[11] https://www.ansible.com/

Section 5.2 focuses on the system development, specifically addressing the modeling and code generation aspects. Lastly, in Section 5.3, 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 15 provides an overview of the high-level structure of the system implemented in this study.

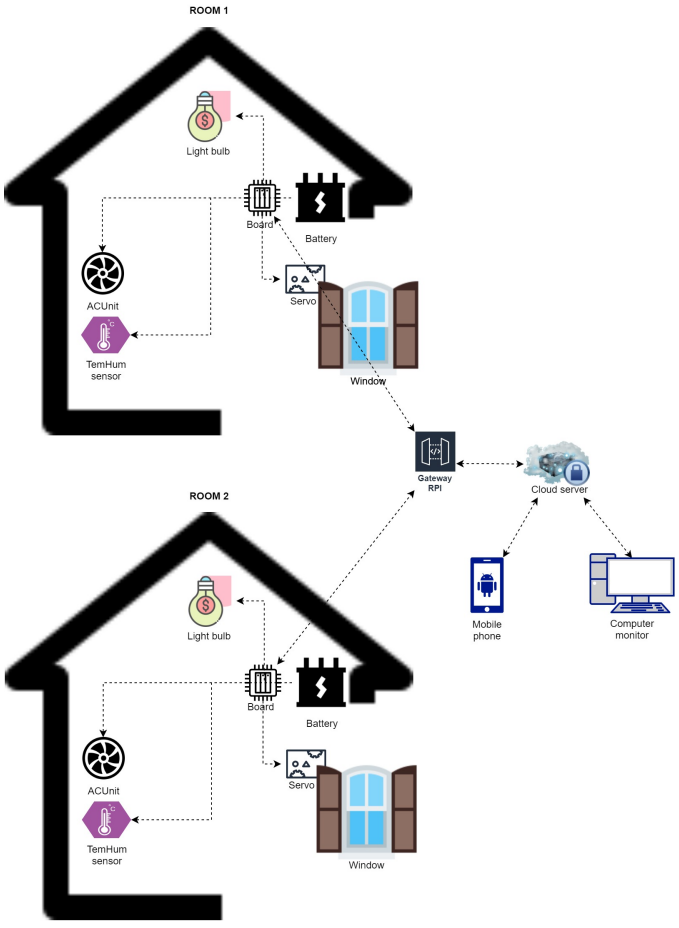


Figure 15: Home Automation System

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.

5.1. HAS system 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 16 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 requiring high power (e.g., window motors). Such considerations are beyond the scope of this study.

Figure 16 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

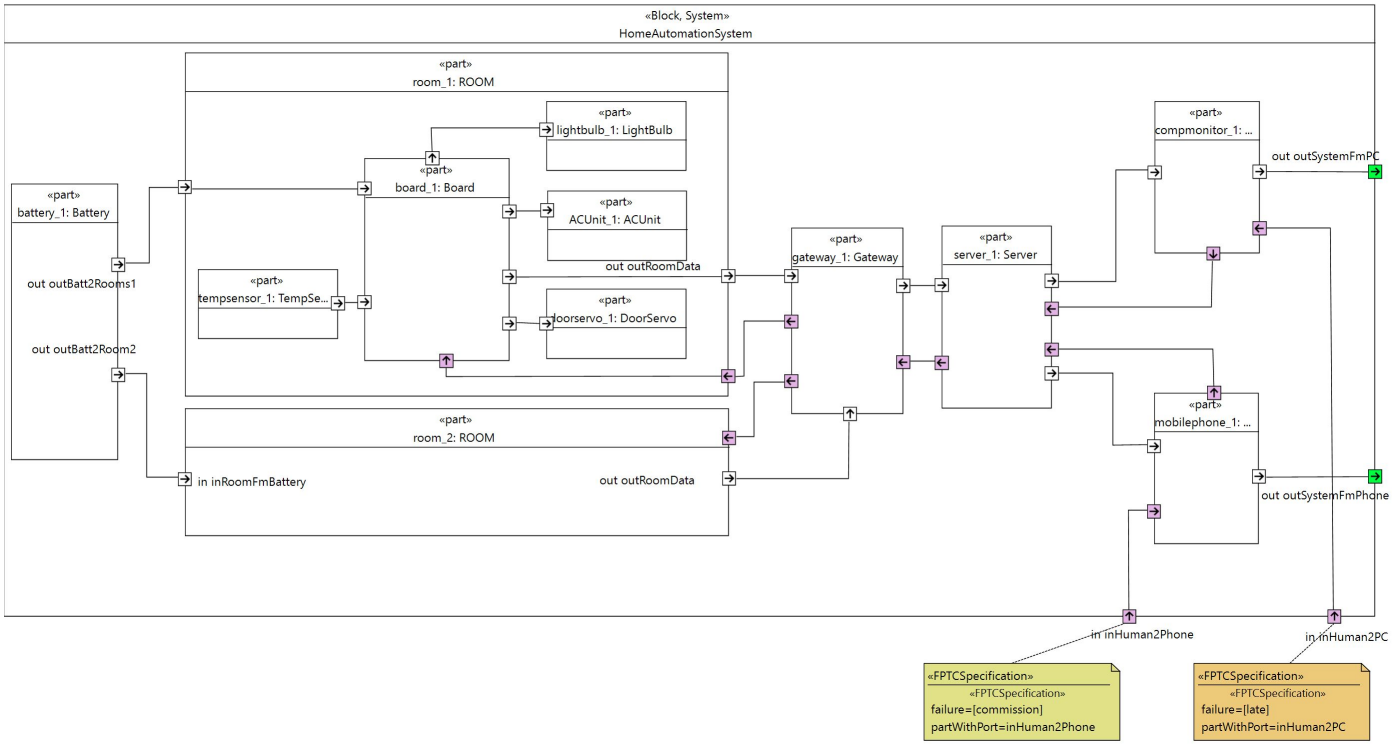


Figure 16: Home automation system model

requirement for the 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 16 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 16

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 4 and 5, 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 [63].

$$FLA : (*) \rightarrow outSensor2Board.\textbf{omission} \tag{4}$$

$$FLA : (*) \rightarrow outSensor2Board.\textbf{valueCoarse} \tag{5}$$

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 [64]. To maintain simplicity, we have set a default probability value of $4 \cdot 10^{-5}$ for all basic failure

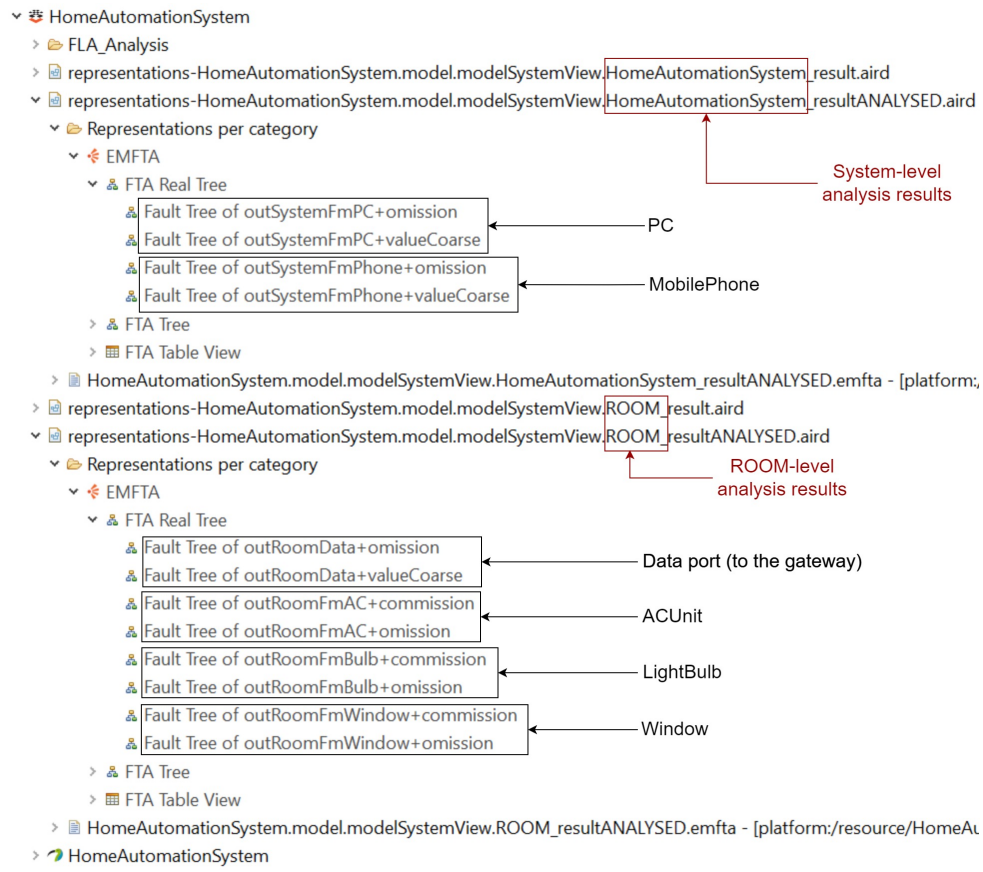


Figure 17: Analysis results

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.

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 17. 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. 18, Fig. 19, and Fig. 20, present generated and analyzed FTs for both at the Room level as well as at the system level.

Figure 18 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 battery"*. Based on the shown fault tree in Fig. 18, 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 19, 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

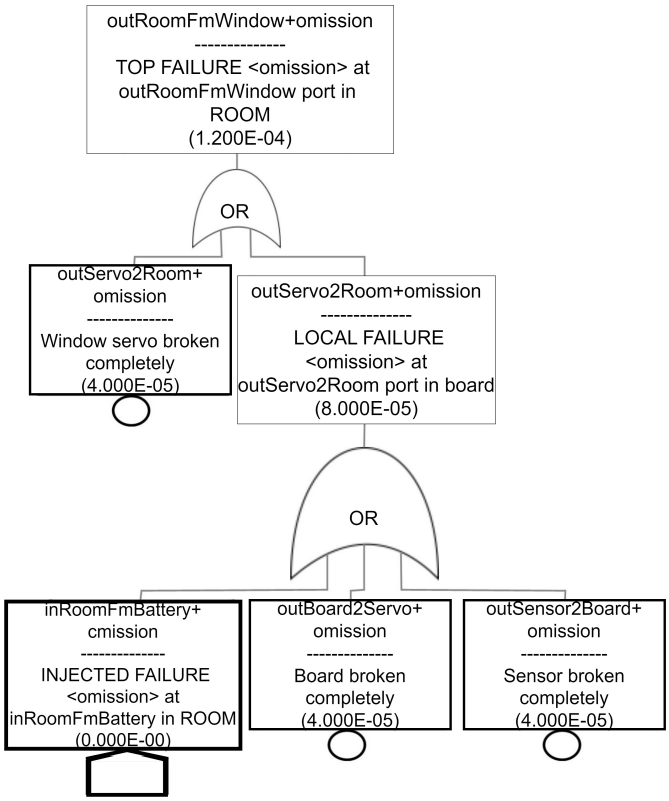


Figure 18: Room-level FT diagram: *Window not working (omission at the window output port)*

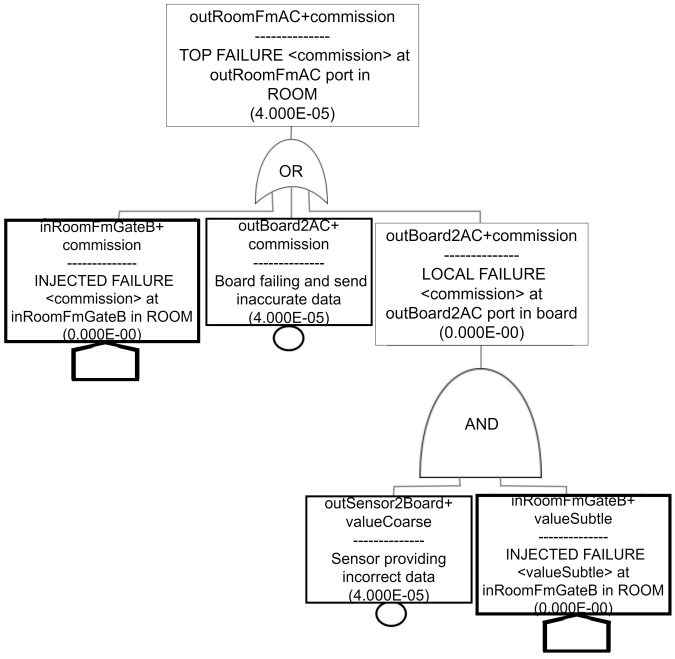


Figure 19: Room-level FT diagram: *ACUnit turn on and off when not expected*

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. 20. 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, 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.

5.2. *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 4.3. 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 16, we refer to it as a point of reference.

5.2.1. *Behavior modeling*

The internal component model representation of the room is depicted in Figure 21. 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 (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,

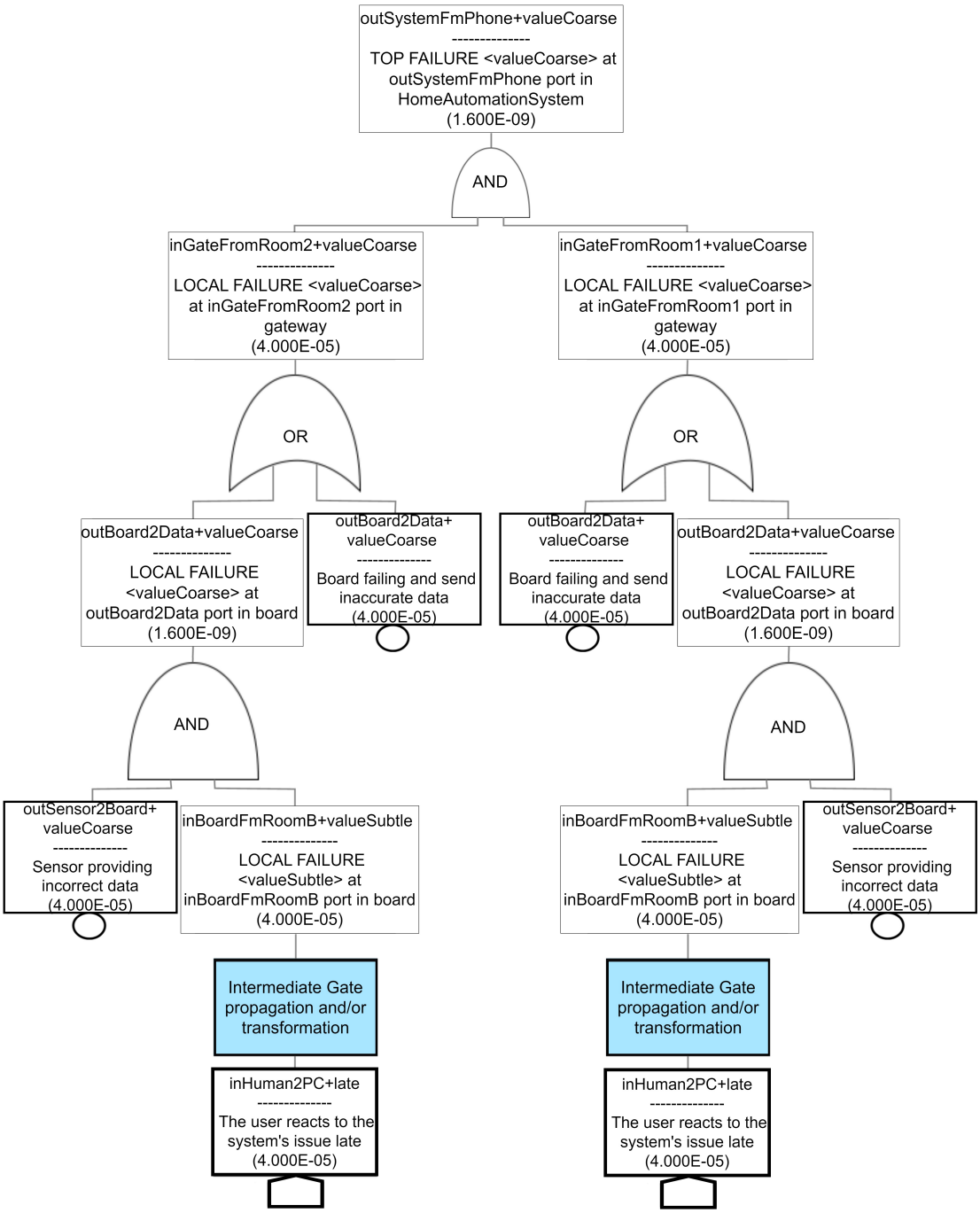


Figure 20: System-level FT diagram: *Mobile phone displays inaccurate data scenario*

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. 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 21).

The board serves as a central computing component in the process, coordinating all connected elements by reusing previously defined actions, payload, and guards. Figure 22 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.

As we can see from the board state machine figure 23, three main states are defined, namely "*IDLE*", "*AC_OFFBulbOFFWindowClose*" as well as the "*AC_ONBulbONWindowOpen*" states are 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 22. 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).

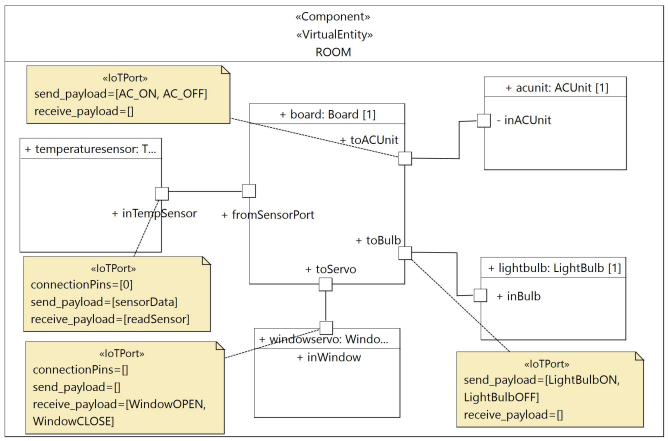


Figure 21: Room internal composite structure

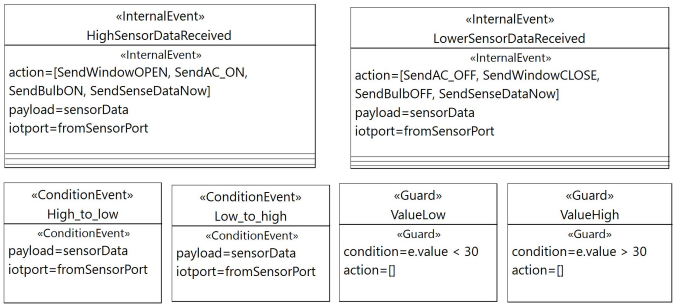


Figure 22: Portion of the Board event, action, guard specification

5.2.2. *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

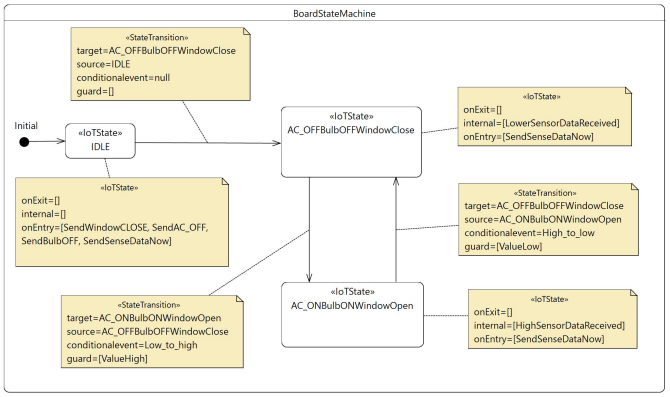


Figure 23: Board state machine

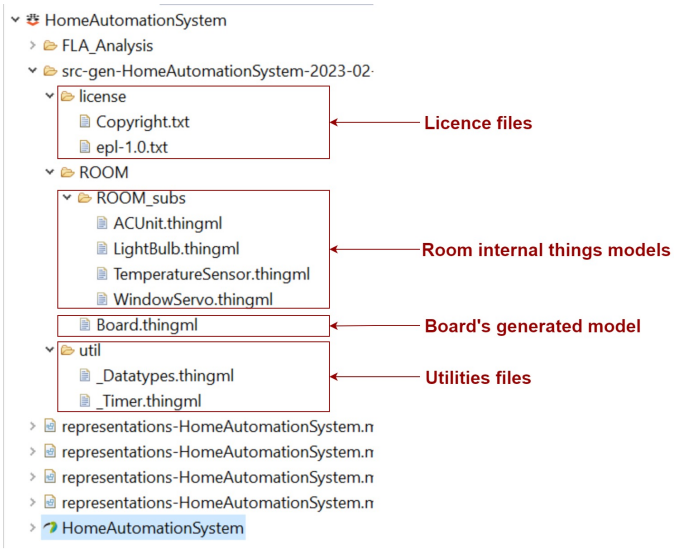


Figure 24: Generated ThingML models

transformation process follows the mapping presented in 5. Figure 24 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 25 depicts the generated ThingML model of the board mapped back to the state machine diagram presented in Figure 23. As we can see from Figure 24, 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

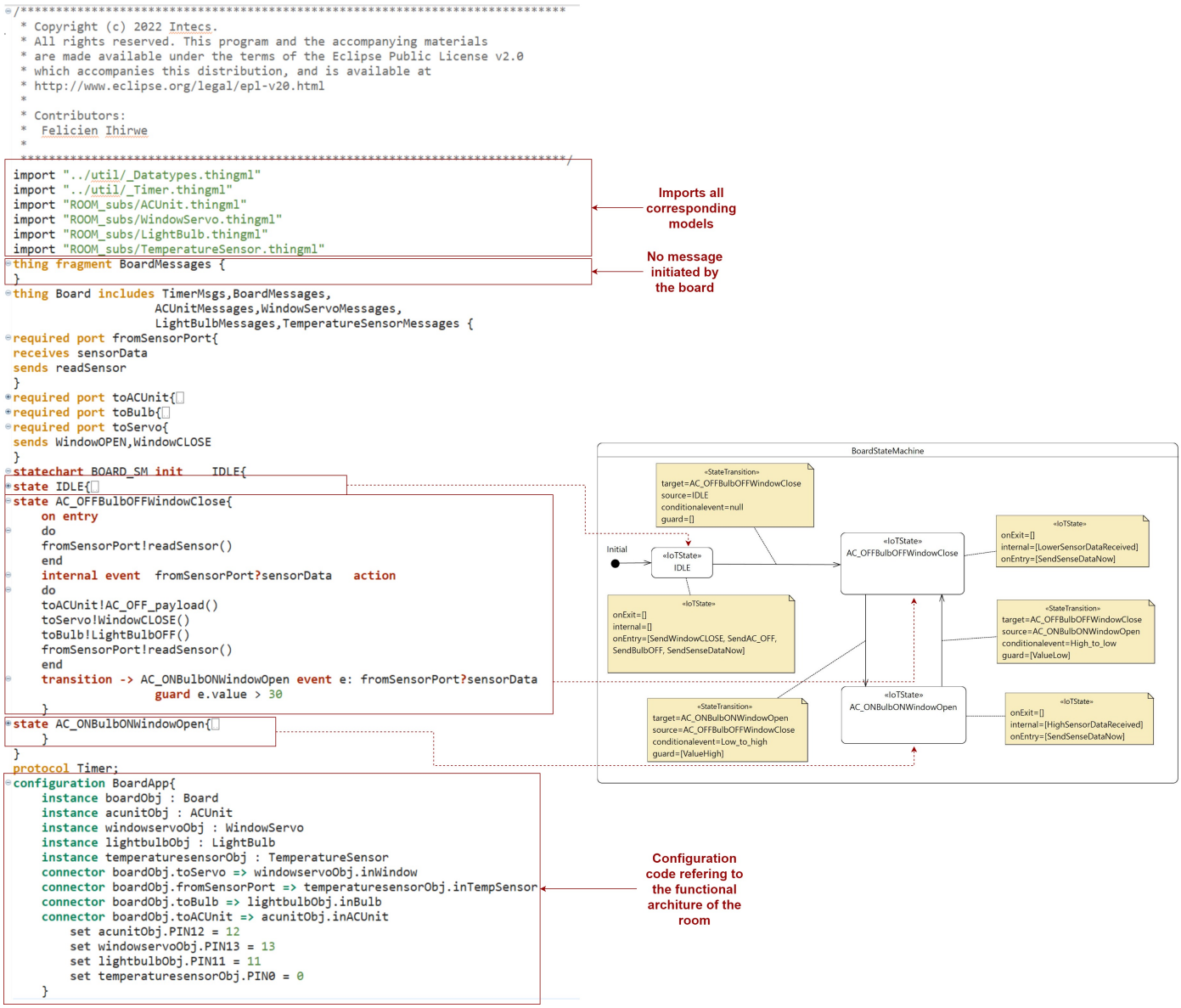


Figure 25: Generated Board's ThingML model mapped to the state machine diagram

component-to-connector architecture [9] that aligns with the Room's internal structure. As depicted in the bottom-left section of Figure 25, 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 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 [12] and the code worked perfectly as expected. The full example with all the materials is online available at https://github.com/fihirwe/HomeAutomationSystem.git

5.3. Deployment and service provisioning

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

5.3.1. 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

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

the MQTT broker, as well as the on the cloud i.e-, a web server running a Node-RED dashboard instance. Figure 26 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.

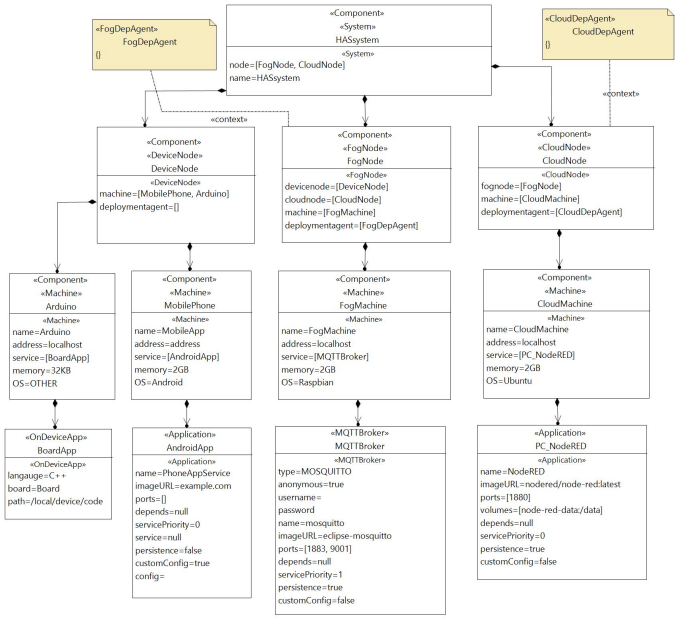


Figure 26: HAS system deployment plan

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 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. 26, 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 27.

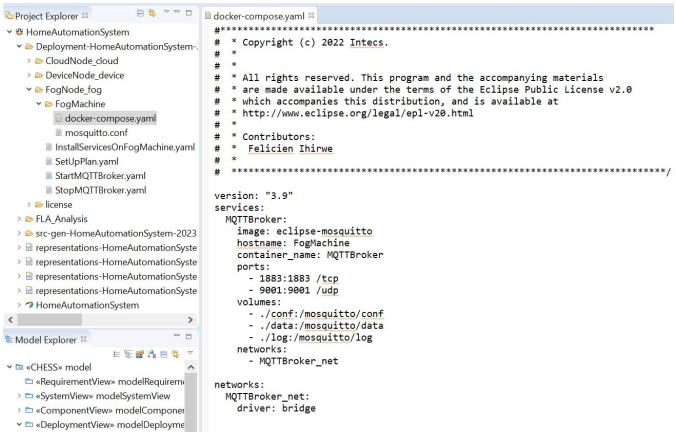


Figure 27: Generated deployment configuration Fog

5.3.2. *HAS runtime service provisioning*

As shown in the deployment plan model in Figure 26, 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.

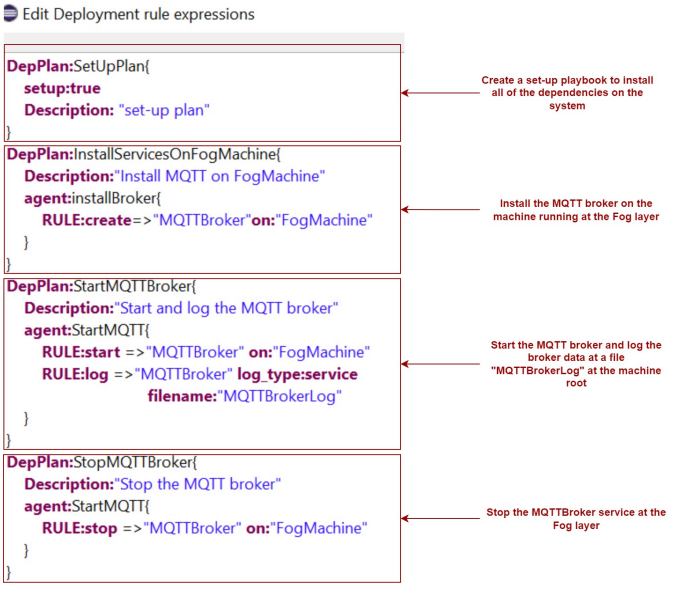


Figure 28: FogDepAgent rules

In Figure 28, 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 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. 27).

6. 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 29 provides an overview of the study's outcomes, displaying the percentage of supporting and non-supporting aspects in both contexts.

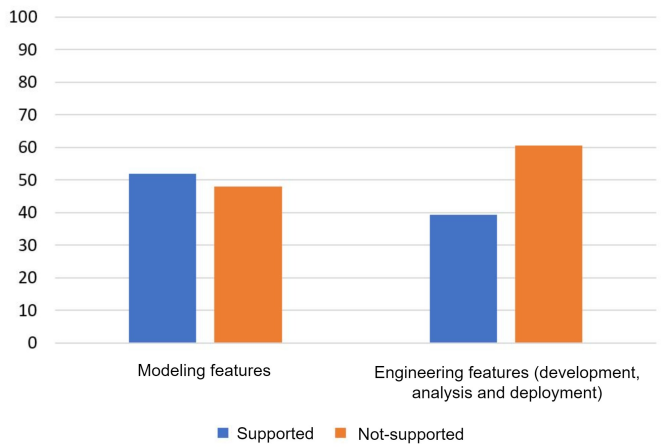


Figure 29: Overall comparative supporting results

According to Figure 29, when considering both modeling and engineering non-supporting aspects, the average gap where CHESSIoT could potentially contribute is 54.34%. While SimulateIoT [12] 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. 3.3, we observed that SimulateIoT [12] 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 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 [53]. Due to the complexity and scale of IoT systems, physical replication and testing become challenging [65]. 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 [53, 54].

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 [2], 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 [66]. 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 means for testing generated software to support safety analysis results that reflect real-world conditions.

7. CONCLUSION AND FUTURE WORK

This article presented CHESSIoT, a model-driven environment designed for developing multi-layered IoT systems. The approach covers all three primary layers of IoT systems, namely edge, fog, and cloud, and is demonstrated through a home automation case study. The article showcases the capability of CHESSIoT to generate fully functional ThingML source models, which can be further transformed into platform-specific code for deployment on low-level

IoT devices. The support for qualitative and quantitative safety analyses using Fault-Tree models is also highlighted. Additionally, a deployment modeling approach and run-time service provisioning approach are discussed. Through comparative analysis and discussions, the article identifies the potential areas where CHESSIoT can make significant contributions.

In the future, our aim is to incorporate testing support for the generated code, which can help identify any potential missing safety rules and enhance overall reliability. We also plan to enhance the qualitative safety analysis mechanism by enabling the generation of minimal cut-set Fault Trees, allowing for more precise analysis. Moreover, we intend to make the system failure mode abstraction method easily customizable to different domains, improving flexibility. Lastly, we are eager to apply the development and deployment approach of CHESSIoT to real-world industrial use cases, leveraging practical scenarios to refine further and improve the platform.

CONFLICT OF INTEREST STATEMENT

Authors have no conflict of interest.

SOURCE CODE

The full code and the instruction on how to use CHESSIoT can be found at https://github.com/fihirwe/CHESSIoT-features

FUNDING INFORMATION

This work has received funding from the Lowcomote project under European Union's Horizon 2020 research and innovation program under the Marie Skłodowska-Curie grant agreement n^{o}813884.

AUTHORS BIOGRAPHY

Felicien Ihirwe. is a doctorate student in the Department of Information Engineering, Computer Science, and Mathematics (DISIM) at the University of L'Aquila, Italy. He's part of Intecs Solutions's Model-driven research and development team under the Innovation Technology Service lab (Pisa-Italy). He graduated from Carnegie Mellon University with a Master's degree in Electrical and Computer Engineering. His research interest involves Model-based software engineering (MBSE), System analysis, Low-code Engineering (LCE), and the Internet of Things (IoT). His current work includes the development of domain-specific languages and methods to support the modeling, development, safety analysis, and deployment of engineering IoT systems by employing low-code software engineering principles.

Davide Di Ruscio is an Associate Professor at the Department of Information Engineering Computer Science and Mathematics of the University of L'Aquila. His main research interests are related to several aspects of Software Engineering, Open Source Software, and Model-Driven Engineering (MDE) including domain-specific modeling languages, model transformation, model differencing, coupled evolution, and recommendation systems. He has published more than 150 papers in various journals, conferences, and workshops on such topics. He is a member of the steering committee of the International Conference on Model Transformation (ICMT), of the Software Language Engineering (SLE) conference, and of the Seminar Series on Advanced Techniques & Tools for Software Evolution (SATTOSE). He is on the editorial board of the International Journal on Software and Systems Modeling (SoSyM), IEEE Software, the Journal of Object Technology, and the IET Software journal.

Simone Gianfranceschi is a Chief Technology Officer at Intecs Solutions. He has over 22 years of industrial experience working in System Engineering, Aerospace, and Telecommunication domains. INTECS provides leading-edge software technologies to support the major European and Italian organizations in the design and implementation of advanced electronic systems for the defense, space, and civilian markets. INTECS provides leading-edge software technologies to support the major European and Italian organizations in the design and implementation of advanced electronic systems for the defense, space, and civilian markets. He has co-authored different research papers in various journals, conferences, and workshops on topics related to Global Earth Observation Information Services, Wireless Sensor Network applications, and GEOSS Interoperability. He has been responsible for over 10 international projects namely IM-Set, MASS-ENV, and ODISSEO (to name a few).

Alfonso Pierantonio is a Full Professor at the Department of Information Engineering Computer Science and Mathematics of the University of L'Aquila. His research interests span many areas of Model-Driven Engineering, Language Engineering, Low-Code, and Software Engineering with a specific emphasis on co-evolution, bi-directionality, and complex model management. He has published more than 160 articles in scientific journals and conferences and has been on the organizing committee of

several international conferences, including MoDELS and STAF. Alfonso is Editor-in-Chief of the Journal of Object Technology and on the editorial and advisory board of Software and System Modeling, and Science of Computer Programming. He has been PC Chair of ECMFA 2018, General Chair of STAF 2015, and is a Steering Committee member of the ACM/IEEE MoDELS. Alfonso will be General Chair of MoDELS 2023, to be held in Västerås, Sweden. He is a co-principal investigator of several research and industrial projects.

References

[1] S. Munirathinam, Chapter six - industry 4.0: Industrial internet of things (iiot), in: P. Raj, P. Evangeline (Eds.), The Digital Twin Paradigm for Smarter Systems and Environments: The Industry Use Cases, Vol. 117 of Advances in Computers, Elsevier, 2020, pp. 129–164. doi:https://doi.org/10.1016/bs.adcom.2019.10.010.
URL https://www.sciencedirect.com/science/article/pii/S0065245819300634

[2] M. Bauer, M. Boussard, N. Bui, J. De Loof, C. Magerkurth, S. Meissner, A. Nettsträter, J. Stefa, M. Thoma, J. W. Walewski, IoT Reference Architecture, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 163–211. doi:10.1007/978-3-642-40403-0_8.
URL https://doi.org/10.1007/978-3-642-40403-0_8

[3] A. Taivalsaari, T. Mikkonen, On the development of iot systems, in: 2018 Third International Conference on Fog and Mobile Edge Computing (FMEC), 2018, pp. 13–19. doi:10.1109/FMEC.2018.8364039.

[4] A. K. Muroor Nadumane, Models and Verification for Composition and Reconfiguration of Web of Things Applications, Theses, Université Grenoble Alpes [2020-....] (Dec. 2020).
URL https://theses.hal.science/tel-03188299

[5] A. Salihbegovic, T. Eterovic, E. Kaljic, 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), 2015, pp. 996–1001. doi:10.1109/MIPRO.2015.7160420.

[6] M. Mernik, J. Heering, A. M. Sloane, When and how to develop domain-specific languages, ACM Comput. Surv. 37 (4) (2005) 316–344. doi:10.1145/1118890.1118892.

[7] M. Fowler, Domain Specific Languages, 1st Edition, Addison-Wesley Professional, 2010.
URL https://dl.acm.org/doi/10.5555/1809745

[8] F. Ciccozzi, R. Spalazzese, MDE4IoT: Supporting the Internet of Things with Model-Driven Engineering, in: Intelligent Distributed Computing X, 2017. doi:10.1007/978-3-319-48829-5_7.

[9] N. Harrand, F. Fleurey, B. Morin, K. E. Husa, ThingML: A Language and Code Generation Framework for Heterogeneous Targets, MODELS '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 125–135. doi:10.1145/2976767.2976812.

[10] R. Nicholson, T. Ward, D. Baum, X. Tao, D. Conzon, E. 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), 2019, pp. 239–246. doi:10.1109/WorldS4.2019.8903975.

[11] D. Conzon, M. R. A. Rashid, X. Tao, A. Soriano, R. Nicholson, E. 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), 2019, pp. 1–8. doi:10.1109/CCCS.2019.8888136.

[12] J. A. Barriga, P. J. Clemente, E. Sosa-Sanchez, A. E. Prieto, SimulateIoT: Domain specific language to design, code generation and execute IoT simulation environments, IEEE Access 9 (2021) 92531–92552. doi:10.1109/ACCESS.2021.3092528.

[13] J. C. Kirchhof, B. Rumpe, D. Schmalzing, A. Wortmann, Montithings: Model-driven development and deployment of reliable IoT applications, Journal of Systems and Software 183 (2022) 111087. doi:https://doi.org/10.1016/j.jss.2021.111087.

[14] A. Debiasi, F. Ihirwe, P. Pierini, S. Mazzini, S. 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,, INSTICC, SciTePress, 2021, pp. 262–269. doi:10.5220/0010269702620269.

[15] B. Gallina, M. A. Javed, F. U. Muram, S. Punnekkat, A Model-Driven Dependability Analysis Method for Component-Based Architectures, in: 2012 38th Euromicro Conference on Software Engineering and Advanced Applications, 2012, pp. 233–240. doi:10.1109/SEAA.2012.35.

[16] K. Görlach, F. Leymann, Dynamic service provisioning for the cloud, in: 2012 IEEE Ninth International Conference on Services Computing, 2012, pp. 555–561. doi:10.1109/SCC.2012.30.

[17] G. Shah, Ansible Playbook Essentials, Packt Publishing Ltd, 2015.

[18] D. Di Ruscio, R. Eramo, A. Pierantonio, M. Bernardo, V. Cortellessa, A. Pierantonio, Model Transformations, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 91–136. doi:10.1007/978-3-642-30982-3_4.

[19] Node-RED, Node-RED: Low-code programming for event-driven applications, https://nodered.org/, last accessed May 2020 (2020).

[20] F. Ihirwe, A. Indamutsa, D. Ruscio, S. Mazzini, 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), IEEE Computer Society, Los Alamitos, CA, USA, 2021, pp. 73–82. doi:10.1109/MODELS-C53483.2021.00018.

[21] B. Costa, P. F. Pires, F. C. Delicato, Modeling IoT applications with sysml4IoT, in: 2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2016, pp. 157–164. doi:10.1109/SEAA.2016.19.

[22] B. Costa, P. F. Pires, F. C. Delicato, W. Li, A. Y. Zomaya, Design and analysis of IoT applications: A model-driven approach, in: 2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech), 2016, pp. 392–399. doi:10.1109/DASC-PICom-DataCom-CyberSciTec.2016.81.

[23] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, Nusmv 2: An opensource tool for symbolic model checking, in: E. Brinksma, K. G. Larsen (Eds.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 359–364. doi:10.1007/3-540-45657-0_29.

[24] K. Thramboulidis, F. Christoulakis, Uml4IoT-a uml-based approach to exploit IoT in cyber-physical manufacturing systems, Comput. Ind. 82 (C) (2016) 259–272. doi:10.1016/j.compind.2016.05.010.

[25] O. M. Alliance, Lightweight machine to machine technical specification, Technical Specification OMA-TS-LightweightM2M-V1 (2013).

[26] C. M. de Farias, I. C. B. et al, Comfit: A development environment for the internet of things, Future Generation Computer Systems 75 (2017) 128–144. doi:https://doi.org/10.1016/j.future.2016.06.031.

[27] H. Muccini, M. Sharaf, Caps: Architecture description of situational aware cyber physical systems, in: 2017 IEEE International Conference on Software Architecture (ICSA), 2017, pp. 211–220. doi:10.1109/ICSA.2017.21.

[28] M. Sharaf, M. Abusair, R. Eleiwi, Y. Shana'a, I. Saleh, H. Muccini, Modeling and code generation framework for IoT, in: System Analysis and Modeling. Languages, Methods, and Tools for Industry 4.0, Springer International Publishing, Cham, 2019, pp. 99–115. doi:https://doi.org/10.1007/978-3-030-30690-8_6.

[29] S. Dhouib, A. Cuccuru, F. L. Fèvre, S. Li, B. Maggi, I. Paez, A. Rademarcher, N. Rapin, J. Tatibouet, P. Tessier, S. Tucci, S. Gerard, Papyrus for IoT – a modeling solution for IoT (2016).

[30] L. Erazo-Garzón, P. Cedillo, G. Rossi, J. Moyano, A domain-specific language for modeling IoT system architectures that support monitoring, IEEE Access 10 (2022) 61639–61665. doi:10.1109/ACCESS.2022.3181166.

[31] OMG, Object constraint language (ocl), version 2.3.1.
URL https://www.omg.org/

[32] ISOGRAPH, Fault tree analysis in reliability workbench, last accessed May 2022 (2022).
URL https://www.isograph.com/

[33] I. Silva, R. Leandro, D. Macedo, L. A. Guedes, A dependability evaluation tool for the internet of things, Computers & Electrical Engineering 39 (7) (2013) 2005–2018. doi:https://doi.org/10.1016/j.compeleceng.2013.04.021.

[34] Y. Chen, Z. Zhen, H. Yu, J. 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). doi:10.3390/s17010153.

[35] L. Xing, M. Tannous, V. M. Vokkarane, H. Wang, J. Guo, Reliability modeling of mesh storage area networks for internet of things, IEEE Internet of Things Journal 4 (6) (2017) 2047–2057. doi:10.1109/JIoT.2017.2749375.

[36] A. Åkesson, G. Hedin, N. Fors, R. Schöne, J. Mey, Runtime modeling and analysis of IoT systems, 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, 2020. doi:10.1145/3417990.3421397.

[37] J. Parri, F. Patara, S. Sampietro, E. Vicario, A framework for model-driven engineering of resilient software-controlled systems, Computing 103 (04 2021). doi:10.1007/s00607-020-00841-6.

[38] F. Mhenni, N. Nguyen, J.-Y. Choley, Automatic fault tree generation from sysml system models, in: 2014 IEEE/ASME International Conference on Advanced Intelligent Mechatronics, 2014, pp. 715–720. doi:10.1109/AIM.2014.6878163.

[39] B. Alshboul, D. C. Petriu, B. Alshboul, D. C. Petriu, Automatic derivation of fault tree models from sysml models for safety analysis, Journal of Software Engineering and Applications 11 (2018) 204–222.
URL https://doi.org/10.4236/jsea.2018.115013

[40] A. H. d. A. Melani, G. F. M. d. Souza, Obtaining fault trees through sysml diagrams: A mbse approach for reliability analysis, in: 2020 Annual Reliability and Maintainability Symposium (RAMS), 2020, pp. 1–5. doi:10.1109/RAMS48030.2020.9153658.

[41] N. Yakymets, H. Jaber, A. 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,, INSTICC, SciTePress, 2013, pp. 210–214. doi:10.5220/0004346902100214.

[42] T. Prosvirnova, AltaRica 3.0: a Model-Based approach for Safety Analyses, Theses, Ecole Polytechnique (Nov. 2014).
URL https://pastel.archives-ouvertes.fr/tel-01119730

[43] F. Durán, A. Krishna, M. Le Pallec, R. Mateescu, G. Salaün, Models and analysis for user-driven reconfiguration of rule-based IoT applications, Internet of Things 19 (2022) 100515. doi:https://doi.org/10.1016/j.IoT.2022.100515.

[44] I. Alfonso, K. Garcés, H. Castro, J. Cabot, Modeling self-adaptative IoT architectures, in: 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), 2021, pp. 761–766. doi:10.1109/MODELS-C53483.2021.00122.

[45] B. Negash, T. Westerlund, A. M. Rahmani, P. Liljeberg, H. Tenhunen, DoS-IL: A Domain Specific Internet of Things Language for Resource Constrained Devices, Procedia Computer Science 109 (2017) 416–423. doi:https://doi.org/10.1016/j.procs.2017.05.411.

[46] F. Li, M. Vögler, M. Claeßens, S. Dustdar, Towards automated IoT application deployment by a cloud-based approach, in: 2013 IEEE 6th International Conference on Service-Oriented Computing and Applications, 2013, pp. 61–68. doi:10.1109/SOCA.2013.12.

[47] N. Ferry, P. Nguyen, H. Song, P.-E. Novac, S. Lavirotte, J.-Y. Tigli, A. Solberg, Genesis: Continuous orchestration and deployment of smart IoT systems, in: 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), Vol. 1, 2019, pp. 870–875. doi:10.1109/COMPSAC.2019.00127.

[48] M. Hussein, S. Li, A. Radermacher, Model-driven development of adaptive IoT systems, in: 2017 MODELS Satellite Event, Vol. 2019, CEUR-WS, Austin, United States, 2017, pp. 17–23.
URL https://hal-cea.archives-ouvertes.fr/cea-01843007

[49] M. Panunzio, T. Vardanega, A component-based process with separation of concerns for the development of embedded real-time software systems, Journal of Systems and Software 96 (2014) 105–121. doi:https://doi.org/10.1016/j.jss.2014.05.076.

[50] B. Andrew, G. Rahul, Mqtt version 3.1.1 plus errata 01 [online], last Accessed: September 2020 (December 2015).
URL http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html

[51] A. Taivalsaari, T. Mikkonen, A roadmap to the programmable world: Software challenges in the iot era, IEEE Software 34 (1) (2017) 72–80. doi:10.1109/MS.2017.26.

[52] A. Cheliyan, S. Bhattacharyya, Fuzzy fault tree analysis of oil and gas leakage in subsea production systems, Journal of Ocean Engineering and Science 3 (1) (2018) 38–48. doi:https://doi.org/10.1016/j.joes.2017.11.005.

[53] F. Ihirwe, D. Di Ruscio, S. Mazzini, A. Pierantonio, Towards a Modeling and Analysis Environment for Industrial IoT systems, in: STAF Workshops, 2021, pp. 90–104.
URL http://ceur-ws.org/Vol-2999/messpaper1.pdf

[54] F. Ihirwe, D. Di Ruscio, S. Mazzini, A. Pierantonio, A domain specific modeling and analysis environment for complex IoT applications, arXiv preprint arXiv:2109.09244 (2021).

[55] M. Wallace, Modular architectural representation and analysis of fault propagation and transformation, Electronic Notes in Theoretical Computer Science 141 (3) (2005) 53–71, proceedings of the Second International Workshop on Formal Foundations of Embedded Software and Component-based Software Architectures (FESCA 2005). doi:10.1016/j.entcs.2005.02.051.

[56] D. S. Kolovos, R. F. Paige, F. A. C. Polack, The epsilon transformation language, in: A. Vallecillo, J. Gray, A. Pierantonio (Eds.), Theory and Practice of Model Transformations, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 46–60. doi:https://doi.org/10.1007/978-3-540-69927-9_4.

[57] H. Ren, X. Chen, Y. Chen, Chapter 6 - fault tree analysis for composite structural damage, in: H. Ren, X. Chen, Y. Chen (Eds.), Reliability Based Aircraft Maintenance Optimization and Applications, Aerospace Engineering, Academic Press, 2017, pp. 115–131. doi:https://doi.org/10.1016/B978-0-12-812668-4.00006-X.

[58] S. Markulik, M. Šolc, J. Petrík, M. Balážiková, P. Blaško, J. Kliment, M. Bezák, Application of fta analysis for calculation of the probability of the failure of the pressure leaching process, Applied Sciences 11 (15) (2021). doi:10.3390/app11156731.

[59] R. Ferdous, F. Khan, B. Veitch, P. R. Amyotte, Methodology for computer aided fuzzy fault tree analysis, Process Safety and Environmental Protection 87 (4) (2009) 217–226. doi:https://doi.org/10.1016/j.psep.2009.04.004.

[60] B. Morin, N. Harrand, F. Fleurey, Model-based software engineering to tame the IoT jungle, IEEE Software 34 (1) (2017) 30–36. doi:10.1109/MS.2017.11.

[61] F. Alkhabbas, R. Spalazzese, M. Cerioli, M. Leotta, G. Reggio, On the deployment of IoT systems: An industrial survey, in: 2020 IEEE International Conference on Software Architecture Companion (ICSA-C), 2020, pp. 17–24. doi:10.1109/ICSA-C50368.2020.00012.

[62] G. Karataş, F. Can, G. Doğan, C. Konca, A. Akbulut, Multi-tenant architectures in the cloud: A systematic mapping study, in: 2017 International Artificial Intelligence and Data Processing Symposium (IDAP), 2017, pp. 1–4. doi:10.1109/IDAP.2017.8090268.

[63] F. Ihirwe, Home automation system failure logic behavior rules (Jan. 2023). doi:10.5281/zenodo.7575082.

[64] F. Afsharnia, Failure rate analysis, in: A. Ali (Ed.), Failure Analysis and Prevention, IntechOpen, Rijeka, 2017, Ch. 7. doi:10.5772/intechopen.71849.
URL https://doi.org/10.5772/intechopen.71849

[65] D. T. Nguyen, C. Song, Z. Qian, S. V. Krishnamurthy, E. J. M. Colbert, P. McDaniel, Iotsan: Fortifying the safety of iot systems, in: Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies, CoNEXT '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 191–203. doi:10.1145/3281411.3281440.
URL https://doi.org/10.1145/3281411.3281440

[66] C. E. Internationale, IEC 61025:2006 - Fault tree analysis (FTA) English and French language, IEC Standards Online, 2006.
URL https://webstore.iec.ch/publication/4311