

Towards a modeling and analysis environment for industrial IoT systems

Felicien Ihirwe^{1,2}, Davide Di Ruscio², Silvia Mazzini¹ and Alfonso Pierantonio²

¹*Innovation Technology Services Lab, Intecs Solutions S.p.A, Pisa, Italy*

²*Department of Information Engineering Computer Science and Mathematics, University of L'Aquila, L'Aquila, Italy*

Abstract

The development of Industrial Internet of Things systems (IIoT) requires tools robust enough to cope with the complexity and heterogeneity of such systems, which are supposed to work in safety-critical conditions. The availability of methodologies to support early analysis, verification, and validation is still an open issue in the research community. The early real-time schedulability analysis can help quantify to what extent the desired system's timing performance can eventually be achieved. In this paper, we present CHESIIoT, a model-driven environment to support the design and analysis of industrial IoT systems. CHESIIoT follows a multi-view, component-based modeling approach with a comprehensive way to perform event-based modeling on system components for code generation purposes employing an intermediate ThingML model. To showcase the capability of the extension, we have designed and analysed an Industrial real-time safety use case.

Keywords

CHESIIoT, Model-driven engineering, Industrial IoT, Schedulability analysis

1. Introduction

The complexity of industrial IoT systems is deemed to increase due to the growing need for data acquisition, processing, and storage techniques. Moreover, the rapid penetration of advanced machine-to-machine communications in cyber-physical systems triggers even more complexity in IoT production systems [1]. The design complexity of such systems has to consider different layers, including the behaviour of the building components, their inter-connectivity, not to mention the message heterogeneity [2, 3].

Model-Driven Engineering (MDE) aims at supporting software development and analysis by promoting the adoption of models as first-class citizens. Performing early analysis on intended systems can help discover how they will behave once deployed. Furthermore, the quantitative results from the conceptual analysis of such systems can provide theoretical support for optimizing system architectures and parameters earlier enough [4]. The challenges present in IoT systems validation and verification (V&V) also poses a significant gap in certifying such

International workshop on MDE for Smart IoT Systems (MeSS'21) co-located with STAF2021, Virtual conference, Bergen, Norway. June 21-25, 2021


✉ felicien.ihirwe@intecs.it (F. Ihirwe); davide.diruscio@univaq.it (D. D. Ruscio); silvia.mazzini@intecs.it (S. Mazzini); alfonso.pierantonio@univaq.it (A. Pierantonio)

🌐 <http://people.disim.univaq.it/diruscio/> (D. D. Ruscio); <https://pierantonio.io> (A. Pierantonio)

🆔 0000-0002-4463-6268 (F. Ihirwe); 0000-0002-5077-6793 (D. D. Ruscio); 0000-0002-5231-3952 (A. Pierantonio)



© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

systems. Different industrial model-driven approaches such as [5, 6, 7, 8] try to cope with such challenges, but we see it as not yet enough.

To cope with the complexity and heterogeneity issues at the levels of system design, operational, and deployment, we propose CHESSIoT. CHESSIoT is a multi-view modeling environment to support the design, development, and analysis of industrial IoT systems. Currently, CHESSIoT is being developed on top of the CHESS tool[9], a mature modeling and analysis environment for the development of complex industrial systems [10]. Since the CHESS tool has been developed to meet industrial needs [11], we decided to rely on it to support industrial IoT development. In addition to CHESS, CHESSIoT offers an IoT-specific modeling infrastructure where the user can specify the system's structural and behavioural architectures, perform different real-time analyses and generate platform-specific code.

To guaranty a fully decoupled extension, CHESSIoT introduced the *"IoT sub-view"*, and once applied in all design stages, the user will benefit from a dedicated IoT-specific modeling infrastructure consisting of specific diagrams and palettes. The CHESSIoT methodology follows a component-based approach where all system components and their internal behaviours are decomposed separately. The specified components can be annotated with extra-functional properties for analysis purposes or used to generate platform-specific code. The inner or external events can be defined using component state machines. The development of CHESSIoT is still in its early phases but with very significant progresses concerning its design capabilities. CHESSIoT will benefit from the system model-based verification and validation resources provided by CHESS.

Moreover, CHESSIoT employs ThingML tool [12] to generate code. ThingML is a well-proven software modeling tool aligned with UML (state-charts and components) and an imperative platform-independent action language to construct the intended IoT applications [13]. ThingML model can compile and generate code in different languages such as C/C++, Java, JavaScript, Arduino, and Go. In this paper, we overview an industrial IoT real-time safety use case to showcase the current capabilities of CHESSIoT. Interested readers can access the complete source code of the developed extension publicly available on GitHub repository.¹

The main contributions of this paper can be summarized as follows:

- The CHESSIoT modeling environment to support the development and analysis of industrial IoT systems;
- Overview of the envisioned CHESSIoT2ThingML transformation that will enable the generation of target platform code.
- Description of a simple use-case on industrial real-time safety to showcase the analysis capabilities for the IoT.
- Showcase the real-time schedulability analysis that can be performed on the CHESSIoT model of the presented use case.

This paper is structured as follows: Section 2 examines the related work. Section 3 gives a concise background of the CHESS and ThingML platforms. Section 4 presents the technical specification of the proposed CHESSIoT extension. Section 5 describes the simple industrial use-case. Lastly, Section 6 concludes the paper and makes an overview of our future work.

¹https://github.com/feliIhirwe/ChessIoT_Dev

2. Related Work

Modeling and analysis of the Internet of Things is not a new topic in general, several approaches have already been studied and validated, but few of them focus on the Industrial domain. This section will look briefly at some related research to our system and discuss their differences and correspondences. For the sake of the topic of interest, we will only cover the MDE approaches that extend uses Eclipse Papyrus² modeling as an underlying environment.

In [14], the CHESS tool has been used to model the life-cycle of scalable and distributed intelligent IoT applications. Although this was a great start towards using CHESS for modeling IoT systems, it is generic considering the specificity of IoT applications. In our approach, we emphasize decoupling the modeling environment for IoT elements to the whole CHESS platform. In [15] the authors presented a Cloud and Model-based IDE for the Internet of Things tool (COMFIT) to target the wireless sensor networks (WSN) applications for IoT. The COMFIT modeling environment is built on top of Papyrus, and it presents a simple multi-view environment to model the system's requirement, structural and behavioural aspects. Ciccozzi et al. presented the MDE4IoT tool [6] developed to support the modeling of self-adaptation of connected Emergent Configurations (ECs) referred to as Things in the IoT domain. The authors present a simple smart street light case to highlight and validate their approach characterized by the multi-view and separation of concern capability of the MDE4IoT tool. The run-time adaptations are meant to be performed automatically by specific in-place model transformations that modify the source models. Although this approach is closely related to CHESIoT, the paper still does not address any analysis aspects of the system.

In [16] the authors introduced Papyrus4IoT, a modeling tool developed under the S3P project. The authors presented a modeling methodology that uses the UML use case diagrams and early system conceptual system specification requirements. Later the designer can define process specification definition, functional, operational platform, and finally, the deployment is done by allocating the system's functional blocks to the device processing units. Unlike [16], our emphasis is given on separation of concerns and supporting analysis that has not been addressed in [16]. Conzon et al. [17] have presented the BRAIN-IoT framework, an integrated modeling tool to ease rapid prototyping of intelligent cooperative IoT systems based on shared models. The constructed models are transformed into XML format before being uploaded to the BRAIN-IoT marketplace for future reuse.

In [5], the authors introduced UML4IoT domain-specific modeling language to tackle the Industrial Automation Thing (IAT) domain. In UML4IoT, the system's components are transformed into IATs by IoTwrapper and later integrated into the IoT-based industrial automation environment. The RESTful paradigm is adopted for enhancing the connectivity with third-parties resources. Although their approach focuses more on the industrial use case, it differs from CHESIoT concerning the multi-view and analysis support. Authors in [7] introduced a SysML4IoT modeling and analysis platform derived from SysML to support the IoT domain. Following the IoT-A reference architectural reference in [18], the authors introduced the SysML2NuSMV translator. The tool has later been extended by [19] to support the design, and the usage of the public/subscribe paradigm to model the communication relationships with other systems.

²<https://www.eclipse.org/papyrus/>

Although the authors suggest the system analysis by considering only the quality of service through model checking, we see it as not sufficient in the industrial IoT domain.

From the above discussion, we can see that most of the proposed UML-based modeling approaches for industrial systems focus more on design and code generation; we see a significant lack in the analysis mechanism of IoT systems in general with the industrial case in particular. We think that this is an excellent direction for CHESSIoT to explore to address such challenges.

3. Background

The following section gives a brief background on the CHESS tool and the motivation behind its extension. Furthermore, we will briefly present the ThingML tool and its development infrastructure and why it is crucial for CHESSIoT code generation.

3.1. CHESS development environment

The CHESS tool is a mature cross-domain model-driven tool developed on top of the Eclipse Papyrus environment to support the modeling and analysis of dependable systems [9]. The CHESSML modeling language provided by CHESS is an integrated modeling language profiled from OMG standard languages: UML, SysML, and MARTE under the Papyrus modeling environment [20]. The CHESSML language was designed to support the component-based development methodology. Emphasis is given to specify the non-functional properties of the modeled components, including critical properties such as time predictability, isolation, transparency, and other real-time and dependability-related characteristics [14]. CHESS tool provides a multi-view modeling environment where each view has its own underlined constraints that enforce its specific privileges on model entities and properties that can be manipulated. Depending on the current stage of the design process, CHESS sub-views are adopted to enhance specific design properties or steps of the current process. Different tools, plugins, and languages have been integrated into CHESS to support model validation, model checking, real-time, and dependability analysis.

Even though CHESS has been successfully applied in different application domains such as Avionics [21], Automotive [22], Space [23], Telecommunication [24], and Petroleum [25, 26], its current status does not explicitly provide modeling capabilities for the IoT domain. Consequently, we aim at extending the existing modeling and analysis infrastructure starting from software modeling infrastructure.

3.2. ThingML framework

ThingML is amongst the most popular domain-specific model-driven engineering tools for the IoT domain. It comprises a custom textual modeling language, a supporting modeling tool, and advanced code generator capabilities. The ThingML language combines well-proven software modeling constructs aligned with UML (state-charts and components) and an imperative platform-independent action language to construct the intended IoT applications [13]. ThingML code generator targets many popular programming languages such as C/C++, Java, and Javascript, and about ten different target platforms (ranging from tiny 8bit microcontrollers to servers) and ten different communication protocols [12].

Several research approaches have been shown interest in applying ThingML as their modeling or code generation framework. To mention a few, in [27] ThingML has been used to generate code for CAPS, an architecture-driven modeling framework for the development of IoT Systems. In [28] ThingML has been used to specify the behaviour of distributed software components, and later it has been extended with mechanisms to monitor and debug the execution flow of a ThingML program. In [29], CypriIoT tool used and extended the ThingML modeling language to model the behaviour of IoT things and as a code generator for platform-specific code.

Although ThingML framework is very mature and looks very promising, it can not be one shoe fit for all aspects that involve IoT systems design and development. For instance, the ThingML framework does not provide the means to conduct any system-related analyses that are very important in the Industrial IoT domain. There is also a lack of system-level design and validation in ThingML. CHESSIoT envisions having a consolidated and fully automated environment where users can combine both ThingML and CHESS technologies for industrial IoT systems development.

4. Proposed Approach

The CHESSIoT tool has been developed on top of the recently released CHESS 1.0.0 tool [9]. At design time, CHESSIoT enforces that all the components and elements be IoT specific and follows the already existing CHESS modeling methodologies. In particular, systems are specified in terms of a component-based design and employing a multi-view paradigm. The CHESSIoT extension consists of four different profiles depending on the specific view and need for the particular task at hand. As an addition to the domain-specific CHESS views, CHESSIoT adds the *IoT sub-view* to permit the user to activate services and palettes related to the IoT domain. The CHESSIoT design extensions are available throughout the whole views provided by CHESS as long as the user activates the *IoT sub-view* from the toolbar. Figure 1 shows the high-level representation of the CHESSIoT extension concerning its development infrastructure, design time, and run-time modeling functionalities. As shown at the top-right of Figure 1 CHESSIoT consists of four profiles that are singularly described as follows.

CHESSIoTSystem profile serves to cover the system-level design aspects in which the system's main blocks and their interconnections are defined. This is done in the CHESS *System View* invoking the *IoT sub-view*. As an extension of the SysML blocks, the IoT high-level blocks and their corresponding flow-ports are defined and later annotated with formal properties as contracts. In this regard, the user benefits from the system level validation and verification, contract refinements, parameterized architecture, and trade-off analysis infrastructure, all provided by the CHESS tool. For the sake of the paper scope, we do not cover such aspects in this paper.

The **CHESSIoTSoftware profile** provides users with modeling constructs to describe the IoT software components and their behaviours. The software design is done in the *Component view*. In this regard, the user can decompose the system's software components and sub-components provided by specific palettes. In CHESSIoT the component behaviours are defined by using state machines. Figure 2 shows the CHESSIoT software profile meta-model, which consists of the following elements:

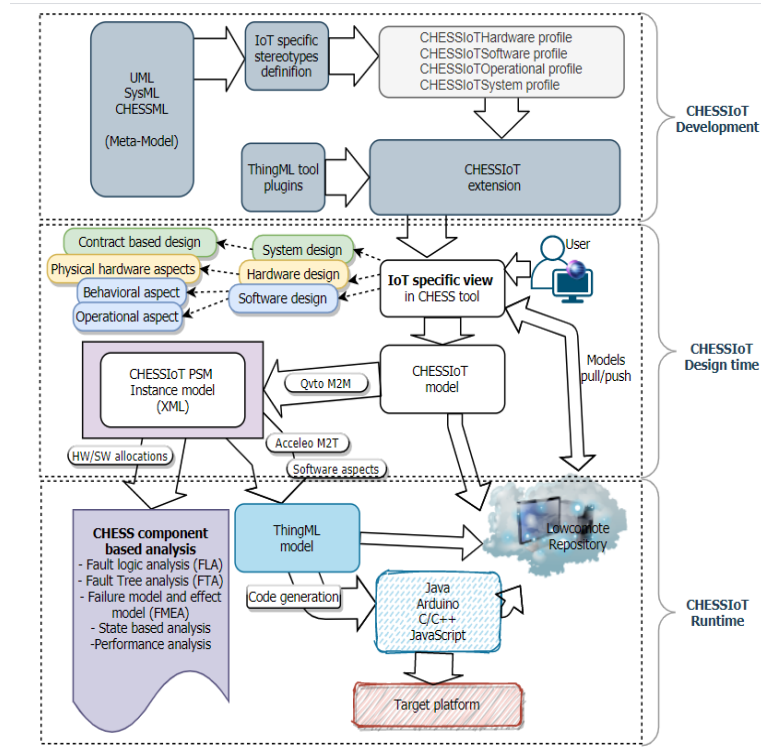


Figure 1: CHESSIoT approach

- The *Virtual entity* permits the definition of digital representations of physical objects in which the systems resources are allocated. This entity consistently links to its corresponding "Physical Entity" to be defined in the deployment view when performing the hardware designs;
- The *Virtual board* enables the specification of IoT computing boards on which the application will run. This component is a sub-component to *VirtualEntity* and can be connected or have reference to one or many other virtual boards. The only crucial property it poses is its state which later gets defined;
- The "*IoTElement*" represents any other IoT Thing that comprises a system aside of being a *VirtualEntity* or a *VirtualBoard*. The internal software architecture can be specified using other internal sub-elements and the connectors and ports, while their behavioural specification is defined using custom states, events, and actions. Furthermore, the *IoTElements* decomposed as IoT system sub-components are then connected to define the system's main components. To this level, the user can instantiate as many components as possible to meet any system complexity wanted.

In CHESSIoT, we have introduced a flexible event-based modeling mechanism. Normally, the UML state machine events and actions modeling process is complex and tricky when done using the normal Papyrus infrastructure. In our case, the *IoTEvents* and *IoTActions* can be modeled separately and later be invoked as many times as possible or by many different *IoTElements* states. An *IoTState* always should be linked to an "*OnEntry*" and an "*OnExit*" events, which in

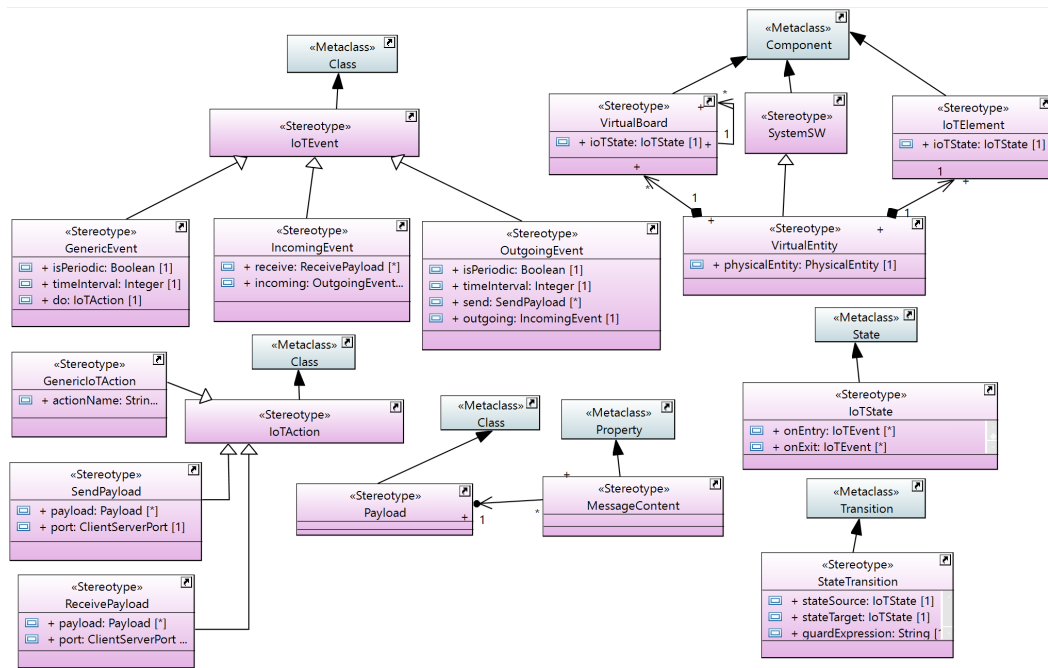


Figure 2: CHESSIoT software meta-model

turn can be either generic, incoming, or an outgoing event. The *IoTAction* element is for sending and receiving payloads, and action types can then be associated with each event in the form of effect. A *payload* is any type of message exchanged between *IoTElements* through ports and can be reused as many times as possible. More information on this is also presented in Section 4.1.

CHESSIoTHardware profile contains a physical deployment representation of the virtual components designed in *Component view*. The hardware modeling activities are performed in *Deployment View*. The hardware design also includes the definition of target platform specifications, such as the number of processors and core units. Most of the elements in this profile rely on MARTE [30].

CHESSIoTOperational profile contains all the information regarding the communication aspects of the system, and it extends the CHESS’s *Component View*. Ideally, the information related to communication mode, servers, communication protocols, and storage resources will be modeled and analysed using this profile. The purpose of adding this part is to support the performance analysis of the involved resource blocks. In the end, CHESSIoT also will allow the user to generate platform-specific code employing the ThingML platform automatically. Lastly, the user will interact with a remote repository by consuming an open API pushing or pulling artifacts. The high-level description of envisioned CHESSIoT to ThingML model transformation is described in the next section.

4.1. The CHESSIoT to ThingML model transformation

Modeling software component in CHESSIoT goes hand in hand with defining its behaviours, resulting in platform-specific code. The CHESSIoT component's semantics differs from the ThingML's to some extent, and that is why the mapping of the elements is needed to solicit an efficient transformation. In the following, we discuss how the different CHESSIoT modeling constructs contribute to the generation of target ThingML elements.

A component: In CHESSIoT, the software components such as *IoTElement*, *VirtualBoard*, *VirtualEntity* are the main modeling elements. They are used to encapsulate the system's main part structure, operations, and behaviours. These components are mapped to ThingML's thing.

Provided/Required port: Ports are used to support the communication between two or more components exposing or requiring the interfaces from other components. In CHESSIoT, the component's messages are passed through the port using the required or provided interface operations. During the transformation, the Required/Provided ports of the components are mapped to the **required/provided port** of a ThingML's thing.

Operation: In CHESSIoT, operations specify the functional behaviour of components. During the transformation, each component's operation is mapped to corresponding Thing's **function**.

Property: Properties represent variable attributes local to a component. The property can be primitive or be an instance of other components. 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 will be mapped to thing's **property**.

Payload: This is a standalone and straightforward object to carry information to be passed between components. The payload will be mapped to a **Message** in the ThingML model. In CHESSIoT, the payload can have zero or many primitive or derived properties to be defined in a message.

IoTState: This serves to keep the component state from its initial participation until its disposal in the system. *IoTState* extends actual UML states but in addition to that, *IoTState* carries information related to what events need to be taken care of at a certain point in time. For instance, *OnEntry* or *OnExit* events are triggered when entering or exiting a state. An IoT state can also trigger an internal event, which corresponds to an internal action to be taken. During the transformation, the IoTState will be mapped to the *ThingML state*, same goes to **State transition** that will also be mapped to their corresponding transition provided by ThingML.

IoTEvent: Events in CHESSIoT are triggered in a different manner depending on the state of the component. An *IoTvent* can be incoming, outgoing, or generic, which means it can come from the inside-out, from the outside, or internal. A *GenericEvent* is an event that gets triggered internally to the component, for example, changing variable value. CHESSIoT events are mapped to corresponding ThingML *event(s)*.

IoTAction: IoTAction(s) can be of different types depending on the kind of action to be performed. For instance, the *SendPayload* action is referred to when an *OutgoingEvent* is triggered to send the payload through a specified port while *ReceivePayload* is used on *IncomingEvents* to receive messages from another components. A *GenericAction* does not require to access the component's ports, for example, changing the component's property value. During the transformation, *IoTActions* will be mapped to corresponding ThingML actions.

State Transition: In CHESSIoT, state transitions enable transiting from the source state to a

target state, abiding the trigger from the guard value. Guard expressions are boolean expressions defined based on state values. They serve to initiate a state transition by checking whether the *OnExit* event has been performed correctly. During the transformation, *State transitions* will be mapped to the corresponding transitions in ThingML.

Table 1
Proposed CHESSIoT2ThingML mapping

CHESSIoT element	ThingML element
Component	Thing
Provided/required port	Provided/required port
Operation	Function
Property	Property
Payload	Message
IoTState/Transition	State/Transition
StateGuards	Guards
IoTEvent/Action	Event/Action

5. A real-time safety use case in the IIoT domain

In this section, an explanatory industrial safety use case is modeled and analysed employing the proposed approach. In the example shown in Figure 3, a basic safety system is proposed. The system objective is to gather information through specific nodes deployed into a modern environment to collect environment data, and in case of an unprecedented issue, the system triggers an alarming mechanism to limit further damages.

In this example, we do not cover the communication layer and the cloud-side designs. This is deemed to be done using the operational profile, which is currently yet fully developed. Besides that, the analyses being performed will be one on the ThingLayer components. More on this is discussed in Section 5.2. In the following, we present the modeling phases of the proposed system.

5.1. Modeling software components

The first step of the proposed approach consists of modeling the node's main components, corresponding types, and the interfaces they implement. We suppose that the system requirements have been modeled in CHESS's requirement view. Modeling the software constructs is done in the component view. Following the approach presented before, the *computing board* in CHESSIoT is defined as a *Virtual board*, other devices such as LEDs, sensors, and buzzer are defined as *IoTElements*. As CHESSIoT follows a component-based approach, only one component needs to be defined, and it can be instantiated and reused as many times as possible.

A *virtual entity* is considered as a physical object where the computing node will be deployed. In other words, they won't implement any interface or perform any action. As described before, the interfaces contain functional operations to be carried out during the communication, and they automatically get applied to the elements that implement them. After defining *Component*,

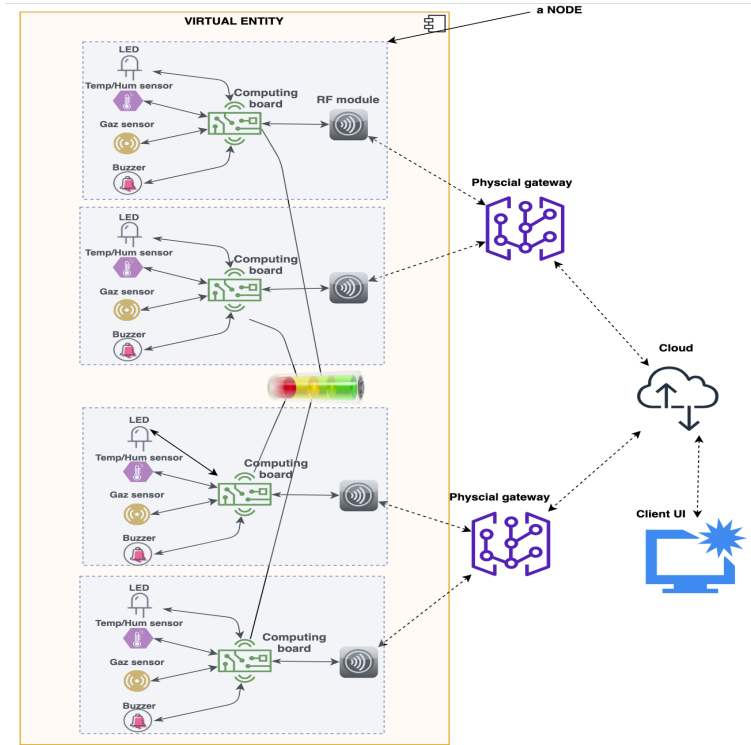


Figure 3: Physical structure of the simple IoT system

ComponentType, *Interfaces* elements and their corresponding *Operations*, each component's internal structure is decomposed. This is done by using the UML composite structure diagram. *IoTElements* such as LED, sensors, and buzzer are considered as sub-components of the *Virtual board*. The internal structure is specified by defining the component's required/provided ports with the corresponding interfaces they expose or require. At this stage, CHESIoT allows modeling the component behavioural aspects using the UML state machine. The component's event, action, and payloads are modeled using the component's inner class diagram and then linked back to their corresponding states. Note that the *IoT sub-view* element provided by CHESIoT needs to be opened here to have the right palette containing the behavioural resources. Figure 4 shows the three different modeling parts of the internal structure of a temperature/humidity sensor.

As shown in Fig. 4, temperature/humidity sensors have only one port which provide the *IsenseHumTemp* interface. The behavioural modeling of it is mainly for code generation purposes. This process has to be done for each defined *IoTElements*. The next step is to model the node, which is a *Virtual board*. At this level, the *IoTElement* can be instantiated as many times as possible to achieve the desired structure of the node. The internal structure of the virtual board can be modeled as shown in Fig. 5. The virtual board also contains required (in red) and provided (in green) ports. The provided ports of the *IoTElements* have to be connected to their corresponding required ports of the board and vice-versa. The ports in yellow represent a port that provides and require an interface. This can apply to the data communication between the

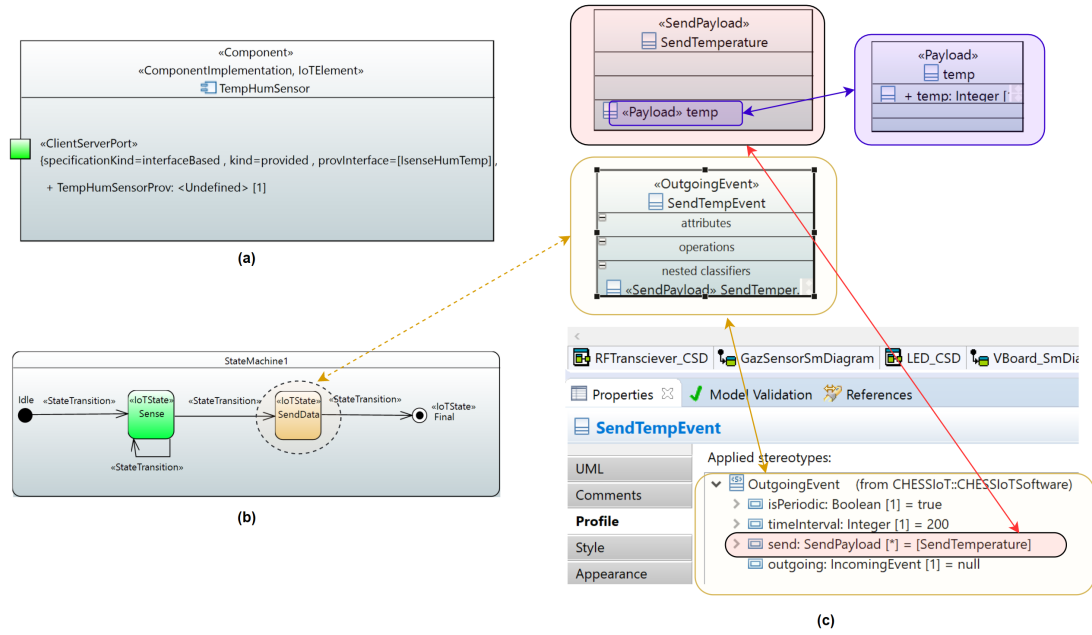


Figure 4: (a) Temp/hum sensor internal structure, (b) State machine, (c) Event, action and payload definition

RFModule and the computing board.

The same as the other elements, the board's behavior specification has to be defined too, but we won't present it for the sake of space. The virtual entity comprises as many virtual board instances as possible and one or more power source instances. In our case, we used four board instances with one power source. The next stage involves specifying different physical characteristics of the target platform, such as the number of processors and number of cores. This part extends the MARTE profile typically and this is done in CHES's deployment view. In our case, the system will be deployed to one *Physical entity* which will comprise four different processors. Three of the processors will run on one core each, while the fourth runs on two cores. This process is performed in *Deployment view*. At this stage, we generated the software and hardware instances that contain the containers and connectors representing components instances and their connectors.

5.2. Real-time schedulability analysis

CHES tool offers the means to perform real-time analyses on the modeled instance model. Schedulability analysis is performed employing MAST [31], a timing analysis tool that relies on the system's component timing requirements. To perform such analysis, the user needs to annotate the real-time temporal logic properties on each component's operations. Those properties include the timing request type (i.e., periodic or sporadic), the worst-case execution time of a request, the priority, and the desired execution deadline. This activity is performed in the CHES's *InstanceView*. CHES also supports the software to hardware component allocation,

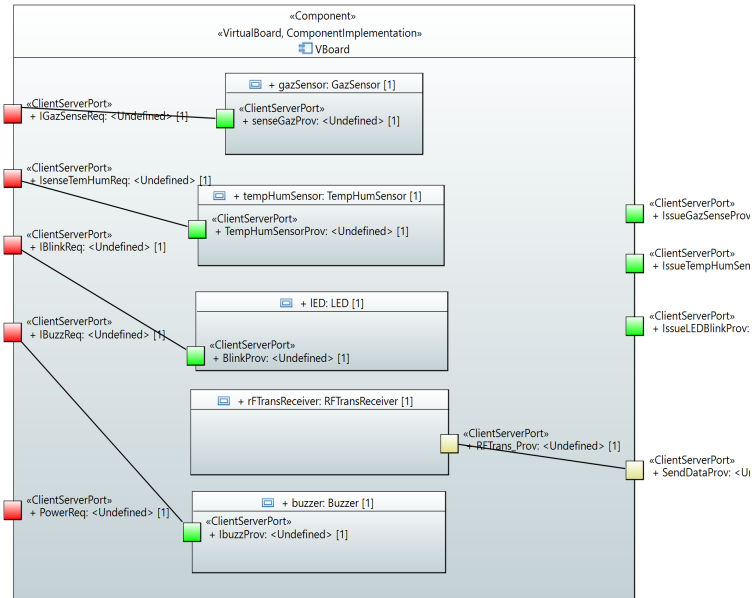


Figure 5: Virtual board internal structure

Schedulability Analysis Report				
Final analysis status: NOT-SCHEDULABLE				
HW Instance	Utilization	Result		
PhysicalEntity_ch_HwProcessor4	50.00%	OK		
PhysicalEntity_ch_HwProcessor3	471.67%	NOT OK: utilization over 100%		
PhysicalEntity_ch_HwProcessor2_core0	471.67%	NOT OK: utilization over 100%		
PhysicalEntity_ch_HwProcessor2_core1	471.67%	NOT OK: utilization over 100%		
PhysicalEntity_ch_HwProcessor1	471.67%	NOT OK: utilization over 100%		
SW Instance	Operation	Response Time	Deadline	Result
VE_System.vBoard.IED	blinkLED	9.223372036854776E16ms	3600ms	NOT OK
VE_System.vBoard2.RFTransReceiver	receive	9.223372036854776E16ms	7000ms	NOT OK
VE_System.vBoard.RFTransReceiver	transmit	9.223372036854776E16ms	5000ms	NOT OK
VE_System.vBoard.gazSensor	senseGaz	9.223372036854776E16ms	600ms	NOT OK
VE_System.battery	power	9.223372036854776E16ms	20000ms	NOT OK
VE_System.vBoard1.RFTransReceiver	receive	9.223372036854776E16ms	7000ms	NOT OK
VE_System.vBoard1.RFTransReceiver	transmit	9.223372036854776E16ms	5000ms	NOT OK
VE_System.vBoard2.IED	blinkLED	9.223372036854776E16ms	3600ms	NOT OK
VE_System.vBoard1.gazSensor	senseGaz	9.223372036854776E16ms	600ms	NOT OK
VE_System.vBoard2.gazSensor	senseGaz	9.223372036854776E16ms	600ms	NOT OK
VE_System.vBoard1.IED	blinkLED	9.223372036854776E16ms	3600ms	NOT OK
VE_System.vBoard0.gazSensor	senseGaz	9.223372036854776E16ms	600ms	NOT OK
VE_System.vBoard2.RFTransReceiver	receive	9.223372036854776E16ms	7000ms	NOT OK
VE_System.vBoard2.RFTransReceiver	transmit	9.223372036854776E16ms	5000ms	NOT OK
VE_System.vBoard2.tempHumSensor	senseHumTemp	9.223372036854776E16ms	6000ms	NOT OK
VE_System.vBoard0.tempHumSensor	senseHumTemp	9.223372036854776E16ms	6000ms	NOT OK
VE_System.vBoard1.tempHumSensor	senseHumTemp	9.223372036854776E16ms	6000ms	NOT OK
VE_System.vBoard0.RFTransReceiver	receive	9.223372036854776E16ms	7000ms	NOT OK
VE_System.vBoard0.RFTransReceiver	transmit	9.223372036854776E16ms	5000ms	NOT OK
VE_System.vBoard0.IED	blinkLED	9.223372036854776E16ms	3600ms	NOT OK
VE_System.vBoard1.tempHumSensor	senseHumTemp	9.223372036854776E16ms	6000ms	NOT OK

Figure 6: Schedulability results (NOT OK)

Schedulability Analysis Report				
The system is schedulable				
HW Instance	Utilization	Result		
PhysicalEntity_ch_HwProcessor4	0.666667%	OK		
PhysicalEntity_ch_HwProcessor3	0.666667%	OK		
PhysicalEntity_ch_HwProcessor1	0.666667%	OK		
PhysicalEntity_ch_HwProcessor2_core0	0.666667%	OK		
PhysicalEntity_ch_HwProcessor2_core1	50.00%	OK		
SW Instance	Operation	Response Time	Deadline	Result
VE_System.vBoard0.tempHumSensor	senseHumTemp	35.0ms	100ms	OK
VE_System.vBoard2.IED	blinkLED	35.0ms	100ms	OK
VE_System.battery	power	10000.0ms	20000ms	OK
VE_System.vBoard2.RFTransReceiver	receive	13.0ms	100ms	OK
VE_System.vBoard2.RFTransReceiver	transmit	13.0ms	100ms	OK
VE_System.vBoard.RFTransReceiver	receive	13.0ms	100ms	OK
VE_System.vBoard.RFTransReceiver	transmit	13.0ms	100ms	OK
VE_System.vBoard.tempHumSensor	senseHumTemp	35.0ms	100ms	OK
VE_System.vBoard1.IED	blinkLED	35.0ms	100ms	OK
VE_System.vBoard1.tempHumSensor	senseHumTemp	35.0ms	100ms	OK
VE_System.vBoard0.RFTransReceiver	receive	13.0ms	100ms	OK
VE_System.vBoard0.RFTransReceiver	transmit	13.0ms	100ms	OK
VE_System.vBoard.gazSensor	senseGaz	35.0ms	100ms	OK
VE_System.vBoard1.gazSensor	senseGaz	35.0ms	100ms	OK
VE_System.vBoard0.gazSensor	senseGaz	35.0ms	100ms	OK
VE_System.vBoard0.IED	blinkLED	35.0ms	100ms	OK
VE_System.vBoard2.gazSensor	senseGaz	35.0ms	100ms	OK
VE_System.vBoard.IED	blinkLED	35.0ms	100ms	OK
VE_System.vBoard2.tempHumSensor	senseHumTemp	35.0ms	100ms	OK
VE_System.vBoard1.RFTransReceiver	receive	13.0ms	100ms	OK
VE_System.vBoard1.RFTransReceiver	transmit	13.0ms	100ms	OK

Figure 7: Schedulability results (OK)

which includes allocating components to processor cores. In our case, each node is allocated to one processor running on a single core, except for the second processor, which had two cores and can accommodate two nodes. For analysis purposes, we have allocated the battery unit to one processor running on one core also.

In the *Analysis View*, we can now specify the analysis context, which will contain the software-

hardware instance specifications to run the schedulability analysis. The schedulability results indicated in Fig. 6 show that the system cannot be schedulable due to the excess memory utilization according to the specified real-time properties. We can also observe that the timing deadline constraint specified is a way less than the response time. In this case, when the deadline is increased, the execution time also increases, amplifying the response time exponentially as more components are still waiting to respond to a given request. To make the system schedulable, we have reduced the worst-case execution time by 70% and fixed the deadline for all operations to 100 milliseconds. We have also separated the nodes from running on the same processor and deploy the battery unit to the second processor's second core. As presented in Fig. 7, the system has now become schedulable. Given the schedulability information presented above, we can now know how we may go ahead with the real-case deployment e.g., by having a clear picture of processor types or core counts. Another helpful information we can account too is the event deadlines specifications. In our example, the *SendPayload* action is periodic, and it is deemed to be sent every 200ms (refer to figure 4), which is perfectly fine because each action in the system needs an average of 100ms to be schedulable.

6. Conclusion and future work

Developing Industrial IoT systems has to cope with several challenges, ranging from the heterogeneity in different players to the types of transferred messages. This paper has proposed the CHESSIoT extension covering the Industrial IoT domain on top of the already existing CHESS tool. CHESS is known for its success in modeling and analysis of industrial system engineering applications. We have presented an envisioned mapping from the CHESSIoT model to the ThingML model to support the code generation later. Finally, we showed a modeling and analysis example of a real-time safety use case to validate the current capabilities of the extension. To enhance the scalability of our approach, in our plans, we would like to explore to possibilities of combining the graphical modeling with a textual-based interface. We would also want to extend different real-time and dependability analyses provided by CHESS to cover the CHESSIoT approach taking care of IoT-specific aspects in the analysis processes. ThingML platform is compelling, but as we have seen, there are still other essential aspects it can't cover now, for example, concerning model checking and analysis. In our plans, we would like to fully automate the ThingML code generation process from CHESS, in which the platform-specific code generation will be compiled and generated directly from the CHESSIoT environment. In the future, we would also like to explore the possibility of exposing the analysis infrastructure so that any external user can consume the tool remotely by consuming an open API.

Acknowledgments

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° 813884.

References

- [1] L. Freund, S. Al-Majeed, Modeling industrial iot system complexity, in: 2020 3ICT, 2020, pp. 1–5. doi:10.1109/3ICT51146.2020.9311942.
- [2] A. Gómez, M. Iglesias-Urkia, A. Urbieto, J. Cabot, A model-based approach for developing event-driven architectures with asyncapi, in: Proceedings of the 23rd ACM/IEEE MODELS'20, 2020, p. 121–131. doi:10.1145/3365438.3410948.
- [3] J. Venkatesh, B. Aksanli, C. S. Chan, A. S. Akyürek, T. S. Rosing, Scalable-application design for the iot, IEEE Software 34 (2017) 62–70. doi:10.1109/MS.2017.4.
- [4] J. Huang, S. Li, Y. Chen, J. Chen, Performance modeling and analysis for iot services, International Journal of Web and Grid Services 14 (2018) 146. doi:10.1504/IJWGS.2018.090742.
- [5] K. Thramboulidis, F. Christoulakis, UML4IoT—A UML-based approach to exploit IoT in cyber-physical manufacturing systems, Computers in Industry 82 (2016) 259–272. doi:https://doi.org/10.1016/j.compind.2016.05.010.
- [6] 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.
- [7] B. Costa, P. F. Pires, F. C. Delicato, Modeling iot applications with sysml4iot, in: 2016 42th SEAA, 2016, pp. 157–164. doi:10.1109/SEAA.2016.19.
- [8] 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 DASC, 14th PiCom, 2nd DataCom, 2016, pp. 392–399. doi:10.1109/DASC-PICom-DataCom-CyberSciTec.2016.81.
- [9] 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 MODELSWARD, 2021, February 8–10, 2021, SCITEPRESS, 2021, pp. 262–269. doi:10.5220/0010269702620269.
- [10] A. Cicchetti, F. Ciccozzi, S. Mazzini, S. Puri, M. Panunzio, A. Zovi, T. Vardanega, CHESS: a model-driven engineering tool environment for aiding the development of complex industrial systems, in: IEEE/ACM, ASE'12, Essen, Germany, September 3–7, 2012, ACM, 2012, pp. 362–365. doi:10.1145/2351676.2351748.
- [11] L. Baracchi, S. Mazzini, S. Puri, T. Vardanega, Lessons learned in a journey toward correct-by-construction model-based development, in: 21st Ada-Europe conference, Pisa, Italy, June 13–17, 2016, Proceedings, volume 9695 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 113–128. doi:10.1007/978-3-319-39083-3_8.
- [12] N. Harrand, F. Fleurey, B. Morin, K. E. Husa, Thingml: A language and code generation framework for heterogeneous targets, in: In the ACM/IEEE 19th MODELS'16, 2016, p. 125–135. doi:10.1145/2976767.2976812.
- [13] F. Ihirwe, D. Di Ruscio, S. Mazzini, P. Pierini, A. Pierantonio, Low-code engineering for internet of things: A state of research, in: In the 23rd ACM/IEEE MODELS'20: Companion Proceedings, MODELS '20, 2020. doi:10.1145/3417990.3420208.
- [14] S. Mazzini, J. M. Mavaro, L. Baracchi, A model-based approach across the iot lifecycle for scalable and distributed smart applications, in: IEEE 18th ITSC, 2015, Gran Canaria, Spain, September 15–18, 2015, IEEE, 2015, pp. 149–154. doi:10.1109/ITSC.2015.33.
- [15] 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>.

- [16] S. Dhoubi, A. Cuccuru, F. L. Fèvre, S. Li, B. Maggi, I. Paez, A. Rademacher, N. Rapin, J. Tatibouet, P. Tessier, S. Tucci, S. Gerard, Papyrus for IoT – a modeling solution for IoT, 2016.
- [17] 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 ICCCS, IEEE, 2019, pp. 1–8.
- [18] A. Bassi, M. Bauer, M. Fiedler, R. van Kranenburg, S. Lange, S. Meissner, T. Kramp, Enabling things to talk, Springer Nature, 2013.
- [19] M. Hussein, S. Li, A. Radermacher, Model-driven development of adaptive IoT systems, in: 2017 MODELS Satellite Event, volume 2019, Austin, United States, 2017, pp. 17–23.
- [20] S. Gérard, C. Dumoulin, P. Tessier, B. Selic, Papyrus: A UML2 tool for domain-specific language modeling, volume 6100 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 361–368. doi:[10.1007/978-3-642-16277-0_19](https://doi.org/10.1007/978-3-642-16277-0_19).
- [21] W. Godard, G. Nelissen, Model-based design and schedulability analysis for avionic applications on multicore platforms, *Ada User Journal* 37 (2016) 157–163.
- [22] L. Bressan, A. L. de Oliveira, L. Montecchi, B. Gallina, A systematic process for applying the chess methodology in the creation of certifiable evidence, in: EDCC, 2018, pp. 49–56.
- [23] L. Pace, M. Pasquinelli, D. Gerbaz, J. Fuchs, V. Basso, S. Mazzini, L. Baracchi, S. Puri, M. Lassalle, J. Viitaniemi, Model-based approach for the verification enhancement across the lifecycle of a space system, in: INCOSE CIISE2014, 2014.
- [24] S. Mazzini, The concerto project: An open source methodology for designing, deploying, and operating reliable and safe cps systems, *Ada User Journal* 36 (2015) 264–267.
- [25] B. Gallina, E. Sefer, A. Refsdal, Towards safety risk assessment of socio-technical systems via failure logic analysis, in: ISSRE Workshops, 2014, pp. 287–292.
- [26] L. Montecchi, B. Gallina, Safeconcert: A metamodel for a concerted safety modeling of socio-technical systems, in: MBSA, 2017, pp. 129–144.
- [27] 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, 2019, pp. 99–115.
- [28] N. Ferry, et al, Development and operation of trustworthy smart iot systems: The enact framework, in: Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment, 2020, pp. 121–138.
- [29] I. Berrouyne, M. Adda, J.-M. Mottu, J.-C. Royer, M. Tisi, Cypriot: framework for modeling and controlling network-based iot applications, in: In 34th ACM/SIGAPP Symposium on Applied Computing, 2019, pp. 832–841.
- [30] M. Faugere, T. Bourbeau, R. d. Simone, S. Gerard, Marte: Also an uml profile for modeling aadl applications, in: 12th IEEE ICECCS 2007, 2007, pp. 359–364. doi:[10.1109/ICECCS.2007.29](https://doi.org/10.1109/ICECCS.2007.29).
- [31] M. Gonzalez Harbour, J. J. Gutierrez Garcia, J. C. Palencia Gutierrez, J. M. Drake Moyano, Mast: Modeling and analysis suite for real time applications, in: Proceedings 13th Euromicro Conference on Real-Time Systems, 2001, pp. 125–134. doi:[10.1109/EMRTS.2001.934015](https://doi.org/10.1109/EMRTS.2001.934015).