



universität
wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

**“Model-based framework for cyber-physical
systems management in OMiLAB”**

verfasst von / submitted by

Petar Byulbyulev, BA

angestrebter akademischer Grad / in partial fulfillment of the requirements for the
degree of

Master of Science (MSc)

Wien, 2020 / Vienna 2020

Studienkennzahl lt. Studienblatt /
degree programme code as it appears on
the student record sheet:

UA 066 926

Studienrichtung lt. Studienblatt /
degree programme as it appears on
the student record sheet:

Masterstudium Wirtschaftsinformatik UG2002

Betreut von / Supervisor:

o. Univ.-Prof. Dr. Dimitris Karagiannis

Acknowledgements

This thesis is the final challenge from my master study. I would like to thank: my family for their support during all these years, Prof. Dr. Dimitris Karagiannis for supporting this thesis, Dr. Michael Walch for guiding me and being a really great supervisor, the whole OMiLAB team for their motivation and ideas. I am grateful for the privilege of being part of the Faculty of Computer Science in the University of Vienna.

Abstract

The digital transformation age brings new opportunities and challenges that are a cause for changing business requirements. An important factor is the alignment between human insights and information systems. Therefore the classical enterprise models - which were mainly developed for human interpretation and understanding - need to be extended with new conceptual designs and functional capabilities, allowing them to represent more automated processes, which can support industry and people in their daily activities. A new disruption is the change from information systems to cyber-physical systems (CPS). CPS are becoming part of industrial practices and personal daily routines and this trend is rapidly increasing. The problem can be split in three main layers - scenario layer, modelling layer and execution layer. The scenario layer represents results from design thinking activities about use-cases for CPS, the execution layer represents the CPS runtime environment and the modelling layer realizes the connection between the other two components. This thesis offers an effective software solution divided into three parts: "smart" models, which encourage design thinking and business mental models decomposition; CPS interfaces common standard, which offers unified connection mechanism; and a service oriented architecture, which links these models and CPS together. The product is a conceptual framework, realized as a standalone component in the form of a web application. It is composed from many different features and modules, transforming this complex task into a rather simplistic one for the end user. It allows the "smart" models to be uploaded and executed on the platform, the CPS to collaborate in different situations using sensors and the user to make decisions during execution. It offers a mechanism for automatic capabilities extraction from the CPS and matching of these capabilities with the model components. It contains a database, where the models and the CPS properties can be stored. It gives the user the possibility to search and browse models and the frontend is also responsive, reliable and fast. Essentially it behaves the same way on different types of devices and operating systems, regardless of size or resolution. To be able to validate the whole architecture, its features and performance, experiments with prototype CPS were conducted in the OMiLAB environment for different use-cases. The experiments were of increasing complexity, exploring the effectiveness of the conceptual framework and the web application developed in this master thesis.

Zusammenfassung

Das Zeitalter der digitalen Transformation bringt neue Gelegenheiten und Herausforderungen, die die Geschäftsanforderungen verändern. Ein wichtiger Faktor ist die Abstimmung zwischen menschlichen Einsichten und Informationssystemen. Aus diesem Grund müssen die klassischen Unternehmensmodelle, die hauptsächlich für die Interpretation und das Verständnis der Menschen entwickelt wurden, um neue Konzepte und Funktionen erweitert werden, um automatisierte Prozesse darstellen zu können, die die Industrie und die Menschen bei ihren täglichen Aktivitäten unterstützen können. Eine neue Störung ist der Wechsel von Informationssystemen zu Cyber-Physical Systems (CPS). CPS werden Teil industrieller Praktiken und persönlicher Alltagsroutinen, und dieser Trend nimmt rasch zu. Das Problem kann in drei Hauptebenen unterteilt werden - Szenarioebene, Modellierungsebene und Ausführungsebene. Die Szenarioebene stellt Ergebnisse aus Entwurfsüberlegungen zu Anwendungsfällen für CPS dar, die Ausführungsebene stellt die CPS-Laufzeitumgebung dar und die Modellierungsebene stellt die Verbindung zwischen den beiden anderen Komponenten her. Diese Arbeit bietet eine effektive Softwarelösung, die in drei Teile unterteilt ist: "intelligente" Modelle, die das Design Thinking und die Zerlegung von Geschäftsmodellen ermutigen; Gemeinsamer Standard für CPS-Schnittstellen, der einen einheitlichen Verbindungsmechanismus bietet; und eine serviceorientierte Architektur, die diese Modelle und CPS miteinander verbindet. Das Produkt ist ein konzeptionelles Framework, das als eigenständige Komponente in Form einer Webanwendung realisiert ist. Es setzt sich aus vielen verschiedenen Funktionen und Modulen zusammen und verwandelt diese komplexe Aufgabe in eine recht einfache für den Endbenutzer. Es ermöglicht das Hochladen und Ausführen der "intelligenten" Modelle auf der Plattform, die Zusammenarbeit der CPS in verschiedenen Situationen mithilfe von Sensoren und das Treffen von Benutzerentscheidungen während der Ausführung. Es bietet einen Mechanismus zum automatischen Extrahieren von Fähigkeiten aus den CPS und zum Anpassen dieser Fähigkeiten an die Modellkomponenten. Es enthält eine Datenbank, in der die Modelle und die CPS-Eigenschaften gespeichert werden können. Es gibt dem Benutzer die Möglichkeit, Modelle zu suchen und auszuwählen, und das Frontend ist responsiv, zuverlässig und schnell. Im Wesentlichen verhält es sich auf verschiedenen Gerätetypen und Betriebssystemen gleich, unabhängig von Größe oder Auflösung. Um die gesamte Architektur, ihre Funktionen und ihre Leistung validieren zu können, wurden in der OMiLAB-Umgebung Experimente mit CPS Prototypen für verschiedene Anwendungsfälle durchgeführt. Die Experimente wurden immer komplexer und untersuchten die Wirksamkeit des konzeptionellen Frameworks und der Webanwendung, die in dieser Masterarbeit entwickelt wurden.

Content

1	Introduction	1
2	Problem Analysis	3
2.1	Problem description	3
2.2	Architecture problems	4
2.3	Related problems	5
2.4	Solution framing	5
3	Conceptualisation	7
3.1	Design modelling environment	8
3.1.1	Environments	8
3.1.2	Scenarios	8
3.1.3	Meta-model	9
3.2	ICT framework	14
3.3	Run-time environment	16
4	Background	17
4.1	ADOxx	17
4.2	Model execution	18
4.3	HTML	19
4.4	JavaScript	20
4.5	NodeJS	20
4.6	Express	23
4.7	Handlebars	24
4.8	XML to JSON	24
4.9	Lodash	25
4.10	HTTP requests	26
4.11	CSS3 and Bootstrap	26
4.12	SQLite	28
4.13	Related work	28

5	Implementation	30
5.1	Database	30
5.2	ICT Framework structure	33
5.2.1	Express and Handlebars	33
5.2.2	Models search	34
5.2.3	CSS3	35
5.2.4	Bootstrap	35
5.3	Upload and commands-endpoints matching	36
5.3.1	XML-Parser	36
5.3.2	Insert of extracted data	37
5.3.3	Read swagger JSON from CPS	40
5.3.4	Model commands - CPS endpoints matching	42
5.4	Models overview	44
5.4.1	Environment and Scenario models	45
5.4.2	Single Environment model	45
5.4.3	Single Scenario model	47
5.5	Models execution	49
5.5.1	NodeGroup	50
5.5.2	SimpleNode	53
5.5.3	CommandNode	56
5.5.4	CPS properties	59
5.5.5	Config	59
5.5.6	CPS interfaces	59
6	Deployment and validation use-cases	61
6.1	Deployment	61
6.1.1	CPS interfaces	61
6.1.2	"smart" models	62
6.1.3	ICT framework	65
6.2	Use-cases	70
6.2.1	Use-case 1	72
6.2.2	Use-case 2	74
6.2.3	Use-case 3	75
6.2.4	Use-case 4	75
7	Conclusion	79
7.1	Summary	79
7.2	SWOT analysis	81
7.3	Future work	81
7.3.1	Conceptual models and elements	81

7.3.2	ICT framework	84
7.3.3	CPS devices	85
7.4	Final comments	85
Acronyms		87
List of Figures		89
Bibliography		92

1: Introduction

The digital transformation age brings many new opportunities and challenges, transforming the way how people interact with the surrounding physical world. Exactly as the internet changed fundamentally how humans access and share information, the CPS reshape our understanding about the physical environment [1]. The CPS are described as interconnected, collaborating control systems. They are deeply connected with the surrounding physical world and all the processes happening there. They receive and process data over the internet [2]. The CPS integrate sensing, controlling, networking and computational possibilities into the physical objects and allow people to orchestrate the infrastructure with real-time performance [3]. The OMiLAB-Rob project provides long-term "research and experimentation" in this field [4]. The project is aiming at understanding, modelling, and engineering of the "knowledge-intensive systems of the future". Connecting the physical and the virtual space, OMiLAB-Rob goal is to integrate enterprise models, which are constructed for humans, with CPS. Inspired from this idea, it can be concluded that the classical models, which are mainly designed for humans are no longer sufficient. They need to be transformed and aligned with the CPS, in order to build more automated processes, allowing production, which needs less human assistance. If the CPS can map and execute such enterprise models, they will be able to perform different tasks, which can change many aspects of our life. There are plenty of examples for such processes on a daily basis like "autonomous cars, robotic surgery, smart manufacturing, automated medical devices" [5]. To be able to transform these enterprise models into "smart" conceptual models, accordingly the enterprise models need to be connected with CPS run-time environment, combining the design thinking processes and providing interpretable mechanisms for the CPS. One of the main technical challenges is how to build "smart" models and execute them in a way that allows the CPS to collaborate with each other, to exchange different commands and how can this connection stay uninterrupted. Another important factor is, how can a communication between many devices be analysed and what kind of interface is needed, in order to con-

trol and track every process, performed from each CPS. This Master thesis introduces an answer to three research questions: 1. What kind of architecture is needed, in order to build an automated link between models invented through design thinking approach and CPS run-time environment. 2. What components should such architecture contain. 3. How to build each of these components as a standalone product and how to link them together.

The result of this thesis is an end-to-end architecture, defined as a model-based framework, which covers every step in the process. Each component is standalone and can be used separately. The model-based framework includes "smart" models, with specific conceptual artefacts, which on one side realize the design thinking idea and on other side satisfy the CPS run-time requirements. The "smart" models are implemented on a meta-modelling platform "ADOxx", using approved modelling methods and standards. Moreover, this end-to-end architecture presents a ICT framework, packed into a web application, which can import these "smart" models and execute them. The ICT framework provides many interesting frontend and backend features. Last but not least, the architecture proposes a common CPS interface standard, making the CPS capabilities machine readable and allowing the CPS to describe their own structure. To validate this work, four experiments will be conducted into the OMiLAB environment, combining these three components together and testing the performance and the capabilities of the model-based framework.

2: Problem Analysis

This chapter explains the general problem, which this master thesis tries to solve. It discusses why this problem is important and how it could be divided into smaller steps. It covers relevant related problems and considerations how the structure of the whole thesis will look like.

2.1 Problem description

The main problem is the need of an effective end-to-end software solution for integration and connection of Scenario layer, through Modelling layer to Execution layer or in other words how to generate universal automated link between models invented through design thinking approach and CPS run-time environment [6]. On one side there is a meta-modelling platform like "ADOxx", which has endless opportunities. It encompasses the creativity of human imagination. The issue is that the standard enterprise models are constructed for human use and need to be somehow connected with the conceptual models and the CPS run-time environment. These conceptual models are so useful, because their modelling elements can represent exactly the needs of the CPS and also



Figure 2.1: Three Layer Framework

realize the conceptual designs. It should be taken into consideration that the CPS can be very different from each other in direction of design and capabilities [7]. Based on these differences, accordingly the requirements for the corresponding architecture are abstract. The problem can be split into the three layer model - Scenario layer, which represents the design thinking, modelling layer, execution layer, which represents the run-time environment, where the CPS are.

2.2 Architecture problems

In order to define how suitable "smart" models should look like and what exactly they should contain to fulfil these requirements, there are couple of things, that need to be considered. First, the conceptual models should be composed from such meta elements that represent the human understandings and the CPS capabilities. Moreover, it is important to define an appropriate structure how to build a process with these models, so the CPS are able to do some actions and collaborate with each other. It must be considered, what are the most important properties, that the model elements should contain. Furthermore, it has to be decided, whether one model type is sufficient to describe the whole architecture and when it isn't, the next problem is, how these different model types will be connected and whether it makes sense to have many-to-many relationship between them.

After the models were briefly discussed, the next important question is, what kind of architecture is needed, to fulfil the connection gap between the models and the CPS. A possible realization is a service oriented architecture (ICT Framework), which can integrate the models and also fulfils the CPS connection requirements. When it comes to such architecture, there are plenty of possibilities, how it can be build. A Web solution could offer many business advantages like - low costs, no need of installation, centralized system, easy to maintain and extend, 24/7 availability, accessible from everywhere, no hardware/software requirements.

Model integration is not an easy task and it should be defined how these models will be transferred into such service oriented architecture. Note, that a platform like "ADOxx" offers various data export formats, like ADL and XML. Due to the proximity of XML with HTML, it may be considered as the more appropriate format for web based application. Once the models are exported and imported into the ICT framework, because of the large file size and the complexity of the XML tree structure, only the important model properties need to be extracted. In order to do that, the data should

be transformed to even more convenient format. Furthermore, it should be discussed where these models will be stored or what kind of database is most appropriate in this situation. The next important issue is, how exactly the database structure should look like, how to split the elements between the different tables and how they will be connected to each other. The design of the database is not a trivial task, as it requires deeper knowledge about the processes, that the ICT framework will have to perform. One of these processes, is the ability of the platform to interact with CPS. The communication process should be smooth and steady. For this purpose it is important to consider, which protocols need to be used and what possibilities do the target CPS interfaces offer. Additionally, the target CPS devices and their Endpoints may need to be unified to some standard format, which can be automatically read by the ICT framework. After all these steps are taken into consideration and both sides of the platform are discussed - on one side the "smart" models and on the other the CPS interfaces, the next important step is how to map the model properties with the CPS capabilities or in other words how to connect the model elements with the actual run-time environment of the CPS.

2.3 Related problems

As the mobile devices usage continues to grow across the world, it should be considered, how to implement this ICT framework on the frontend. For a web application is important to make the users feel comfortable to interact with the interface. Moreover, this user interface must fit with different operating systems and types of devices, regardless of size or resolution. Furthermore, the interface should allow the user to be naturally able to click on different elements on the screen, both with a mouse and with finger. This means, that all clickable elements must be very precisely chosen. Their size and resizing ability play an important role. Additionally, as the available space on the screen could be really scarce, one of the most important questions is, which elements are really necessary and how to arrange them in a way, that the design doesn't change, regardless of space or resolution.

2.4 Solution framing

Last but not least, an important question is, how to frame the whole solution on a methodological level. In order to deliver such end-to-end product, first the most important aspects should be summarized and explained in details. That includes the design

modelling environment, the ICT framework and the run-time environment, which will be discussed in the next chapter "Conceptualisation". Moreover, in chapter "Background" will be defined, what kind of technologies and techniques are used for the development of the solution. Furthermore, it is essential to describe the whole implementation process and the major problems, occurred during this phase in chapter "Implementation". In addition, the model-based framework will be deployed as an application, including the "smart" models, the ICT framework and the CPS interfaces, which have specific technical and physical requirements. Each part will be explained in details from a user point of view. There will be a validation phase, in which different use-cases will be performed and the real capabilities of the platform will be demonstrated. All these topics will be described in chapter "Deployment and Use-cases". Finally, it will be summarized what this thesis tried to achieve and will be discussed, what are the future possibilities for further development and which modules can be extended on a higher level.

3: Conceptualisation

The aim of this master thesis is to provide an effective end-to-end software solution of the following problem - how to generate universal automated link between models invented through design thinking approach and CPS run-time environment. The resulting model-based framework can be divided into three parts:

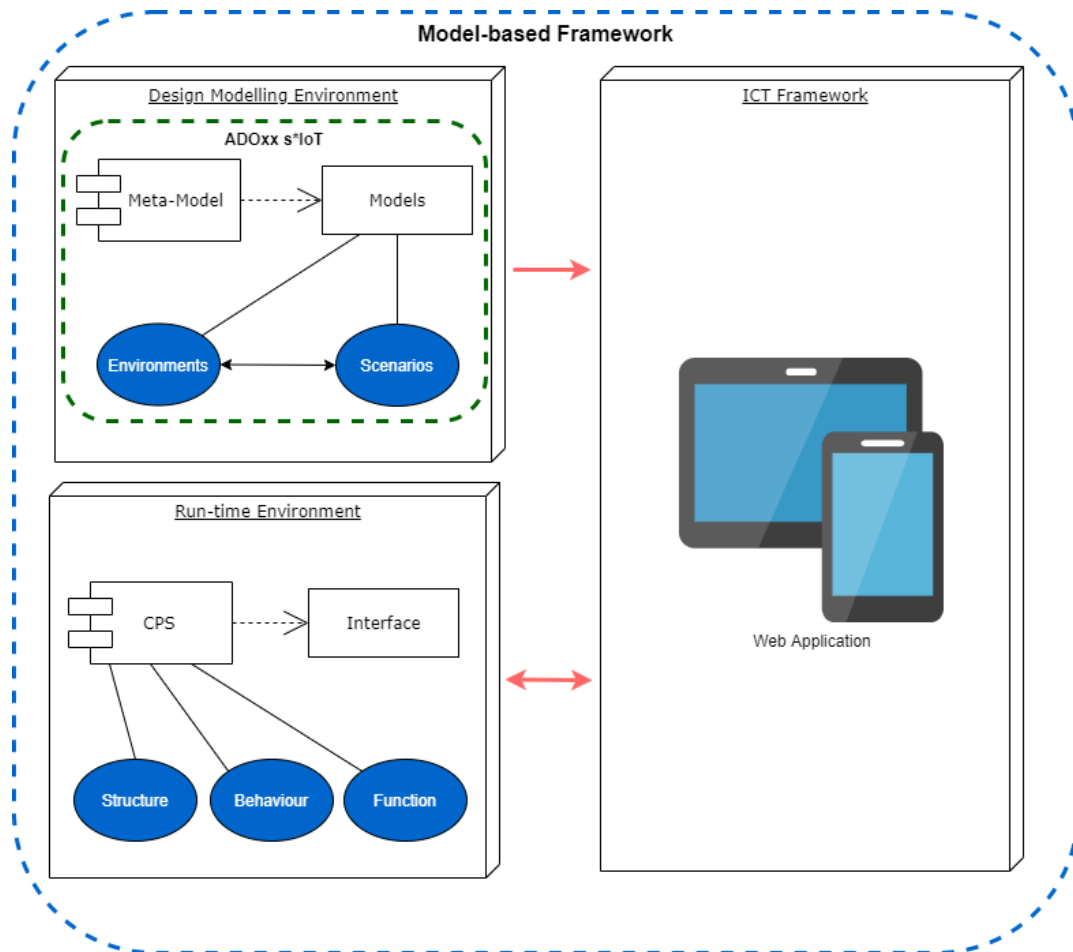


Figure 3.1: Model-based Framework

3.1 Design modelling environment

The design modelling environment consists of meta-model and two types of "smart" models, invented in relation to design thinking and satisfying the CPS run-time environment needs. They are all part of the s*IoT conceptual modelling framework. As shown on figure 3.1 the connection between the design modelling environment and the ICT Framework is unidirectional. The models are uploaded and extracted into the platform.

3.1.1 Environments

In order to perform a technical implementation, the first step is to generate a mental model. The mental model is defined as the human interpretation of some idea. The Environments offer an abstract view of a designer's vision, describing the surrounding area, the conditions. One model of this type presents one particular environment, where the CPS are acting and collaborating together. An Environment model could include many different CPS, with diverse roles. On one side it is important to allow the business to define its conceptual design requirements and on other side the CPS run-time environment should be considered. The basic structure of the Environments represent real physical models, which realize the business value. In order to fulfil the CPS run-time environment requirements, it is crucial to integrate specific technical artefacts, which fulfill the CPS needs. The CPS could be very different from one another. They have different designs, contain different capabilities and could also be accessed in various ways. Every CPS inside an Environment model contains the most critical basic properties, which every possible heterogeneous CPS should have in common:

- CPS Type - describing what kind of device is being used
- Host address - containing the IP/Domain address of the CPS

With these in hand, the run-time environment requirements of the CPS are fulfilled, as it is clear, exactly which CPS should be reached and what its capabilities are. How exactly the CPS type and the capabilities are connected will be explained in the next chapters.

3.1.2 Scenarios

As the Environments were briefly discussed, the next most important step is to define, how exactly the CPS are going to perform some actions and how they will be able

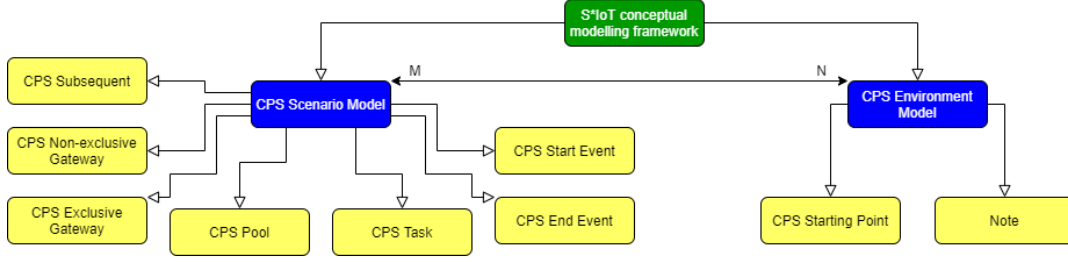


Figure 3.2: Meta-model

to collaborate. On a design level, one of the most intuitive ways to describe some activity, is to build a process. The business already has a stable solution regarding this problem - BPMN. BPMN is a sequence of activities, which try to achieve some common business goal. The Scenarios are based on a flow-charting technique similar to BPMN, but also aligned with the CPS. The process describes some specific actions, executed by one or more CPS sequentially or in parallel. The Scenarios contain many different elements, all fulfilling the business needs and the run-time environment requirements of the CPS. A Scenario model shouldn't limit the designer vision, so it can contain as many CPS as needed. In order to build such a process, which also the CPS are able to interpret, just like by the Environment models, every CPS should be separated accordingly. The most intuitive way just like by standard BPMN models, where every pool/swim-lane describes one actor, is to put every single CPS and his activities into separate group. Moreover, every process needs a start event and end event, defining which one is the first and respectively the last activity. Furthermore, if the whole process is just sequential and the performed actions are straight-forward, there will be no possibility for collaboration between the CPS. Therefore, a parallel execution is introduced to allow them to interact with each other and to perform some common operations. In addition to make the interaction even more attractive from business point of view, decision elements are included, where the designer can implement the models in such way, that during the execution it could be decided in which direction the process continues.

3.1.3 Meta-model

The meta-model is the “formalism of modelling” [8]. It describes predefined classes containing frames, rules, constraints for modelling elements. The meta-model is fundamental for the “Scenario layer”. In this subsection it will be explained in details, what kind of elements were implemented on meta-level, in order to fulfil the business needs

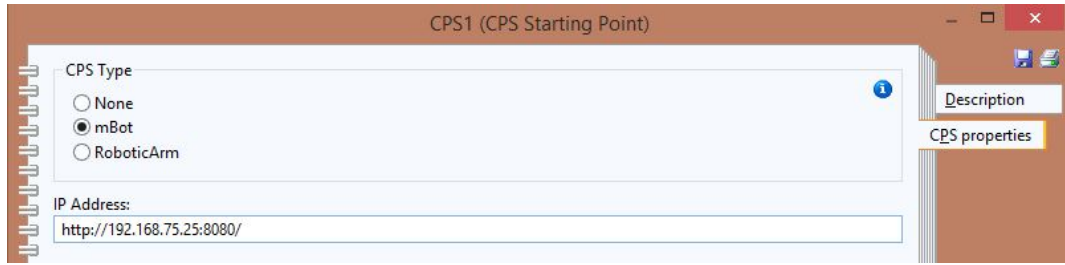


Figure 3.3: CPS Starting Point

and CPS run-time environment requirements. It should be taken into consideration, that all meta-elements and the models are created as part of the s*IoT conceptual modelling framework project as shown on Figure 3.2:

- **CPS Starting Point** - this meta-element is part of the Environment models, describing one particular CPS. The design role of this element in the model is to show on which position exactly the CPS will execute some action, or where it will start with its tasks. It contains two sub-sections - Description and CPS Properties. In the description sub-section, there are important properties like "Name" and "Text Position". They allow the designer, to adjust the element on the screen in the most appropriate way. In the CPS Properties sub-section, there is a list with all possible CPS Types, which are already integrated. One of them can be selected. There is also an input field for the IP Address, which should point to the device. If it's needed, a port for connection must be provided. An example is shown on Figure 3.3. If there is demand to add a new CPS Type, the designer can add such in the ADOxx Development Toolkit. Inside the toolkit, under "Class Hierarchy" - "_D-construct_" - "_Conceptual_" - "_Annotation_" - "CPS Starting Point" the attribute "CPS Type" should be extended.
- **Note** - it is part of the Environment models. This element was derived from the s*IoT conceptual modelling framework. Its most important property is the ability to inherit user-selected photos as background of the element, which can describe the environment, the conditions and in some cases also the visual representation of the CPS. This property is contained into sub-section Presentation, under External graphic, under Program arguments.
- **CPS Pool** - it is part of the Scenario models, the CPS Pools serve as a container for elements. Their main role is to combine all contained elements into a single group both from a design and technical point of view. The most important property,

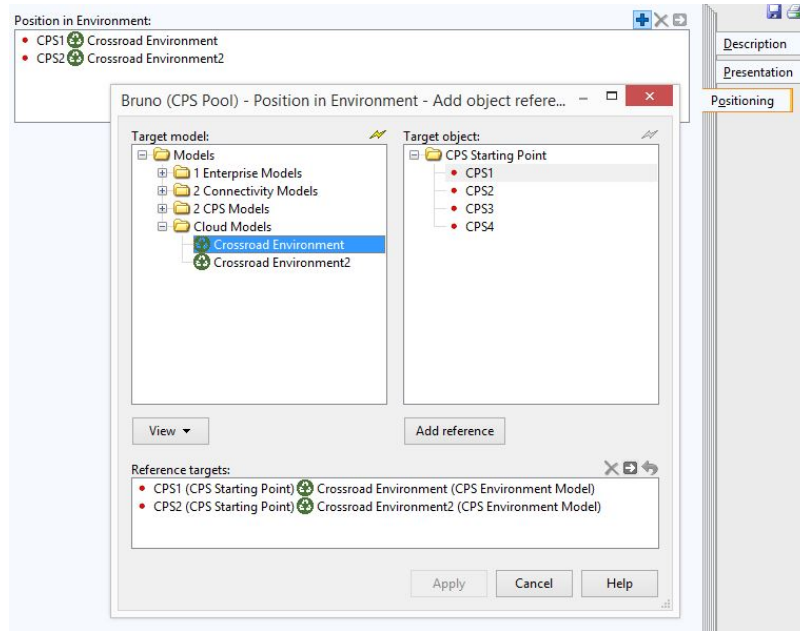


Figure 3.4: CPS Pool Positioning Connections

that the CPS Pools contain, is the Inter-Ref ability, which is set inside sub-section Positioning. It is implemented in a way, that every CPS Pool can be connected to a CPS Starting Point (from an Environment model). In order to maintain a many-to-many connection between Scenario and Environment models every CPS Pool can be connected to exactly one CPS Starting Point from a single Environment model, but to unlimited many Environment models. This way a Scenario model can be executed with each connected Environment model, if there is one or more CPS Pools connected to CPS Starting Point from it. An example is visible on Figure 3.4. It is also desirable, that every Scenario model contains one CPS Pool that represents the ICT framework itself. This CPS Pool is not connected to any CPS Starting Point. Its role is more representative, to show, how the workflow goes from one CPS to another CPS and how they collaborate together. Examples with such Scenario models will be presented in chapter 6.

- CPS Start Event - it is part of the Scenario models. This element shows where the model starts and informs the ICT framework, that this is the first element in the workflow. There must be only one element of this type inside a Scenario model. It has two sub-sections - Description and Execution. In the Description part, the element position can be adjusted on the screen. There is also a possibility to hide or show the element name text and to place it on different locations. The Execution

sub-section is designed, to contain commands in the form of semantic triples - subject, predicate, object. The triples represent a binary relationship between objects in a machine readable way. With the help of the triples, the designer is able to describe every command or resource that is needed, in a way that is understandable from humans and machines. He can define different parameters and their respective values. This approach guarantees a wide degree of freedom for interpretation and transformation of the commands and their parameters.

- CPS Task - this element is part of the Scenario models. The CPS Task is the main element in the process workflow. It is designed on one side to describe the process and the current step, but on other side to apply different actions, which fulfill the CPS run-time environment requirements. The CPS Tasks contain three sub-sections - Description, Execution and Task type. The Description part is similar to the other above described elements. It determines, how the element is adjusted visually on the screen. The Execution contains triples, again in the form of subject, predicate, object. The Task type is very important property, as every type has different role. If no task type is assigned, the default one is "Not specified". This type basically means, that the task has only design role, to represent some step in the workflow, but doesn't contain any executable action inside it (usually no triples inside Execution sub-section). The second option is "Function". The "Function" is used for CPS Tasks, which contain commands in the form of triples. They are part of the actual execution process. The CPS Task from this type receives automatically indication icon on the top-left side, defining its role. The last option is "Collaboration". This type is used only inside Non-exclusive Gateway/Parallel elements. Task type "Collaboration" can be used for now only with CPS type "mBot", which means that the CPS Pool, containing this CPS Task has to be connected with CPS Starting Point, representing CPS of type "mBot". Task Type "Collaboration" indicates that before requesting particular command/endpoint, the "mBot" will use his "distance" sensor to calculate the available distance in front of him. If the received distance value is less than certain threshold (meaning that there is an object in front of the "mBot" on a close distance), the "mBot" will wait for the other parallel CPS Task being executed to finish and will start with the current CPS Task again. This way a possible collision can be avoided. More details about this process will be explained in Chapter 5. CPS Task from this type also receives indicating icon on the top-left corner.

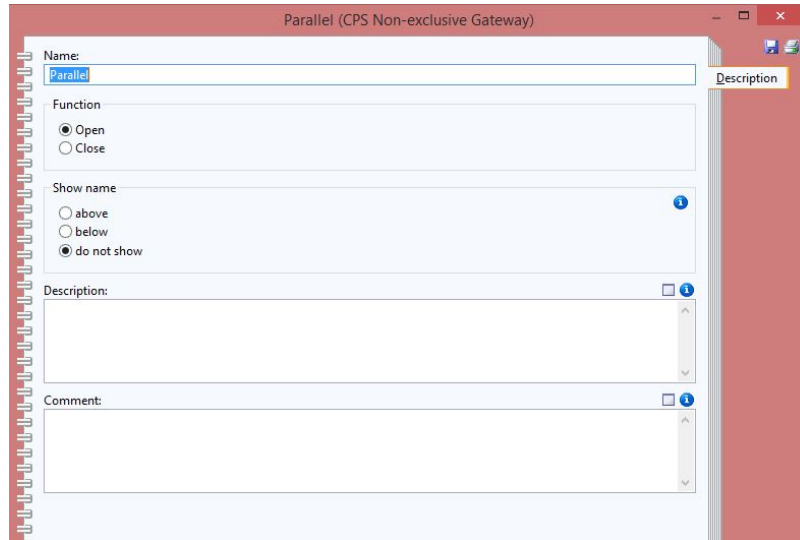


Figure 3.5: Non-exclusive Gateway (Parallel)

- CPS End Event - it is part of the Scenario models. The CPS End Event defines, where the model ends. It is the last element in each Scenario model which informs the ICT framework, that the execution of the current model is finished. In each Scenario model must exist only one CPS End Event. It has two sub-sections - Description and Execution. The Description defines how the element and the element name visually are presented on the screen. In the Execution sub-section are contained the desired commands in the form of triples.
- CPS Non-exclusive gateway (parallel) - it is part of the Scenario models. The parallels contain only one sub-section - Description. Just like by the other elements, it describes how the element is adjusted on the screen. In addition, there is also one important inside-property called "Function". It has two possible states "Open" and "Close". Every time, when there is a need of parallel element in the Scenario model, open and close element should be placed, describing where the parallel process starts and ends. The starting parallel element should have value "Open" for the inside-property "Function" and the closing parallel element should have value "Close" for the inside-property "Function". An example is shown on Figure 3.5. A Scenario model can be infinitely complicated, including as many CPS Non-exclusive gateways as needed. The parallel elements have specific role for the model-based framework. They define, that the elements inside them will be step-wise executed. This property will be explained in more details in Chapter 5.
- CPS Exclusive gateway (decision) - it is part of the Scenario models. The decisions

have one sub-section - Description. It defines the visualisation of the element. It contains the same inside-property like the CPS Non-exclusive gateways - "Function". Same like the parallel elements it has two states "Open" and "Close", defining where the decision starts and ends. The decision elements give freedom to the user, during execution of the model, to decide in which direction the workflow process continues, after the opening decision element. Every Scenario model can contain infinitely many CPS Exclusive gateways.

- CPS Subsequent – This element realizes the process, connecting all the described Scenario model elements into a workflow. It indicates the direction of the workflow, so the ICT Framework knows, how exactly the sequence is defined.

3.2 ICT framework

The focus of this paper is a service oriented architecture, covering the web 2.0 application standards. A framework is described as a complex structure of functions and features with existing control flow in a particular domain. It consists of rules, frames and defines the entire application design. A framework expects specific type of content and on basis of this content generates some output. A framework aims to automate some process or activity. The main idea of the ICT framework is to provide a connection between the "smart" models and the CPS interfaces, giving a possibility to the users to control and customize the entire process, as shown on Figure 3.6.

The platform consists of many different modules, with different roles:

- Models upload - the both types of models Environments and Scenarios are imported into the ICT framework.
- Read CPS Capabilities - allows the user to acquire all CPS capabilities and their properties and saves them into the database (API Endpoints and parameters). The user can see all capabilities for each CPS into specific interface, when they are already available on the platform.
- Protocols, Communication - the role of this module is to maintain the two way communication channel with the CPS (requests and responses) and to provide the output to other necessary modules.
- Frontend - responsive, reliable and fast, the frontend is very important part of the ICT framework, giving the user the possibility to control every process, to trace

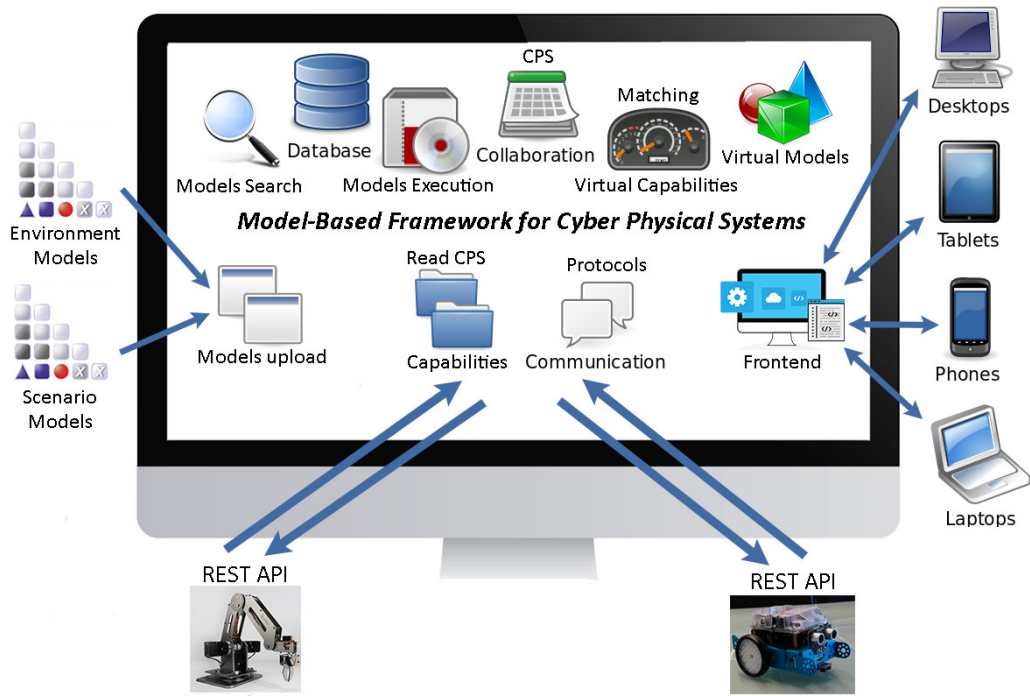


Figure 3.6: Architecture Overview

the execution of the models, to make decisions during execution, to review the models and their properties, to select which pair of models should be executed together and so on.

- Models Search - for more user convenience, there is a models search bar in the navigation, which gives a short-cut, to search for every possible model, regardless of the current location of the user on the platform.
- Database - important structure in this thesis, allowing the users to save their models, CPS types, commands, matched endpoints and everything important related to the platform for further usage.
- Models Execution - this module serves to execute a pair of models, always one Environment and one Scenario, presenting the whole process on the frontend and if necessary asking the user for input.
- CPS Collaboration - the collaboration between CPS is not a trivial task. In order to maintain it, usually there are some sensors and widgets involved. The collaboration means that one CPS can execute a specific action and react properly, depending on another CPS actions.

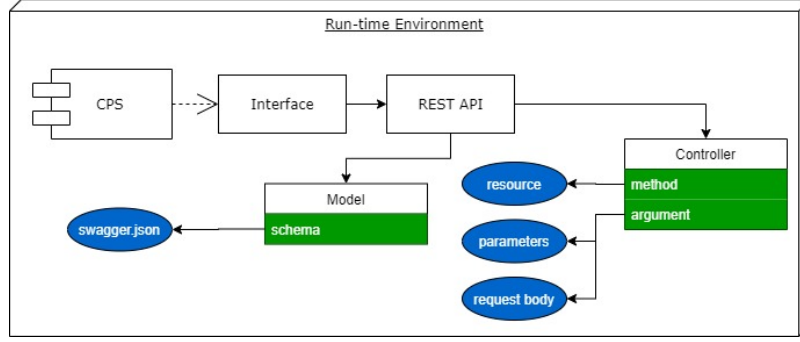


Figure 3.7: Run-time Environment

- Matching Virtual Capabilities - the Scenario model element commands are matched with the actual extracted CPS capabilities, during the Scenario model upload process. The user can select the corresponding CPS Endpoint and the necessary parameters to particular triples.
- Virtual Models - after the models are imported into the platform and inserted into the database with their properties, the ICT framework generates a virtual copy of them for more convenience. The user is able to see all of the contained model elements and their features.

3.3 Run-time environment

The run-time complexity, structure, behaviour and function of a CPS play an important role for the whole communication and implementation process. In order to achieve a unified connection algorithm, it is necessary to align the CPS on a common standard visualized on Figure 3.7. The most intuitive approach, is to build REST API on the top of the CPS interfaces, allowing external participants to make requests and to get responses, granting them access to the needed resources. The executed requests will be often described as commands. The ability of the CPS to provide their own API structure is very important feature for the platform. This way the ICT framework can acquire and extract the API schema from a particular CPS with a single request. Swagger OpenAPI offers machine-readable JSON or YAML file, which is automatically generated from the API annotations. JSON was selected as the preferred format for a web-based service oriented architecture. Another benefit of this approach is that the swagger.json file could be manually created. It should contain all existing endpoints, parameters and basePath according to the original Swagger schema.

4: Background

This chapter covers many topics, used for the implementation of the model-based framework. These technologies are important, to give an idea to the reader about the fundamentals, used in the remaining part of the thesis. Nevertheless, this chapter gives an interesting overview of the current state of the technologies. The model-based framework includes many areas, like implementation of meta-models and models with the "ADOxx" tool, importing and processing of these models in "xml" format into a web application, inserting them into SQLite database, building an user interface to control and track the whole operation, converting and executing these models in such way that the CPS are able to understand them. The management of these devices is a complex task, that involves REST, matching of abstract user based commands to actual device capabilities, sensor data perception, react time and many others.

4.1 ADOxx

ADOxx is a platform for development of modelling methods. The platform uses specific modelling language, different procedures, mechanisms and algorithms. With these tools the task to implement meta-models, models and modelling tool-kits is much simplified [9]. The models can be described as abstraction or representation of some event, action, operation. The models focus on different important aspects and they have a particular purpose. The models are used for simplification of an object. They often exclude some properties and irrelevant details. The models can also be representation of a vision, of an idea. On other side the meta-models represent a further abstraction, building tools necessary for the creation of the models. They determine the modelling language. The meta-models consist of frames, rules and constraints. A meta-model can be described as "formalism of modelling" [10].

4.2 Model execution

Model execution can be defined as a software method, which involves specific types of models that are constructed in a way with certain properties and capabilities, allowing them to be executed on particular system. The paradigm involves many different areas, which will be tackled in this section.

One of the first and more concrete model execution researches in this field was the "Executable UML", which was introduced in 2002 in the book "Executable UML: A Foundation for Model-Driven Architecture" [11]. It is described as an abstract software development method, which idea is to reduce the complexity of a software system. The method allows developers to implement applications at a higher level of abstraction. The main goal is to improve the development cycle, through less effort, better quality and maintainability, re-usability of modules and at a lower cost. Another benefit is to create a cross-platform oriented executable UML domains, which are more generic and are not bound to a certain programming language or technological platform. The executable UML method allows automatic compilation and code generation in an optimized, less abstract form. The executable UML also supports model-driven architecture (MDA), which is the next interesting topic, closely related to model execution paradigm.

MDA was introduced in 2001 by the Object Management Group (OMG) and is defined as a specific approach in the software design area for implementation of software platforms, which are independent of a particular infrastructure, used in the development cycle. MDA provides different guidelines and structures, which are represented in the form of models [12]. The main idea of this approach is to separate the business logic, which should concentrate on the business needs, from the implementation environment. Such independent application models can be built using different approved modelling standards, like UML, BPM, BPMN and many others. The main goal is to separate the application core from particular technology, which allows it to be more flexible and suitable for cross-platform behaviour. The MDA is supported by the model-driven engineering (MDE), which will be the next important pillar, closely related to model execution.

MDE is a development method, which focuses on the creation of conceptual models, which specify and describe particular application domain [13]. The basic idea is to concentrate and extract the knowledge about specific application domain and to maximize the compatibility between the different participants and counter-parties in the development process. MDE deals with all the different activities, best practices and

standardization of the terminology, rather than the computational part of the process. The main goal of this method is to increase the efficiency and to simplify the design process.

Last, but not least the model-driven development (MDD) or also known as model-driven software development (MDSD) will be considered [14]. The MDD concerns itself mainly with the idea how to generate software more efficiently, with less complexity and at lower cost. The basic idea in this method is to create a software model, which describes how a particular software platform should operate, what features it should offer and how it should response in different situations. After the system is implemented, specific test cases are generated, based on the same software model. The test cycle is also called model-based testing (MBT). By this method again the close cooperation and collaboration between IT and business sides is highly desirable, as it provides more accurate idea, how exactly the software should behave and what goals it should achieve. The MDD allows all counter-parties to have similar understanding about the software product, as the generated and executed software model provides a higher level of abstraction. Often such software model is implemented in a domain-specific language (DSL), which ensures a good collaboration process between the different stakeholders.

4.3 HTML

The idea of this project and the resulting ICT framework is a web application, covering the latest standards in web development. Usability and usefulness are the main characteristics, which are considered as essential for the modern websites. HTML5 introduced a new built-in form validation, which earlier needed to be implemented separately by the developers [15]. It is not as customizable as the analogous JavaScript one, but it provides great features like - input field content mandatory check, input field length requirements check or specifications for the type of the field (for example number). HTML5 provides a great set of APIs which allow the development of smoother user experience and more dynamic web applications. The number of mobile devices being produced and used continues to grow fast, so people are using their mobile browsers more than ever [16]. HTML5 affirms as one of the best options in the process of developing a cross-platform mobile web application [17]. With the help of HTML5, CSS3 and JavaScript an application can work properly on different operating systems using the same markup language. The benefit of this process is, that it requires less technical knowledge, time and investments, than developing a separate native application for each single platform (Android,

iOS, Blackberry etc.).

The World Wide Web Consortium (W3C) is constantly working on improving the HTML standards and the reason for this is very simple - the web should work equal on every browser and every platform, regardless of the technology being used. This process also improves the search engine optimisations. The usage of external plugins for simple multimedia tasks should be reduced as much as possible.

4.4 JavaScript

To increase the control over a web application behaviour and make it more automated and interactive, the usage of frontend scripting languages is necessary. At this point, JavaScript is the programming language, which fits best for the web. It is a cross-platform, object-oriented and lightweight scripting language. JavaScript is supported and integrated into all known web browsers, which includes Google Chrome, Internet Explorer, Safari, Firefox, Opera and so on. It allows the developer to access and change the HTML elements by manipulating the Document Object Model (DOM). JavaScript is event-driven, which gives to the developers the possibility to design how a web applications reacts on different events like onclick, onmouseover, onload etc. The language supports both procedural and object-oriented programming styles. One of the biggest advantages of JavaScript even by coding a desktop application, is the possibility for abstraction from the target platform (mobile or desktop). The language has many APIs, which can be described as interfaces allowing the developer to manipulate different elements on the web page. For example in this master thesis AJAX is used to retrieve data from the backend and update different sections of the ICT framework without having to reload the whole page. This step has a big impact on the performance and the interactivity of the web application [18]. The JavaScript ecosystem is expanding every day and this is also the main reason behind the constant success of the programming language. Many big companies adopt and evolve JavaScript on regular basis - Google, Microsoft, Facebook and so on.

4.5 NodeJS

In this master thesis JavaScript is used on both the frontend and the backend side of the ICT framework. Just a few years ago, we couldn't imagine the things that can be done on the web with JavaScript running on the server side and in the browser [19]. This has

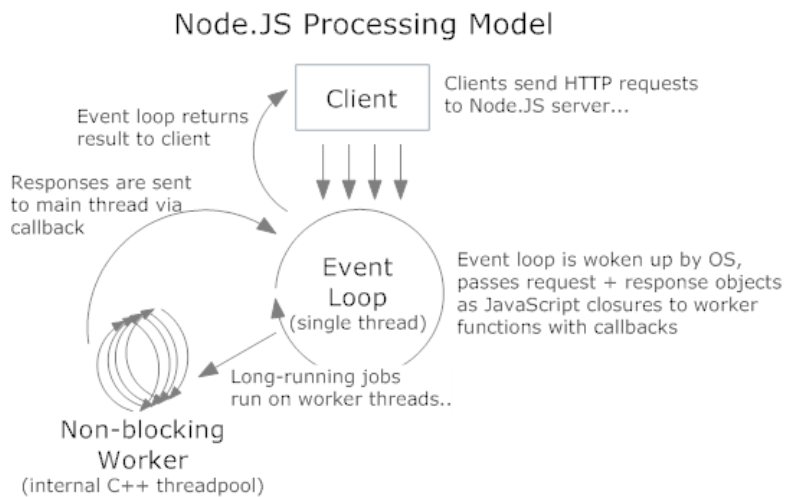


Figure 4.1: NodeJS processing model [21]

the following advantages:

- Same language is used on both sides, with the opportunity to share code between them, for example: form validation, views rendering.
- The number of NPM packages grows every day, including many backend, frontend libraries and tools, where most of them are hosted on GitHub. They use one identical issue reporting system and often such issue is resolved within few hours, after it occurs.
- Well fitted for the "event loop"-based concurrency model, which is also applied in the framework for handling server requests.
- Serialises and deserialises native JSON on backend and frontend (`JSON.stringify()` and `JSON.parse()`) - this makes the process of communication and data exchange between them smooth.
- Event-driven, data-intensive architecture, optimised for asynchronous (non-blocking) Input/Output.
- Multi-user environment support.
- JavaScript is the top language on GitHub, with the most active repositories. [20]

A Stack Overflow survey conducted in 2020 shows that 67.7% of the developers use JavaScript and that JavaScript is most used programming language for a eighth year in

a row [22]. The survey summarizes that NodeJS is the most commonly used technology. The server-side of the ICT framework is implemented in NodeJS [23].

The most important issue, when developing a web application is, that there are three perspectives, which should be taken into consideration. The first one is the server-side, the second one is client-side and then the developer should also consider the possibility for a mobile/native application. JavaScript has already proved as the best web client-side language for the web. NodeJS has also enforced in the last years as the fastest growing server-side language. Netflix and PayPal moved and transformed their products few years ago with NodeJS. According to the companies, this way they achieved "tremendous result-economically and in performance" [24]. LinkedIn's mobile software stack is also built in NodeJS. Thanks to the performance benefits, they decreased the number of servers and instances from 15 servers with 15 instances each to just 4 instances and now they can handle two times more traffic as before [25]. Walmart adopted NodeJS for their mobile app, as they had more than "20,000 hits per second". They needed faster and more reliable solution [26]. eBay deploys on regular basis different applications written in NodeJS in the production process [27].

For the responsive web design, CSS3 and Bootstrap have proven as very good solutions. They will be discussed in the next sub-chapter.

NodeJS is an open-source, lightweight, cross-platform run-time environment. The core of NodeJS is written in C and C++. It utilises the Google's V8 JavaScript engine, which is the same JavaScript engine that runs in Chrome. V8 is an open source "Just in Time" C++ compiler and it compiles JavaScript into assembly level code. NodeJS uses single threaded, event-driven, non-blocking I/O model. It is able to handle many concurrent connections simultaneously. Every connection is executed in parallel and for each of them a callback is fired. Through the callback it can decide if the particular connection ended with success or failure. Every single request is processed in a loop, called the event loop and every request can also trigger another requests inside the system. NodeJS works differently in comparison with other traditional servers like Apache. When the user starts NodeJS, it is automatically registered in the operating system. The same way the operating system processes a callback, when connection is established. The NodeJS event loop model is used for scalability, allowing the server to respond in a non-blocking way, in difference to Apache, where the number of threads or processes is limited. V8 engine compiles JavaScript to a sequence of native machine code instructions, one function at a time. V8 parses the code and extracts an AST (abstract

syntax tree). It performs scope analysis in order to understand to which context a symbol refers to, and translates it to native machine code instructions. V8 also has a garbage collector. The memory consists of two parts: old space and new space. Both of them belong to the heap memory. It keeps track all the time of the JavaScript objects. When a new object is created it is stored into the new space. When the new space memory is full, the V8's garbage collector removes old JavaScript dead objects and place them into the old space. This garbage collector plays an important role for the high performance of NodeJS. In other words NodeJS is very well suited in cases where persistent connection between server and client is needed. That includes data streaming, chat applications or summed up requests where very little CPU computational power is needed from single user for an operation.

The technology called "Web Sockets" in NodeJS allows real-time connection between server and clients. This type of technique is independent of HTTP requests and responses. This way data can be sent and received simultaneously from both sides. It can be described as event based subscription. The WebSocket is a sub-protocol (just like other sub-protocols XML, JSON etc.), which lays on the top of the TCP/IP. The predecessor of this technique is the "long-polling", which was still based on the old HTTP requests and responses, with the simple difference, that a request connection remains open from either side, until the whole response is received, which again results in two-way communication data stream [28].

4.6 Express

Express is the most popular flexible NodeJS web application framework, which provides many important features for handling of HTTP requests. [29] It is very well suited for both web and mobile applications. Express can be described as server-side rendering tool. It uses middle-wares, routing and allows dynamic rendering of HTML pages. The important variables are sent directly from the backend side as parameters and visualised in the HTML templates. Routing refers, as how the app responds to particular endpoints or URIs from user requests. These endpoints can have different HTTP methods like GET, POST, PUT, DELETE and so on. The routing can include every specific path and also the base path. A single path can contain more than one HTTP request methods, like for example the path `www.domain.com/userregister`. There could be GET method, where the user register form is displayed and also POST method, after the form is submitted. When launching an Express web application, there are numerous settings,

```

{{#each models}}
<div class="row element-row">
  <div class="col-2">{{id}}</div>
  <div class="col-4">{{name}}</div>
  <div class="col-4">{{#each startingPoints}}({{envsp_name}}) {{/each}}</div>
  <div class="col-2"><a class="btn btn-primary btn-sm" role="button" href="/environmentmodel/{{id}}">Details</a>
</div>
{{/each}}

```

Figure 4.2: Handlebars Block Expression

which can be set - view template engine, port and so on.

4.7 Handlebars

The view engine used with Express in this master thesis is Handlebars. It allows the developer to build semantic templates with great variety of features. [30] The Handlebars can be described as HTML files with extra functionality inside them. This functionality is in the form of embedded expressions. A Handlebars expression is closed inside of `{{content}}`. Handlebars is very well suited for Express, like described above, the Express routing mechanism generates some variable with content and sends it directly to the Handlebars frontend view template. On the frontend side, it is easy to render the value. Handlebars also supports block expressions like - if, else, each and so on. The block expressions always start with `#`.

The developer is able to write his own Handlebars helpers, which can be used inside the view templates.

Figure 4.3 is an example for two custom helpers used in this master thesis - JSON stringify and JSON parse.

4.8 XML to JSON

"XML-JS" is a JavaScript API for converting XML to JSON or to JavaScript Objects. It is supported by all major browsers. One of the most important benefits of this library compared to similar libraries is the different approach that it uses. When it converts XML tree and for example, there is a sequence of XML elements from the same type, followed by other XML elements from different type, the API converts every single element from the sequence into array of JSON objects. The usual approach of similar libraries is to group all XML elements from the same type into one array, next type into second array and so on. This library was chosen as the most appropriate one for the XML Parser, as exactly the above mentioned situation occurs during conversion of

```

var register = function(Handlebars) {
  var helpers = {
    // all helpers are set here
    jsonStringify: function(content) {
      return JSON.stringify(content);
    },
    jsonParse: function(content) {
      return JSON.parse(content);
    }
  };

  if (Handlebars && typeof Handlebars.registerHelper === 'function') {
    // register helpers
    for (var prop in helpers) {
      Handlebars.registerHelper(prop, helpers[prop]);
    }
  } else {
    // return helpers object if helpers can't be registered
    return helpers;
  }
};

module.exports.register = register;
module.exports.helpers = register(null);

```

Figure 4.3: Handlebars Helper

the ADOxx models. All elements inside a single Scenario model, like CPS Pools, CPS Tasks, the connections between the elements, the inside properties are on the same level of the XML tree. When they are converted into JSON/JavaScript Objects they need to remain on the same level, in order to be processed correctly. The API allows the developer to make the reverse conversion, from JSON/JavaScript Objects back to XML and the end result remains unchanged.

4.9 Lodash

The "Lodash" is very useful JavaScript library for iterating over arrays, objects and strings. It is supported by all major browsers. It has various features like searching for strings in array of objects. This particular feature is often used in this master thesis. "Lodash" gives the possibility to search in arrays from top to bottom and also reverse from bottom to top. This mechanism is very important in situation like when a Scenario model should be executed. Every Scenario model is composed from many parallel and decision elements, which means, that every edge should be processed before the closing exclusive/non-exclusive element is reached. The exact implementation will be discussed in more details in the next chapter of this master thesis. Another important benefit of "Lodash" is that it is supported by Internet Explorer 11. Most of the features provided by the library are already available with JavaScript ECMAScript 6 (ES6), but

unfortunately Internet Explorer 11 doesn't support it.

4.10 HTTP requests

"Request" is very simple library for making HTTP requests. It supports HTTPS and files stream. One of the main benefits over the standard NodeJS "HTTP" library is, that "Request" can be easily debugged, also in cases where there are missing header content-types for example in PUT request. The library checks the file extension and adds the necessary headers. The structure is also much more simplified in compare to the integrated "HTTP" library.

4.11 CSS3 and Bootstrap

CSS is the abbreviation for Cascading Style Sheets. It is used to give set of properties to the HTML elements and to define how they are arranged on the screen. It describes the style of the web page, the used colors, fonts, shapes and so on. It gives the user the visual experience and satisfaction from browsing. CSS can be stored inline the HTML elements, internal the HTML file in header block or outside in separate files, which can be directly included. Several years ago, before the smartphones, most of the websites were built with fixed width, usually around 960px, because this size was appropriate both for users with laptops and users with wider monitor screens. If the user resolution was too large, the developers used to put margins on both sides of the screen, giving them similar browsing experience. This practice changed dramatically since the first iPhone was published in 2007 and even more the years after that. With the help of CSS3 and Bootstrap website can be made in a way, that looks similar on all kind of devices and platforms. This process is called responsive web design. Responsive means that every HTML element has corresponding size and position depending on the target device and resolution. Text size and arrangement of elements also change [31].

The W3C is constantly improving CSS, depending on the web requirements and progress of internet. CSS3 introduces a set of new features and properties. The one, which will be often used in this master thesis is called "Rounded Borders". This feature enables the developer to use a simple property, which defines how rounded the border edges of a table are, making the user experience great. Earlier on older CSS versions, it was necessary to set four different rounded pictures on every edge in order to achieve the same visual effect. Another feature, which is very important for the responsiveness

of a web page is called "Media queries". It allows the developer to use different selectors depending on the screen size, resolution and view-port, which makes the web site looking similar on mobile devices, tablets, laptops and desktop-PCs.

Bootstrap is a frontend framework created by Twitter for developing of responsive web design, which combines HTML, CSS and JavaScript. It is open-source and each feature or template can be extended by the developer, depending on the specific needs. Bootstrap is supported by all modern browsers, including Internet Explorer, Edge, Chrome, Firefox, Safari, Opera. There are five predefined break-point classes in bootstrap, which correspondingly support five different ranges from screen width:

- Extra small - less than 576px
- Small - greater or equal than 576px
- Medium - greater or equal than 768px
- Large - greater or equal than 992px
- Extra large - greater or equal than 1200px

These classes can be extended and used practically on almost every element or template, determining how they will look like on particular screen resolution [32]. The most important features, which will be used in this master thesis are the responsive grid system, form elements, navigation bar and modal windows. The grid is table-like system, which is separated into 12 columns. The developer can aggregate as many columns as needed into a single column, the only rule is that they sum up to 12 at the end. This allows the columns to have different sizes. They and the objects inside them, resize proportionately to the screen size, allowing similar visual effect on different devices and operating systems. Another important feature is the navigation. It has two dynamic layouts depending on the user device and screen size - extend and collapse. On small resolutions it transforms into a clickable single drop-down, which lists the menu tabs below. The navigation supports drop-downs and logo, which are also responsive. All menu items and drop-downs are intended for desktop (mouse clicks) and mobile (touch) devices. The navigation can be placed horizontally or vertically if necessary. It is also possible to integrate different form elements into the navigation bar (like for example search bar). The HTML5 form elements can easily be transformed into bootstrap form elements by adding an additional classes to them. Their properties can be extended and adjusted according to the developer needs. The modal windows are iFrames with

different purposes. They pop up and position themselves over all other elements in the current web page. They can be customized accordingly to the developer needs. In this master thesis often modals will be used to ask the user for confirmation and to prevent an accidental deletion of a model.

4.12 SQLite

SQLite is a small, fast, open source, self-contained relational database engine written in C, that supports all SQL features. It doesn't need any configuration, installation or server process. SQLite is one of the most popular database engines in the world, widely used because of its simplicity. It is embedded directly inside an application. The whole database is contained into one single file. The only thing necessary to access the database file from an application is an external module, which fulfills the connection. SQLite supports one particular module called "sqlite3" for two-way connection with the database file. The module provides control over queries, debugging support, error forwarding, serialized and parallel query execution flow, query placeholders (against SQL-injections).

4.13 Related work

This section gives an overview about important literature findings in the area of conceptual models execution and CPS/IoT. There are some interesting projects, which concern themselves with the topic, but unfortunately most of them cover only different parts from the whole solution.

One exciting project was published recently - "Modeling the Internet of Things in Configurable Process Models". It was created by M. Kunal Suri for the Université Paris-Saclay [33]. The project is dealing mainly with the possibility to connect business process models with IoT. It introduces new specific transformed type of business process models called Configurable Process Models (CPM). These specific models on one side contain the most important business process properties and on other side speed up the implementation by avoiding "development of processes from scratch, which is both time-consuming and error-prone". The project concentrates deeply on how to use and manage the IoT resources efficiently. It looks at the heterogeneous properties of the IoT and tries to minimize the costs for execution, like "energy cost, computation cost".

Another project published in year 2017 - "Model-driven development of user inter-

faces for IoT systems via domain-specific components and patterns” deals with the idea of generating IoT frontend user interfaces through domain specific models. This paper concentrates on the idea, how to improve and implement frontend interfaces, which are more convenient for the end-users. The goal is to define ”specific components and design patterns using a visual modeling language for IoT applications” [34]. At the end the solution generates automatically code from the initial models, which can be used for mobile and web applications.

A project - ”BPMN4CPS: A BPMN Extension for Modeling Cyber-Physical Systems” written by Imen Graja, Slim Kallel, Nawal Guermouche and Ahmed Hadj Kacem in year 2016, considers the idea of modelling CPS processes through extended BPMN. The project tackles many different extensions for the BPMN standard, which include the relevant CPS properties and capabilities. The process logic is split into three main parts: cyber part, controller part and physical part. All these processes are shaped as pools in the extended BPMN structure and offer different executable activities. The BPMN meta-model is also extended with new task types, specifically designed for CPS processes: cyber task, manual task and physical task.

5: Implementation

This chapter explains in details how the ICT framework and the CPS interfaces are implemented and what parts they consist of. The system can be split logically into several general modules. The ICT framework is built according to the MVC software design pattern. Figure 5.1 gives an overview of the whole architecture and the relationships between the different modules.

5.1 Database

The database structure represents the ICT framework needs. The general idea is to generate such hierarchy, that can be easily extended on later stage, without the need to change the current architecture. It is important that the structure shouldn't limit the degree of freedom, which the developers need, to make proper relationships between the tables. Figure 5.2 gives an general overview of all tables and their relationships. There is a table for each type of model: Environment - "environmentModels" and Scenario - "scenarioModels". As follows the Scenario models contain CPS Pools - "scenarioPools", the CPS Pools contain elements - "scenarioElements" (CPS Start Event, CPS End Event, CPS Tasks, CPS Exclusive gateways, CPS Non-exclusive gateways) and the elements contain commands - "scenarioElementCommands" (in the form of triples - subject, predicate, object). The table "scenarioNextElements" describes the element sequence inside a Scenario model. If there are more than one next elements, after particular element, there are two separate records in the table, with two different "scnel_next_scel_id" values. The table "scenarioElementTypes" contains all of the element types mentioned above (CPS Start Event, CPS End Event, CPS Task, CPS Exclusive gateway, CPS Non-exclusive gateway), the only difference is that the CPS Task element is split into two sub-types - Collaboration and Function. This split has specific important role behind it, which will be explained in the Section 5.5. This table is the only static table from the whole database, which can be edited only by administrator. A new record

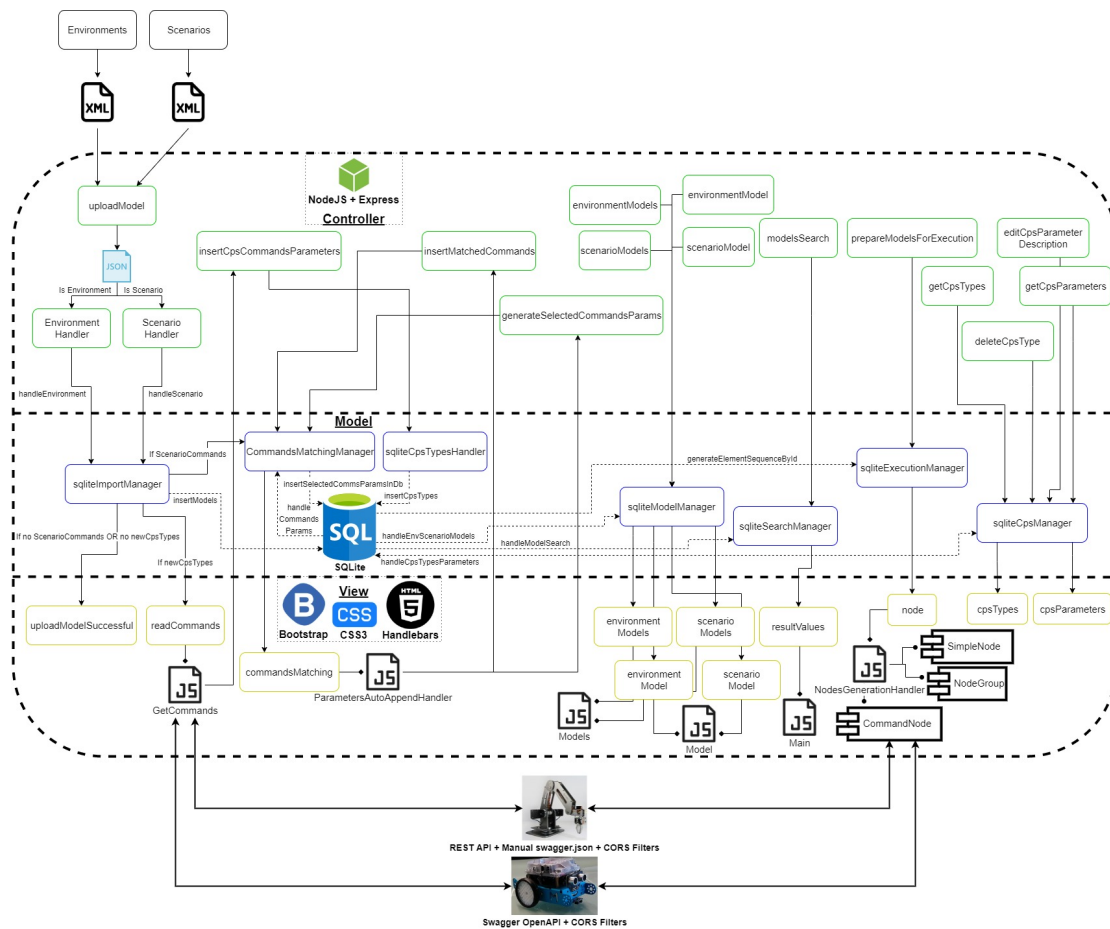


Figure 5.1: Software Architecture Overview

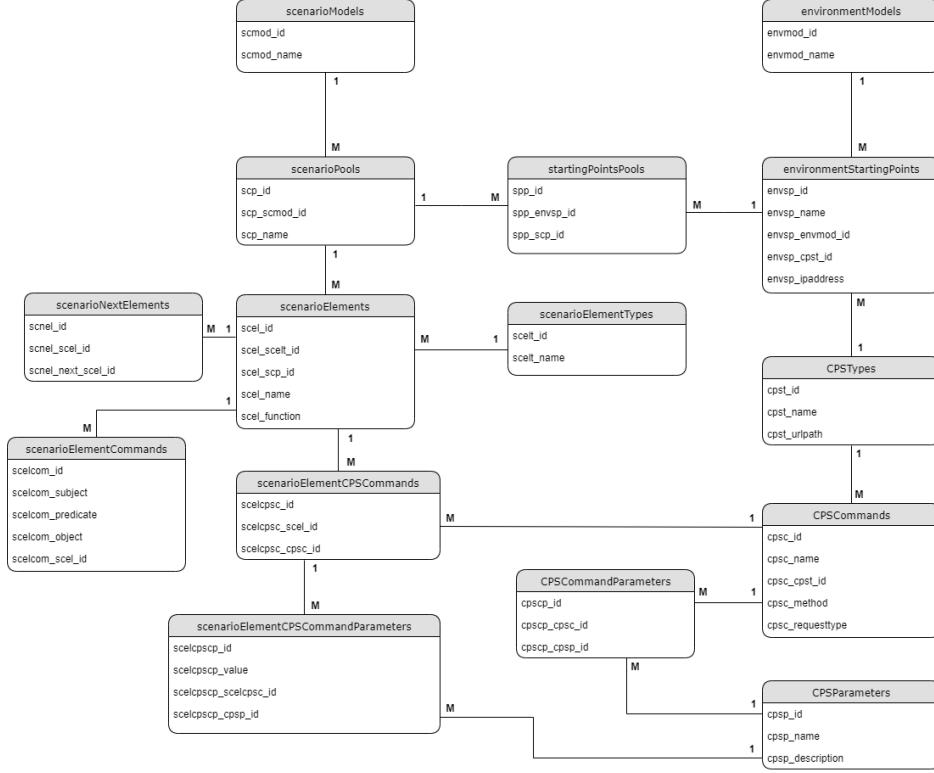


Figure 5.2: SQLite Database

should be inserted in this table, in case of a new type of element is invented and added to the Scenario models. On other side the Environment models contain CPS Starting Points - "environmentStartingPoints". The connecting point between the both types of models is the middle table "startingPointsPools". As the user connects CPS Pools from Scenario models with CPS Starting Points from Environment models, a table between these two elements is sufficient. The relationship is many-to-many, which fulfills the requirement to be able to execute one Environment model with many different Scenario models and vice versa. The CPS Starting Points may contain CPS Types - described in table "CPSTypes". The CPS types are actually the different types of devices registered on the app. It is assumed, that all CPS of the same type have the same interface properties (endpoints, parameters, basePath). Every CPS Type has particular commands - contained in table "CPSCommands" (CPS commands are actually the extracted CPS endpoints from the swagger JSON file contained in the CPS interface). These commands also have parameters - in table "CPSParameters". There is a middle table between the commands and the parameters - "CPSCommandParameters", as in many cases there is one parameter belonging to more than one command (endpoint), and also the commands (endpoints) can have many parameters. The relationship between the Scenario elements

```
function initUpload(app) {
  app.get('/uploadmodel', renderWelcomeUpload);
  app.post('/uploadmodel', upload.single('fileUpload'), uploadModel);
  app.get('/commandsmatchingparameters/:commandId', generateSelectedCommandsParameters);
  app.post('/commandsmatched', insertMatchedCommands);
  app.get('/uploadsuccessful', uploadSuccessful);
  app.post('/insertcpscommandsparameters', insertCpsCommandsParameters);
}
```

Figure 5.3: Express Routing

and the CPS Commands/Parameters are the tables "scenarioElementCPSCommands" and "scenarioElementCPSCommandParameters". These tables actually contain the exact endpoint that should be requested when the execution of the model reaches one particular element. It will be explained later how these tables are filled up.

5.2 ICT Framework structure

The structure of the ICT framework is built, according to the MVC software design pattern. MVC divides an application program logic into three layers - Model, View, Controller. To achieve that, Express web application framework is used. The Express is an important part of every web and mobile application as described in Section 4.6. It organizes the ICT framework structure with HTTP request handler, transforming it to a MVC architecture. It supports the whole process from the routing, through the database model, to the view engine.

5.2.1 Express and Handlebars

The Express is initialized, right from the start in a variable "app", inside the "index.js" file. It defines the whole structure of the ICT framework. The variable requests and inherits the Express main function. All Express properties are applied to this variable. The first important property is to add the public directory to the app structure. The public directory will be used to store all frontend JavaScript, CSS and image files. It is defined as a static path. Next, a view engine is configured and appended to the app variable, together with the main layout view file. The chosen view engine for this master thesis is Handlebars. It provides semantic templates with many different features as described in Section 4.7. These templates have the same structure like normal HTML files, but also with extra functionality. Moreover, because of the constant data exchange between frontend and backend, JSON encoded bodies should be enabled.

```

{{#each models}}
<div class="row element-row">
  <div class="col-2">{{id}}</div>
  <div class="col-4">{{name}}</div>
  <div class="col-4">{{#each startingPoints}}{{envsp_name}} {{/each}}</div>
  <div class="col-2"><a class="btn btn-primary btn-sm" role="button" href="/environmentmodel/{{id}}">Details</a>
</div>
</div>
{{/each}}

```

Figure 5.4: Handlebars Block Expression

Figure 5.3 is an example for Express routing (part from the Upload module in Section 5.3), or more precisely how the framework endpoints respond to client requests. When a particular endpoint (URI) is requested, the corresponding function is executed. The other modules, which are part of the implementation of the ICT framework, are included and initialized with the Express app. The database models are split into separate classes and requested inside each routing function. The same way every Handlebars view engine is requested inside the routing function. One of the major benefits of Handlebars compared to normal HTML, is the ability to provide embedded expressions. All expressions are enclosed inside "`{{expression}}`". Often in this master thesis block expressions are used. They have the same enclosed form, but also provide opening and closing part, like shown on Figure 5.4. The block expression is enclosed inside `{{each}}{{/each}}` loop structure for traversing items in a collection. Inside this block expression, the properties of each item can be accessed. The developer is able to write his own expressions, in the form of view helpers. There are also built-in "if", "else" expressions, which are used for the generation of the elements by the initialization of the Scenario models (part of Section 5.4).

5.2.2 Models search

There is an integrated search bar inside the navigation of the ICT framework. It allows the users, regardless of their current location on the platform, to be able to search for all kind of models. In order to achieve that, the search bar is implemented to generate database requests on every key input. It searches for patterns, which are part of the name of the models. As it is more convenient to search for both types of models - Environments and Scenarios, an union between both tables is executed in the SQL query. In such cases, where an user inputs some content and it is directly queried from the database, a SQL injection can be easily applied. To prevent that, parameterized queries, or more specifically placeholders are used overall for all SQL queries. This way

the database is protected and the performance is improved. After some value is inserted into the search bar, a query is automatically performed and the result is shown as data list under the search bar. The user can click and select the model he wants. He is automatically redirected to the page of the model.

5.2.3 CSS3

CSS3 contributes a lot for the interactivity and the responsiveness of the ICT framework. The first property, which is used practically everywhere, is the automatic calculation of the font size according to the user device. This way depending on the resolution and screen size, the font size is adjusted. Another great feature is the media queries CSS3 property. Like it was explained in Section 4.11 the media queries play an important role in the responsive web design. In this master thesis a media query is used, to adjust the navigation according to user screen size. It transforms the navigation bar into collapsible drop-down menu on lower resolutions and adjusts the elements size. Another very important feature is the sizing of all measurements in "rem" units, instead of pixels. The "rem" unit is used to scale the page elements, according to the font size of the root elements. In other words, as described earlier, as the font size is adjusted to the screen size and resolution, the space and the size of the elements is also changed. Another CSS3 property, which is used overall, is the "Rounded Borders". It makes the edges of all elements rounded and visually much more pleasing for the users. it is used basically for all shapes and elements, which are some kind of containers. When the models are imported into the ICT framework, a virtual copy of them is available. All the original ADOxx model elements, are rebuild with the help of CSS.

5.2.4 Bootstrap

In this master thesis Bootstrap is often used. As it was explained in Section 4.11, the most commonly used Bootstrap features are the grid system, the navigation and the modal windows. Figure 5.5 shows an example of the grid system used for the Environment models.

The grid system is used for all table-like views (Environment and Scenario models, Commands-Endpoints matching, CPS Types and Parameters). All these modules will be described in the next sections.

The Bootstrap navigation is part of the layout. It consists of OMiLAB logo, Introduction, Models drop-down with both types of models, Models Upload, CPS Types,

```

<script src="/js/Models.js"></script>
<div class="container mb-4">
  <h3>Environment Models</h3>
</div>
<div id="models" class="container">
  <div class="row header-row py-1">
    <div class="col-2">#</div>
    <div class="col-4">Name</div>
    <div class="col-4">Starting Points</div>
    <div class="col-2">Details</div>
  </div>
  {{#each models}}
  <div class="row element-row">
    <div class="col-2">{{id}}</div>
    <div class="col-4">{{name}}</div>
    <div class="col-4">{{#each startingPoints}}{{envsp_name}} {{/each}}</div>
    <div class="col-2"><a class="btn btn-primary btn-sm" role="button" href="/environmentmodel/{{id}}">Details</a>
  </div>
  {{/each}}
</div>

```

Figure 5.5: Bootstrap Grid

CPS Parameters and the search bar. The navigation resizes automatically according to user resolution and device size.

The Bootstrap modal windows are used for confirmation of deletion of models. The window appears in front of all other HTML elements. It positions itself and resizes automatically in the middle of the screen even on smaller devices.

5.3 Upload and commands-endpoints matching

The upload module is separated into four sub-modules:

- XML-Parser
- Insert of extracted data
- Read swagger JSON from CPS
- Model commands - CPS endpoints matching

5.3.1 XML-Parser

The whole process of uploading a particular model starts on the frontend side of the app. The user selects some model and uploads it into the platform. In order to be able to convert this model into some other format, first the sourcing file needs to be saved somewhere. It is assumed in most of the cases that the file is saved in some public directory inside the ICT framework. The main issue with this approach is, that the number of files will increase progressively, so they need to be deleted on some point. On

```

function uploadModel(req, res) {
  const json = JSON.parse(convert.xml2json(req.file.buffer));

  const modelElements = json.elements[1].elements[0].elements[0].elements;
  const modelName = json.elements[1].elements[0].elements[0].attributes.name;
  const modelType = json.elements[1].elements[0].elements[0].attributes.modeltype;

  if (modelType === 'CPS Scenario Model') {
    let elementsForDB = ScenarioHandler.handle(modelElements, modelName);
    sqliteImportManager
      .handleScenario(elementsForDB)
      .then(scenarioCommands => {
        if (typeof scenarioCommands !== 'undefined' && scenarioCommands !== null && scenarioCommands.length > 0) {
          CommandsMatchingManager.handleCommands(scenarioCommands)
            .then(commands => {
              res.render('matching/commandsMatching', commands);
            })
            .catch(err => res.render('error', { error: err }));
        } else {
          res.render('upload/uploadModelSuccessful', { message: VIEW.SCENARIO_MODEL_WITHOUT_TRIPLES });
        }
      })
      .catch(err => res.render('error', { error: err }));
  } else if (modelType === 'CPS Environment Model') {
    let elementsForDB = EnvironmentHandler.handle(modelElements, modelName);
    sqliteImportManager
      .handleEnvironment(elementsForDB)
      .then(newCpsTypes => {
        if (typeof newCpsTypes !== 'undefined' && newCpsTypes.length > 0) {
          res.render('upload/readCommands', { newCpsTypes: newCpsTypes });
        } else {
          res.render('upload/uploadModelSuccessful', { message: VIEW.ENVIRONMENT_MODEL_UPLOADED });
        }
      })
      .catch(err => res.render('error', { error: err }));
  } else {
    res.render('error', { error: VIEW.ERROR_SELECTED_MODEL_NOT_CPS_SCENARIO_OR_ENVIRONMENT_MODEL });
  }
}

```

Figure 5.6: Upload Model

the other side, it is also important to consider if the original XML file is needed at all after it is converted. In order to avoid this issue and to increase the performance of the whole process, a decision was taken, that all files can be just saved in the memory in the form of a buffer and deleted right after conversion. An external library "xml-js", which was described earlier in section 4.8 is used for the conversion of the XML model to JSON. The library converts the data in very specific way - every branch of the XML tree is transformed to a JSON array. As described earlier, this approach was very useful with the ADOxx models, as in many cases, there are elements from different types on the same level of the XML tree. After the conversion is successfully finished, every branch can be easily accessed from the new generated JSON structure.

5.3.2 Insert of extracted data

After a model is converted to JSON, the important properties are extracted from it - model name, model type and all related elements. First the model type is checked, so that the platform can automatically recognize the next steps. If the model type is not recognised, the user is redirected to an error page and should upload the model again with the right properties. If the model is correct, the whole extracted data is passed to a Handler class - "EnvironmentHandler" or "ScenarioHandler", depending on the type

of the uploaded model. The Handler class has a specific role - to generate a new JSON structure containing the model name and all related sub-elements, which are important for further implementation.

By the Environment models, such are the CPS Starting Points. They have couple of important properties - CPS Type, IP Address. If a CPS Type is selected, inside the model, but no IP Address is provided, again an error occurs on the frontend and the upload process can't continue.

By the Scenario models, the process is much more complicated, as there are more elements included in them. The generated JSON structure contains model name, CPS Pools, CPS Tasks, CPS Start Event, CPS End Event, CPS Exclusive gateways, CPS Non-exclusive gateways, structure describing which element (CPS Start/End Event, CPS Task) is inside which CPS Pool and the sequence of the elements inside the model. First the CPS Pools are extracted. Except their role as containers, they also provide the connection between the Scenario and Environment models. In order for the process to work properly, it is very important, that no CPS Pool is connected more than once to the same Environment model. The assumption is that every CPS Pool can be connected to unlimited many Environment CPS Starting Points (representing a CPS), but from different models. As the ADOxx doesn't give an opportunity to restrict this option, from inside of the model, the handler checks the connections and produces an error screen. The next step is to extract all CPS Tasks and their properties. The CPS Tasks have three important properties - name, task type and commands (in the form of triples - Subject, Predicate, Object). For further implementation, also an empty array with next elements is added to the structure, as it is expected that every CPS Start Event, CPS Task, CPS Exclusive gateway and CPS Non-exclusive gateway has a next element. The task type property can be "Not specified", "Function" or "Collaboration". The usage of these types will be explained in Section 5.5. The CPS Start/End Events have the same properties like the CPS Tasks, with the difference that their element type is either CPS Start or End Event. The next important elements are the CPS Exclusive and Non-exclusive gateways. They contain name, type and function. The function property can acquire two values - 0 and 1. One means that it is an opening and zero that it is a closing gateway. After all of the elements are extracted and placed in their own JSON structures, the next important step is to define, which element is inside which pool. In order to do that the Scenario models provide a connection mechanism, defining if every element has a parent. These parents are extracted and added to the corresponding

```

handleEnvironment(data) {
  this.db = new sqlite3.Database('./app/database/omilab.db');

  return Promise.resolve()
    .then(() => this.checkEnvironmentModelByName(data.name), error => this.handleErrorAndCloseDB(error))
    .then(() => this.insertEnvironmentModel(data.name), error => this.handleErrorAndCloseDB(error))
    .then(environmentModelId => this.checkCpsTypesExist(data.startingPoints, environmentModelId), error => this.handleErrorAndCloseDB(error))
    .then(({ environmentModelId, newCpsTypes }) => this.getCpsTypes(environmentModelId, newCpsTypes), error => this.handleErrorAndCloseDB(error))
    .then(
      ({ environmentModelId, newCpsTypes, cpsTypesRows }) =>
        this.insertEnvironmentStartingPoints(data.startingPoints, environmentModelId, newCpsTypes, cpsTypesRows,
        error => this.handleErrorAndCloseDB(error)
      )
    )
    .then(newCpsTypes => this.returnCpsTypesWithoutUrlpathAndCloseDb(newCpsTypes), error => this.handleErrorAndCloseDB(error));
}

```

Figure 5.7: Insert Environment Model

elements. Moreover, the sequence of the elements need to be derived. The Scenario models offer again a structure, defining in what order the elements are connected inside the model. These connections are extracted and placed to the corresponding elements.

After the whole structure with the JSON elements is ready, it is passed again to the "sqliteImportManager", which main role is to insert the data into the database. Because of the asynchronous nature of JavaScript, each query is executed sequentially. For this purpose promises are used.

The promises execute each function, wait for it to finish and to return some output value. Then they continue with the next function, giving as input the received value. One promise has three possible states: resolved, rejected or pending. Resolved means that the function is executed successfully and the value is received. Rejected means that some error occurred during execution and Pending means that the function is still executing.

The class "sqliteImportManager" is split into two main parts - "handleEnvironment" and "handleScenario". Each of these parts is a sequence of promises, correspondingly for each type of model. In this sequence, functions are called and executed, where each function gives some output for the next one.

By the Environment models, first the model name is checked. If model with same name already exists in the database, error occurs on the frontend. If such model doesn't exist, the model name is inserted in table "environmentModels". The function "checkCpsTypesExist" has similar role. It selects all CPS types from the database and checks if the new model contains some CPS type, which still doesn't exist. If such CPS types are available, they are inserted into the "CPSTypes" table and are sent as output to the next function. Next, the CPS Starting Points are inserted into the database. After this process is also done, the class returns the results to the main function. If there are any new CPS types missing from the database, the user is redirected to a special

```

function requestRobotCommands(url) {
  return new Promise((resolve, reject) => {
    const xhttp = new XMLHttpRequest();
    xhttp.onload = function() {
      if (xhttp.readyState !== 4 || xhttp.status !== 200)
        reject(
          JSON.stringify({
            status: xhttp.status,
            statusText: xhttp.statusText
          })
        );
      const response = JSON.parse(this.responseText);
      if (typeof response === 'undefined' || response.length === 0)
        reject(
          JSON.stringify({
            status: xhttp.status,
            statusText: xhttp.statusText
          })
        );
      resolve(response);
    };
    xhttp.onerror = function() {
      // handle error
      reject(
        JSON.stringify({
          status: xhttp.status,
          statusText: xhttp.statusText
        })
      );
    };
    xhttp.open('GET', url);
    xhttp.send();
  });
}

```

Figure 5.8: Request CPS Endpoints

view. This view contains all new CPS types. For each CPS, there is a description like CPS name and host address. There is also an input field, where the user should enter the corresponding path to the swagger JSON file of this CPS. The swagger JSON file is a self-describing OpenAPI structure, which contains all REST API request types, endpoints and parameters. If there is more than one CPS from the same type, only one description and input are available for this CPS, as it is assumed that all CPS from the same type have a common interface architecture.

5.3.3 Read swagger JSON from CPS

In order to retrieve all endpoints from particular CPS, first its interface should be extended with the expected unified structure. The interface must be built with Swagger OpenAPI. There is also a workaround - to create manually the swagger.json file and to describe all existing endpoints, parameters and basePath according to the original Swagger schema. The file can be put in some public directory, making it available for external requests.

When the user enters the path to the actual CPS swagger file and clicks "Get Commands", a JavaScript function on the frontend side of the app is executed. First, a

```

handleCpsTypes(robotInfo) {
  this.db = new sqlite3.Database('./app/database/omilab.db');

  return Promise.resolve()
    .then(() => this.updateCpsTypeUrlpath(robotInfo), error => this.handleErrorAndCloseDB(error))
    .then(() => this.insertCpsParameters(robotInfo), error => this.handleErrorAndCloseDB(error))
    .then(() => this.insertCpsCommands(robotInfo), error => this.handleErrorAndCloseDB(error))
    .then(insertedCpsCommands => this.getCpsParameters(insertedCpsCommands), error => this.handleErrorAndCloseDB(error))
    .then(
      ({ cpsParametersRows, insertedCpsCommands }) => this.insertCpsCommandParameters(cpsParametersRows, insertedCpsCommands, robotInfo),
      error => this.handleErrorAndCloseDB(error)
    )
    .then(() => this.db.close(), error => this.handleErrorAndCloseDB(error));
}

```

Figure 5.9: Insert CPS Types

regular expression is used to normalize the path to the file, no matter how the user entered it. The idea is to clear the path from unnecessary signs, which will brake the request. The button "Get Commands" is temporarily disabled and its text is changed to "Sending...", so the user can understand that the process is already running. Like it is shown on Figure 5.8 the ICT framework tries to execute a GET request at the given full path and to retrieve the endpoints and the CPS properties. JavaScript contains default method for requests called "XMLHttpRequest". Again the request is closed inside promise, which resolves only if the target file is successfully read. Otherwise the promise is rejected, the button is enabled again and his text is changed to "Failed! Try again".

If the executed request is successful and the file is retrieved, the next step is to extract the important values. For this purpose a new JSON structure is generated, which will be filled with the derived properties. Each endpoint can have a method from three possible types - GET, POST or PUT. DELETE is not considered as possible method for endpoint at this point. Every endpoint can contain method, requesttype and parameters. The parameters on their side can be as many as needed and can contain parameter name and description. After the whole data is extracted and saved into the new JSON structure the result is sent back to the main function, where the next step of the promise is executed.

Now, after the necessary data is available. it needs be inserted into the database. To achieve this, a new XMLHttpRequest is performed. It makes a request to the backend side of the app and sends the new generated JSON structure.

As shown on Figure 5.9, the whole process is again closed inside a promise chain. First the table "CPSTypes" is updated with the actual UrlPath. The next step is to insert all retrieved parameters and their descriptions into table "CPSPParameters". After this process is also done, the endpoints need to be inserted into "CPSCCommands" table. As it was shown earlier in the database schema, there is a middle table between the CPS commands and parameters. During the implementation with different types of

```

handleScenario(data) {
  this.db = new sqlite3.Database('./app/database/omilab.db');

  return Promise.resolve()
    .then(() => this.checkScenarioModelByName(data.name), error => this.handleErrorAndCloseDB(error))
    .then(() => this.checkStartingPointsPoolsReference(data.pools), error => this.handleErrorAndCloseDB(error))
    .then(() => this.insertScenarioModel(data.name), error => this.handleErrorAndCloseDB(error))
    .then(scenarioModelId => this.insertScenarioPools(data.pools, scenarioModelId), error => this.handleErrorAndCloseDB(error))
    .then(poolsInserted => this.insertStartingPointsPools(data.pools, poolsInserted), error => this.handleErrorAndCloseDB(error))
    .then(poolsInserted => this.getScenarioElementTypes(poolsInserted), error => this.handleErrorAndCloseDB(error))
    .then(
      ({ scenarioElementTypesRows, poolsInserted }) => this.insertScenarioElements(data, scenarioElementTypesRows, poolsInserted),
      error => this.handleErrorAndCloseDB(error)
    )
    .then(elementsInserted => this.insertScenarioNextElements(elementsInserted), error => this.handleErrorAndCloseDB(error))
    .then(elementsInserted => this.insertScenarioElementsCommands(elementsInserted, data), error => this.handleErrorAndCloseDB(error))
    .then(commandsInserted => this.getCommandsWithCorrespondingElementsAndPools(commandsInserted), error => this.handleErrorAndCloseDB(error))
    .then(scenarioCommands => this.returnCommandsAndCloseDb(scenarioCommands), error => this.handleErrorAndCloseDB(error));
}

```

Figure 5.10: Insert Scenario Model

CPS, it was noticed that many endpoints use the same parameters, like for example - "moveForward" has parameters time and speed, defining how long and with what motor speed, they need to move. The same parameters were used almost by all other endpoints. The best approach in such situation is to define a middle table (many-to-many relationship), where one parameter can belong to many different endpoints and also one endpoint can have many different parameters. After the relationships between commands (endpoints) and parameters are also inserted into the table "CPSCCommandParameters", the backend part is ready and the response is sent back to the frontend. If some error occurred during execution, the user is redirected to an error page. If the process was successful, the "Get Commands" button text is changed to "Ready!" and no further steps are required. The same process applies to all new CPS types.

5.3.4 Model commands - CPS endpoints matching

The both modules "Upload" and "Commands-Endpoints matching" are closely related. They are both part of the Scenario model upload process.

The main difference from the Environment models upload is, that the Scenario models contain much more different elements, which also have to be inserted sequentially in order to build the right connections between the elements.

First, the model name is checked, if it already exists in the database. If such model with the same name is found, error occurs on the frontend. The next step is to check the relationships between Scenario CPS Pools and Environment CPS Starting Points. If there is some CPS Pool connected to a CPS Starting Point from Environment model, which still doesn't exist in the database an error occurs and the user is redirected. This step is very important for the whole process to work properly, because if the Environment model is not in the database, proper records in the middle table can't be inserted.

In order to check the relationships correctly, Environment model name + CPS Starting Point name are used. The next step in the sequence is to insert the Scenario model name into "scenarioModels" table. When this process is done, the CPS Pools need to be inserted into "scenarioPools" table. When the CPS Pools are also available, the process continues with the relationship between both models - the middle table "startingPointsPools" between Scenario CPS Pools and Environment CPS Starting Points. After these records are in the database, in order to insert the elements, first all element types need to be retrieved. They are contained in the table "scenarioElementTypes". With these in hand, all elements can be correctly inserted. The query is executed for every type of element (CPS Start Event, CPS End Event, CPS Task, CPS Exclusive and CPS Non-exclusive gateway). When the elements are ready, it is important to insert their sequence, corresponding to the Scenario model into the "scenarioNextElements" table. It is expected that every element except the last one (CPS End Event) has an following one. Next, the commands inside the elements need to be inserted in the corresponding "scenarioElementCommands" table. At the end of the promise chain, all elements containing commands are fetched with their CPS Pools and are sent back to the main function.

If the Scenario elements (CPS Start Event, CPS Tasks, CPS End Event) contain commands, in the form of triples - subject, predicate, object, the user is redirected. Here the module "Commands-Endpoints matching" is involved. The user has to select, which element with triples correspond to which CPS endpoint. In order to present so many different elements on a single page and to keep the page responsive, a Bootstrap grid system is used. There are two important HTML elements on this page, which execute different frontend JavaScript functions. The first one is - selecting commands (endpoints) from the drop-down elements.

When the user clicks on a drop-down element and selects a corresponding endpoint for a particular triple-expression, the JavaScript function "renderParametersForCommand" is executed. Its role is to select all parameters available for this specific endpoint and to render them back in the HTML inside a separate div element. In order to retrieve all parameters, the function first performs a XMLHttpRequest and sends the selected endpoint to the backend side of the application. On the backend a SQLite query is executed and all connected parameters to this endpoint are fetched. Then, they are sent back to the frontend, where the JavaScript function generates HTML elements (Figure 5.12) and appends them inside specific div elements in the last-right column of

```

// selects all parameters available for a specific command and renders them in separate div
// source: commandsMatching.hbs -> dropdown menu with commands
function renderParametersForCommand(optionEl, commandId, cpsTypeId) {
  const xhttp = new XMLHttpRequest();
  const elementEl = getParentByClass(optionEl, 'element-row');
  const parameterWrapperEl = elementEl.querySelector(`div.cps_type_wrapper_${cpsTypeId}`);

  const optionEls = optionEl.parentElement.querySelectorAll('dropdown-item');
  optionEls.forEach(ddOptionEl => ddOptionEl.classList.remove(CSS_OPTION_SELECTED));
  optionEl.classList.add(CSS_OPTION_SELECTED);

  parameterWrapperEl.innerHTML = '';

  xhttp.onreadystatechange = function() {
    if (this.readyState !== 4 || this.status !== 200) return;

    const response = JSON.parse(this.responseText);
    if (typeof response === 'undefined' || response.length === 0) return;
    else fillParameterWrapperEl(response, parameterWrapperEl);
  };
  xhttp.open('GET', '/commandsmatchingparameters/' + commandId, true);
  xhttp.send();
}

```

Figure 5.11: Render Parameters for Endpoint

the Bootstrap grid system. For every available parameter an input field with name and description is appended.

The second frontend JavaScript function is executed, when the user clicks the "Save" button, after all necessary endpoints are selected and their parameter inputs are filled up. When the "Save" button is clicked, a function "postCommandsWithParameters" is called. This function has a specific role. It extracts all important element values from the HTML page (element id, scenario element commands, CPS endpoint, parameters) and generates a JSON structure with them. Then, the function executes a XMLHttpRequest and sends them to the backend. The backend inserts this structure into the database in the form of a promise chain. First the selected endpoints are inserted into the "scenarioElementCPSCCommands" table with the corresponding element id. On the next step, the parameters are inserted into the "scenarioElementCPSCommandParameters" table. With this final step the upload of the models is finished and they are now ready to be executed.

5.4 Models overview

The models module is separated into four sub-modules:

- Environment models
- Scenario models
- Single Environment model

```
function fillParameterWrapperEl(elements, parameterWrapperEl) {
  const parameterWrapperEls = [];
  elements.forEach(element => {
    const wrapperEl = document.createElement('DIV');
    const inputEl = document.createElement('INPUT');
    const divEl = document.createElement('DIV');

    wrapperEl.className = `parameter_${element.cpspId}`;
    inputEl.placeholder = element.cpspName;
    inputEl.type = 'text';
    inputEl.min = 1;
    inputEl.required = true;
    inputEl.className = 'form-control parameter';
    divEl.innerText = element.cpspDescription;

    wrapperEl.appendChild(inputEl);
    if (element.cpspDescription !== null) wrapperEl.appendChild(divEl);
    parameterWrapperEls.push(wrapperEl);
  });

  parameterWrapperEls.forEach(el => parameterWrapperEl.appendChild(el));
}
```

Figure 5.12: Fill Parameters-div

```
function environmentModels(req, res) {
  sqliteModelManager
    .handleEnvironmentModels()
    .then(environmentModelsRows => res.render('models/environmentModels', { models: environmentModelsRows }))
    .catch(err => res.render('error', { error: err }));
}
```

Figure 5.13: Environment Models

- Single Scenario model

5.4.1 Environment and Scenario models

The first two sub-modules are displayed in a table form. They present all models from one particular type. The corresponding models and their properties are fetched from the database and rendered (Figure 5.13). Again the Bootstrap grid system is used.

In order to display all models from a selected type, first the models need to be retrieved from the corresponding table with all belonging properties (Figure 5.14). This way an user has a better overview of the models and their related elements. The last column of the grid system is a link, which sends the user to the selected model.

5.4.2 Single Environment model

Both, single Environment and single Scenario models are selected by model id. When an Environment model is requested, the backend retrieves this model from the database. The whole process is again in the form of a promise chain. First, the ICT framework

```

fetchAllEnvironmentModels() {
  // fetch all environment models with their starting points
  return new Promise((resolve, reject) => {
    const sqlEnvironmentModels = QUERIES.SELECT_ALL_ENV_MODELS;
    this.db.all(sqlEnvironmentModels, [], (err, environmentModelsRows) => {
      if (err) {
        reject(err.message);
      }
      const sqlEnvStPoints = QUERIES.SELECT_ALL_STARTING_POINTS_BY_ENV_MODEL_ID;
      const stPointsPromises = [];
      environmentModelsRows.forEach(environmentModelsRow => {
        const stPointsPromise = new Promise((stPointResolve, stPointReject) => {
          this.db.all(sqlEnvStPoints, [environmentModelsRow.envmod_id], (err, startingPointsRows) => {
            if (err) {
              stPointReject(err.message);
            } else {
              stPointResolve({ id: environmentModelsRow.envmod_id, name: environmentModelsRow.envmod_name, startingPoints: startingPointsRows });
            }
          });
        });
        stPointsPromises.push(stPointsPromise);
      });
      Promise.all(stPointsPromises).then(resolve, reject);
    });
  });
}

```

Figure 5.14: Fetch all Environment Models

checks if a model with such id exists in the table. If it is not present, an error occurs and the user is redirected. If the model is available, all connected Scenario models to this Environment model are fetched. On the next step, all CPS Starting Points and their CPS types are selected. When this process is finished the results are sent back to the main function.

The CPS Starting Points are received in some random order and need to be sorted, before visualised to the user. For this purpose, a simple sorting function is used. It compares the names of the CPS Starting Points and sorts them alphabetically. After the model properties are sorted, they are finally sent to the frontend and presented to the user.

The model overview is split into three collapsible sections. For this purpose Bootstrap Collapse (Accordion) is used. By the Environment model, in the first section are described all CPS Starting Points and their properties. The second section contains all possible Scenario pair-models, which this model can be executed with. In order for the execution process to work properly, always two models are needed - one Environment and one Scenario. As described earlier, during modelling the user can connect a single CPS Pool to many different CPS Starting Points, from different Environment Models. That makes the connection between both types of models - many to many. So in this section, the user is able to see which exactly are the connected pair-models to the selected model and execute them. The last section "Delete Model" is implemented with Bootstrap Modal. It opens a modal window for confirmation of the deletion process.

```

sortScenarioElements(modelProperties) {
  return new Promise(resolve => {
    configElementTypes.getElementTypes().then(configElementTypes => {
      const elementsById = this.prepareElementSequence(modelProperties.elementSequenceRows);
      const preparedElementsById = this.prepareElementsForView(elementsById, configElementTypes);
      const startingElement = _.find(preparedElementsById, element => element.typeId === configElementTypes.START);
      const sortedElements = [];
      this.recursiveCreateSequence(preparedElementsById, configElementTypes, [startingElement.id], sortedElements);
      modelProperties.sortedElements = sortedElements;
      delete modelProperties.elementSequenceRows;
      resolve(modelProperties);
    });
  });
}

```

Figure 5.15: Sort Scenario Model Elements

5.4.3 Single Scenario model

By the Scenario model, the process goes in similar way. First, it is checked if model with such id exists in the database. If the model is found, all connected Environment models are fetched and sent as output to the next function in the promise chain. On the next step, all properties of this Scenario model like CPS Pools, elements (CPS Start Event, CPS End Event, CPS Tasks, CPS Exclusive and Non-exclusive gateways), element types and the sequence of the elements inside the model are taken and the whole information is returned to the main function.

Just like by the Environment models, here the sequence of elements is fetched in some random order. In order to sort them, it is important to define a right sorting algorithm, which can place the elements in the same order, like they are actually inside the original Scenario model in ADOxx. This task is not a trivial one, because the model can contain many parallel and decision processes. In case of parallel or decision the sorting algorithm should take first all elements from the first edge of the parallel/decision, then from the second and so on, until the parallel/decision is finished and then move further with the next elements. For this purpose a recursive function is used.

As it is shown on Figure 5.15 first, the element sequence structure fetched from the database is reworked. Currently it is in the form of element_id - next_element_id. If there is a case, where a single element has two following elements, that is represented as two separate records/rows. The elements are rebuilt into a JSON structure, where every element is defined by his unique id (key) and contains as value all needed properties, like name, type, open/close function (if it is parallel or decision) and pool. The next elements inside the structure are represented in the form of array - element_id - [element_id2, element_id3]. On the next step, the new element structure is prepared for visualisation - depending on the element type, a specific attribute is added to every element, which

```

recursiveCreateSequence(preparedElementsById, configElementTypes, currentElementIds = [], sortedElements, waitEdges = []) {
  if (currentElementIds.length > 0) {
    currentElementIds.forEach(id => {
      const el = preparedElementsById[id];
      const currentWaitEdgeIndex = _.findLastIndex(waitEdges, waitEdge => waitEdge.modeTypeId === el.typeId);
      if (currentWaitEdgeIndex !== -1 && el.openClose === 0) {
        let readyEdges = waitEdges[currentWaitEdgeIndex].readyEdges;
        ++readyEdges;
        waitEdges[currentWaitEdgeIndex].readyEdges = readyEdges;
        if (waitEdges[currentWaitEdgeIndex].readyEdges === waitEdges[currentWaitEdgeIndex].count) {
          waitEdges.splice(currentWaitEdgeIndex, 1);
          if (el.nextIds.length > 1) {
            if (el.typeId === configElementTypes.PARALLEL || el.typeId === configElementTypes.DECISION) {
              waitEdges.push({ modeTypeId: el.typeId, count: el.nextIds.length, readyEdges: 0 });
            }
          }
          sortedElements.push(el);
          this.recursiveCreateSequence(preparedElementsById, configElementTypes, el.nextIds, sortedElements, waitEdges);
        }
      } else {
        if (el.nextIds.length > 1) {
          if (el.typeId === configElementTypes.PARALLEL || el.typeId === configElementTypes.DECISION) {
            waitEdges.push({ modeTypeId: el.typeId, count: el.nextIds.length, readyEdges: 0 });
          }
        }
        sortedElements.push(el);
        this.recursiveCreateSequence(preparedElementsById, configElementTypes, el.nextIds, sortedElements, waitEdges);
      }
    });
  } else {
    return sortedElements;
  }
}

```

Figure 5.16: Recursive Scenario Model Elements Sequence

has the role of a flag and will be later used during visualisation. After this step is also finished, it is important to find the CPS Start Event element in the Scenario model. This element will be the first one inside the recursive function and every following element will be mapped as a follower to this one.

On Figure 5.16 is displayed the recursive function. The purpose of this function is to generate an array "sortedElements", containing all elements in the right order, just like they are in the source ADOxx model. To achieve that, the function observes when an open parallel/decision (non-exclusive/exclusive gateway) element is reached. The number of the edges of the parallel/decision are saved and passed on to the loop. When a closing parallel/decision is reached, the expected number of edges is reduced with one, until the number of edges reaches zero. This means that the parallel/decision execution is finished and the recursion can continue with the next elements after the closing parallel/decision. If during the processing of one edge, the recursion reaches a new parallel/decision, the same operation starts there, until no further parallel/decision is available. As it is expected every opening parallel/decision element, has a counter-closing parallel/decision element, so it doesn't matter what complexity the model gets.

After the elements are correctly sorted, the result is returned to the main function, which sends them to the frontend for visualisation. The model overview is split again into three collapsible sections. In the first section, the sorted elements are appended.

Through the flag that was assigned earlier, the app can decide what kind of element will be generated and what properties should be added. The elements look equally to the original ADOxx model. The only difference is that they are not drawn in a process with arrows and edges. Every element is with his belonging pool and type icon. The second section contains all of the possible Environment pair-models. The user can select one of them and start the execution process. The third section is the same like by the Environment models - "Delete Model". In order to delete a concrete model, an confirmation modal window is displayed.

5.5 Models execution

The final step in the framework is the execution of the models. Here all the individual pieces described until now are connected. As mentioned earlier for a proper execution, two models from each type are needed - Environment and Scenario. They both need to be connected.

First, as both model ids come as GET parameters, they are parsed as integers. Then the element sequence is generated. All Scenario model elements, their sequence, types and assets are fetched from the database. Next, their CPS corresponding properties are added from the selected Environment model (respectively the CPS Type id, CPS Type name, IP address, URL path). On the next step, for every Scenario element with command, the corresponding CPS Command (CPS endpoint - the endpoint was selected earlier during Scenario model import from user), method (POST, GET or PUT) and request type (JSON, Boolean and so on) are added. Furthermore, the CPS endpoint parameters are also attached to the Scenario elements - in the form of a parameter name - parameter value. Finally, the elements are restructured, where each element is identified by his unique id and the following elements are saved into array. If the request method for some element command is GET, the parameters are replaced with their values in the endpoint with regular expression (In case of a GET request-method, the original endpoint, coming from the swagger JSON file, contains the parameter names inside the URL, so they can be easily replaced with the parameter values). When the new element structure is ready, it is sent to the frontend.

On the frontend side, first an empty div is created. In this div will be placed every output, decision, content in the form of a log during the execution of the models. The whole structure is converted from JavaScript object to JSON string and sent to a function as input parameter. In the function, it is checked if there exists some CPS, which requires

Authentication Token. If such token is needed, the user is asked to enter it. One example for such CPS is the RoboticArm in the OMILAB environment. Every HTML element is generated in the JavaScript function and rendered to the user. An Event Listener is added to a HTML button, to allow the user to place his input and confirm by clicking on the button. The whole set of HTML elements is closed inside a promise, which resolves only when the button is clicked. Until then the function stays on hold. After the button is clicked or if the process is started (depending if there is a CPS needing Authentication Token), all the Scenario element types are fetched from the corresponding table in the database and the first element from the above mentioned structure (sent from the backend) is found. To be able to find the first element, a "Lodash" library is used. The "Lodash" library was described in Section 4.9. The first element from the model is always the CPS Start Event.

On the next step, the first element, the whole structure of elements and the token (if available) are send to a recursive function, which generates the Node sequence. The main idea of this recursive function is to rebuild the generated JSON structure in a way that the first element contains array of next elements, they on their side contain array with the next elements and so on, until the whole sequence is in the form of a tree and on each branch of the tree, there are one or more elements. Each element, depending on its type and properties is put in a particular class. Three main classes have been implemented to fulfill the needs of the ICT framework. During the execution of the recursive function, each element is transformed to one of this classes but the tree structure is kept, so that the next elements are contained in the previous class.

These three classes are respectively:

- NodeGroup
- SimpleNode
- CommandNode

5.5.1 NodeGroup

The NodeGroup is the class, which fulfills the role of the CPS non-exclusive gateways (parallels). The main idea behind it, is to enable step-wise execution of all elements inside a parallel. This means, if there are for example three edges after parallel start, on each step one element will be executed from each edge, until the respective edge is finished and the parallel end is reached. If for example one edge is finished and there

```

function recursiveCreateNodeSequence(config, elementsById, nextElementIds = [], token = '', nodeGroup = null) {
  const nodes = [];
  nextElementIds.forEach(id => {
    let node;
    const el = elementsById[id];
    const hasCommand = !_.isNil(el.command) === false && el.command.length > 0;

    if (el.typeId === config.DECISION) {
      const nextNodes = recursiveCreateNodeSequence(config, elementsById, el.nextIds, token, nodeGroup);
      node = new SimpleNode(el.name, el.typeId, config, nextNodes, nodeGroup);
    } else if (el.typeId === config.PARALLEL && el.function === 0) {
      // Parallel close
      const next = recursiveCreateNodeSequence(config, elementsById, el.nextIds, token, nodeGroup.nodeGroup);
      node = new SimpleNode(el.name, el.typeId, config, next, nodeGroup, nodeGroup);
    } else if (el.typeId === config.PARALLEL && el.function === 1) {
      // Parallel open
      node = new NodeGroup(el.name, nodeGroup);
      const groupNodes = recursiveCreateNodeSequence(config, elementsById, el.nextIds, token, node);
      node.setNodes(groupNodes);
    } else if (hasCommand) {
      const next = recursiveCreateNodeSequence(config, elementsById, el.nextIds, token, nodeGroup);
      const params = [el.command, el.commandMethod, el.commandRequestType, el.commandParameters, el.robotUrl, el.robotIp, token];
      node = new CommandNode(el.name, el.typeId, config, el.cpsTypeName, next[0], nodeGroup, ...params);
    } else {
      const next = recursiveCreateNodeSequence(config, elementsById, el.nextIds, token, nodeGroup);
      node = new SimpleNode(el.name, el.typeId, config, next, nodeGroup);
    }

    nodes.push(node);
  });

  return nodes;
}

```

Figure 5.17: Recursive Scenario Nodes Sequence

are still remaining elements on the other two edges, they will continue executing step-wise, until the parallel end is also reached there. When all edges reach the end parallel element, the NodeGroup is finished and the responsibility for the execution is given back to the next element in the sequence. If a new parallel is reached inside one of the edges during parallel execution, the same process starts there, wherefore the new NodeGroup has the responsibility to execute the elements inside of it step-wise.

The constructor of the NodeGroup receives only two parameters:

- text - name of the element, which will be displayed to the user, during execution
- nodeGroup - if this NodeGroup is part of another outer NodeGroup/parallel

First, it is defined that the next element in the execution workflow is this same class, which means that the class has the responsibility for the workflow, until the parallel execution is finished. The main div (created earlier to visualize the results from the execution) is found by id and assigned to a variable. The NodeGroup class is composed from five functions:

- setNodes(nodes) - This function is requested during the generation of the elements tree in the recursive function. Its role is to set elements, which should be executed step-wise. Depending on the number of edges coming after the parallel open element, all the children elements are added. It also generates an empty array with

```

class NodeGroup {
  constructor(text = '', nodeGroup = null) {
    this.text = text;
    this.nodeGroup = nodeGroup;
    this.isEndGroupNode = false;
    this.next = this;
    this.executionDiv = document.getElementById('execution');
  }

  setNodes(nodes) {
    if (_.isNil(nodes) || nodes.length === 0) console.error('No nodes provided for NodeGroup!');

    this.nodes = nodes;
    this.finishedNodes = [];
  }

  step() {
    this.nodes = this.nodes.filter(n => n.closesNodeGroup !== this).map(n => n.next);
    this.finishedNodes = [];

    if (_.isNil(this.nodeGroup) === false) {
      this.nodeGroup.nodeFinished(this);
    } else {
      const span = document.createElement('SPAN');
      const text = document.createTextNode('Next step in parallel execution');
      this.executionDiv.appendChild(span);
      span.appendChild(text);

      this.nodes.forEach(node => node.start());
    }
  }
}

```

Figure 5.18: NodeGroup1

```

nodeFinished(node) {
  this.finishedNodes.push(node);
  if (this.finishedNodes.length === this.nodes.length) {
    const endGroupNodes = this.nodes.filter(n => n.closesNodeGroup === this);
    const preEndGroupNodes = this.nodes.filter(n => n.next.closesNodeGroup === this);
    if (endGroupNodes.length + preEndGroupNodes.length === this.nodes.length) {
      this.next = endGroupNodes.length > 0 ? endGroupNodes[0].next : preEndGroupNodes[0].next.next;
      this.finish();
    } else {
      this.next = this;
      this.step();
    }
  }
}

start() {
  if (_.isNil(this.nodeGroup)) {
    const span = document.createElement('SPAN');
    const text = document.createTextNode('Parallel execution started');
    this.executionDiv.appendChild(span);
    span.appendChild(text);
  }

  this.nodes.forEach(node => node.start());
}

finish() {
  if (_.isNil(this.nodeGroup) === false) {
    this.nodeGroup.nodeFinished(this);
  } else {
    const span = document.createElement('SPAN');
    const text = document.createTextNode('Parallel execution finished');
    this.executionDiv.appendChild(span);
    span.appendChild(text);

    this.next.start();
  }
}
}

```

Figure 5.19: NodeGroup2

finished nodes, which should be filled up, after particular edge reaches the end parallel.

- `start()` - The start function is called when a parallel execution is started. It is responsible for outputting that on the screen (in the main div) and starting the first series of step-wise elements (each one of them can be either from type `SimpleNode` or `CommandNode`).
- `step()` - The step function is responsible to execute each step in the step-wise execution. First, it checks for every element in each edge, if it is the parallel closing element. If it isn't, the function maps the next elements of the current elements, outputs that the step-wise execution goes further and starts the next elements. If the next element is the closing parallel, the function maps the element after it as next and calls the function `nodeFinished`.
- `nodeFinished(node)` - The `nodeFinished` function adds elements that have reached the end parallel to an array. The idea is to verify if every edge has reached the end parallel and to finish the parallel execution. It checks how many edges have reached the end parallel compared to how many started. It sums the elements that are currently on the closing parallel plus the elements, which next element is the closing parallel and compares them with the starting edges number. If both values are equal the parallel execution is finished and the parallel is closed. The next element is defined as the subsequent element after the closing parallel and the finish function is called. If the both values are not equal, the step-wise execution continues.
- `finish()` - This function is called, when the parallel execution is finished. Its role is to output that on the screen and to start the next element.

5.5.2 SimpleNode

The `SimpleNode` is the most basic class, responsible for all normal elements. Every opening/closing CPS exclusive gateway (decision), closing CPS non-exclusive gateway (parallel) and every kind of normal CPS Task, CPS Start Event, CPS End Event without command is classified as `SimpleNode`.

The constructor of the `SimpleNode` receives several parameters:

- `text` - name of the element, which will be displayed to the user, during execution

```

class SimpleNode {
  constructor(text = '', typeId = 0, config, nextNodes = [], nodeGroup = null, closesNodeGroup = null) {
    this.text = text;
    this.typeId = typeId;
    this.config = config;
    this.nextNodes = nextNodes;
    this.nodeGroup = nodeGroup;
    this.closesNodeGroup = closesNodeGroup;

    this.isDecisionNode = nextNodes.length > 1;
    this.next = this.isDecisionNode ? {} : nextNodes[0];
    this.executionDiv = document.getElementById('execution');
  }

  start() {
    if (!this.closesNodeGroup) {
      const span = document.createElement('SPAN');
      let text = null;
      if (this.typeId === this.config.DECISION) text = document.createTextNode('Start CPS Exclusive Gateway: "${this.text}"');
      else if (this.typeId === this.config.PARALLEL) text = document.createTextNode('Start CPS Closing Non-exclusive Gateway: "${this.text}"');
      else if (this.typeId === this.config.START) text = document.createTextNode('Start CPS Start Event: "${this.text}"');
      else if (this.typeId === this.config.END) text = document.createTextNode('Start CPS End Event: "${this.text}"');
      else text = document.createTextNode('Start CPS Task: "${this.text}"');
      this.executionDiv.appendChild(span);
      span.appendChild(text);
    }

    if (this.isDecisionNode) {
      // User choice between 0 and this.nextNodes.length - 1
      this.askUserForDecision()
        .then(nextNodeIndex => {
          this.next = this.nextNodes[nextNodeIndex];
          this.finish();
        })
        .catch(error => this.handleSimpleError(error));
    } else {
      this.finish();
    }
  }
}

```

Figure 5.20: SimpleNode part 1

- typeId - Scenario element type id of the current element
- config - containing all Scenario element types for comparison
- nextNodes - array with the next elements
- nodeGroup - if this element is part of a NodeGroup/parallel
- closesNodeGroup - if this element is a closing non-exclusive gateway (parallel)

The first important step, which should be defined, during the placement of the element in the corresponding class is, if this particular element is of type opening CPS Exclusive gateway (decision). If the count of the next elements is more than one, the element is recognised as decision. When an element is placed as decision, the next element is unclear, as it depends on user input. The main div is selected and assigned to a variable. The SimpleNode class is composed from four functions:

- start() - The start function is performed during execution. It is responsible for outputting the element on the screen and requesting the function askUserForDecision(), if the element type is opening CPS Exclusive gateway (decision). A corresponding output is generated, depending on the element type. If the element type is decision, a promise is generated and resolved, after the user gives his input.

```

askUserForDecision() {
  return new Promise(resolve => {
    const wrapperEl = document.createElement('DIV');
    wrapperEl.className = 'execution-wrapper';
    const headerDivEl = document.createElement('DIV');
    headerDivEl.className = 'execution-header';
    const headerEl = document.createTextNode('Please choose the next element:');
    wrapperEl.appendChild(headerDivEl);
    headerDivEl.appendChild(headerEl);
    for (let i = 0; i < this.nextNodes.length; i++) {
      const buttonEl = document.createElement('BUTTON');
      const spanEl = document.createElement('SPAN');
      const textEl = document.createTextNode(this.nextNodes[i].text);
      buttonEl.className = 'execution-task';
      buttonEl.addEventListener('click', function() {
        resolve(i);
      });
      const elId = this.nextNodes[i].text.split(' ').join('_');
      buttonEl.id = elId + i;
      wrapperEl.appendChild(buttonEl);
      buttonEl.appendChild(spanEl);
      spanEl.appendChild(textEl);
      if (this.nextNodes[i].typeId === this.config.COLLABORATION) {
        const collaborationIconDiv = document.createElement('DIV');
        collaborationIconDiv.className = 'icon collaboration';
        buttonEl.appendChild(collaborationIconDiv);
      } else if (this.nextNodes[i].typeId === this.config.FUNCTION) {
        const functionIconDiv = document.createElement('DIV');
        functionIconDiv.className = 'icon function';
        buttonEl.appendChild(functionIconDiv);
      }
    }
    this.executionDiv.appendChild(wrapperEl);
  });
}

```

Figure 5.21: SimpleNode part 2

```

finish() {
  if (_.isNil(this.closesNodeGroup)) {
    const span = document.createElement('SPAN');
    let text = null;
    if (this.typeId === this.config.DECISION) text = document.createTextNode('End CPS Exclusive Gateway: "${this.text}"');
    else if (this.typeId === this.config.PARALLEL) text = document.createTextNode('End CPS Closing Non-exclusive Gateway: "${this.text}"');
    else if (this.typeId === this.config.START) text = document.createTextNode('End CPS Start Event: "${this.text}"');
    else if (this.typeId === this.config.END) text = document.createTextNode('End CPS End Event: "${this.text}"');
    else text = document.createTextNode('End CPS Task: "${this.text}"');
    this.executionDiv.appendChild(span);
    span.appendChild(text);
  }
  if (_.isNil(this.nodeGroup) === false) {
    this.nodeGroup.nodeFinished(this);
  } else if (_.isNil(this.next) === false) {
    this.next.start();
  } else {
    const span = document.createElement('SPAN');
    const text = document.createTextNode('Model execution finished!');
    this.executionDiv.appendChild(span);
    span.appendChild(text);
  }
}

handleSimpleError(error) {
  // Update UI, inform user that command has failed and execution will not proceed.
  console.error('Error executing decision node: "${this.text}"');
  console.log('Model execution finished with an error!');
  console.error('Error returned from decision element: ${error}');
}

```

Figure 5.22: SimpleNode part 3

The next element is selected and the workflow continues with the class function `finish()`.

- `askUserForDecision()` - This function is closed inside a promise. It generates a series of HTML elements, which visualise the next possible elements, the same way as they look inside the original ADOxx model. An Event Listener is attached to each element, which activates on click. The user can select one of the next elements and his decision influences the general execution workflow of the model.
- `finish()` - The finish function is called on the end of the start function. Its role is to output, that the framework has finished executing the current element. It checks if the current element is part of some `NodeGroup` or if that's the last element in the model. If this is not the case, it just calls the start function of the next element.
- `handleSimpleError(error)` - The error handling is very important part of the whole process, so in case some error occurs during execution on each step, the `handleSimpleError` function is called. It notifies the user that some error occurred and stops the execution process.

The CPS Exclusive Gateway (decision) is `SimpleNode` type of element, because after decision is taken, there is only one next element (only one child) and also the decision elements (just like CPS Non-Exclusive gateways - parallels) are not allowed to contain any commands.

5.5.3 CommandNode

The `CommandNode` is the last unit in the execution process. It represents a class, which contains command (endpoint) with or without parameters. Such elements can be CPS Tasks, CPS Start Event and CPS End Event.

The constructor of the `CommandNode` receives several parameters:

- `text` - name of the element, which will be displayed to the user, during execution
- `typeId` - Scenario element type id of the current element
- `config` - contains all Scenario element types for comparison
- `cpsTypeName` - name of the CPS type
- `next` - array with the next elements

- nodeGroup - if this element is part of a NodeGroup/parallel
- command - the command/endpoint, which should be requested from the CPS
- commandMethod - POST, GET or PUT
- commandRequestType - JSON or Boolean (coming from swagger JSON)
- parameters - parameters attached to the current command
- robotUrl - basePath of the CPS
- robotIp - IP/Web Address of the CPS
- token - Authentication Token given from user (if needed)
- collision - this parameter is set to indicate, if a collision between two or more CPS is detected

In order for the elements of this class to work properly, all the important parameters, which generate the endpoint URL address should be cleaned from unnecessary characters. These are respectively the IP/Web Address, the basePath and the command/endpoint. A regular expression is used to remove all "/" characters. The ICT framework checks the values of these parameters. If there is some parameter with empty value, an error occurs. Again the main div is selected and assigned to a variable, where the output will be printed. The CommandNode class is composed from seven functions:

- start() - Unlike the classes considered so far the start function of the CommandNode is responsible not only for outputting the current element on the screen, but also to trigger the actual execution of the endpoint. If a collision is detected, it notifies the user. The execution of the endpoint is split, depending on few properties. If the execution is inside a parallel, the task type is "Collaboration" and the CPS type is "mBot", there is a specific case, where the "mBot distance sensor" is used. The function executeRobotDistanceSensorRequestGet is called. This function returns as result the current value of the "mBot distance sensor" in centimeters. A threshold is set to compare the value. If the value is less than thirty-five centimeters, a collision is triggered. A class variable for collision is set to "true" and a five seconds timeout is started. When the timeout is up, the start function is called again. As this process is only active during parallel execution, the assumption is, that there are at least two CPS doing some actions on this step (on two different edges of the

parallel). If one of them detects a collision, it is assumed that the second one will execute his action, during that time. After the timeout, the first CPS will see that the collision is avoided (or will detect collision again and repeat the whole process) and can continue again with the process. If the value is greater than thirty-five, the actual command (endpoint) inside the element is executed. As the "mBot" always uses GET as request method, the function `executeRobotRequestGet` is called.

For every other case, the URL is generated from the above mentioned parameters and the corresponding request is performed. Depending on the request method, there are three different functions, respectively for GET, POST, PUT. For each single request method, also the parameters and the responses are processed in different way.

- `finish()` - The finish function is called at the end of the start function, after the command is executed and the response is received. It outputs, that the framework has finished the execution of the current element. Then it checks if the current element is part of NodeGroup or if it's the last element in the model. In any other case, it calls the start function of the next element.
- `executeRobotDistanceSensorRequestGet()` - This function is used only for CPS of type "mBot", during parallel execution. It performs a HTTP request to the distance sensor of the "mBot". It is closed inside a promise. First, it builds the whole URL address of the CPS from the given parameters. Then, it performs a `XMLHttpRequest` to this URL and gets the "mBot distance sensor" value as a response. If the value is empty or zero, an error occurs. The promise resolves with the given value.
- `executeRobotRequestPost(url)` - This function is closed inside a promise. It performs POST HTTP requests. First, the parameters are concatenated as string. A new `XMLHttpRequest` is performed. If an Authentication Token is required, it is set to the header. The parameters are send during the request.
- `executeRobotRequestPut(url)` - This function is similar to the above one. It performs PUT HTTP requests. The parameters are generated in the form of JSON. Again, if an Authentication Token is required, it is set to the header and the parameters are send during the request.
- `executeRobotRequestGet(url)` - This is the most basic GET HTTP request function. As the parameters were generated earlier, during the generation of the ele-

ment tree inside the endpoint paths, a simple HTTP request is performed to the given URL.

- `handleCommandError(error)` - The `handleCommandError` function is called, if some error occurs during execution on each step. It notifies the user about the type of the error and the name of the current element.

5.5.4 CPS properties

The CPS properties module contains two sub-modules: "CPS Types" and "CPS Parameters". Both are represented with the help of the Bootstrap grid system. The idea of the "CPS Types" is to give the user possibility to see all the available commands for each CPS and to delete CPS types through the interface, for example when the swagger JSON file has changed and the user wants to load again all available endpoints and their parameters. When a CPS type is deleted, all belonging commands (endpoints), parameters and records from their relationship table are deleted. The idea of the "CPS Parameters" is to give the user the possibility to change the automatically derived parameter descriptions from the swagger JSON file for more convenience, like for example in cases where the description lists the parameter properties, but not how this parameter is actually used.

5.5.5 Config

The Config module is just a simple fetch query to the table "scenarioElementTypes", which as explained earlier is the only static table in the database. It selects all Scenario element types and gives them back as JSON object. This object is used on many locations for different functions for comparison and code visibility.

5.5.6 CPS interfaces

An existing OMiLAB project was used as a basis for the CPS interfaces - Smart City CPS Integration [35]. The "mBot" devices are using Apache Tomcat running on port 8080, started via crontab. They offer a web interface with endpoints and a Swagger OpenAPI, which was aligned as a common standard for all CPS in this thesis, as it fulfills the requirement to be able to describe its own API structure into a single machine-readable JSON file. The endpoints had a basic structure, according to the Smart City CPS Integration project needs.

One of the main goals of this master thesis is to extend the CPS interfaces and respectively capabilities on such level, that they offer as much control as possible to the end-user. To achieve that the API endpoints were extended with some extra commands and parameters, giving higher degree of freedom to the users. As an example, the "mBot moving forward" endpoint was extended with two extra GET parameters - speed and distance. Now the user can adjust these from the "smart" models and operate the devices according to his personal needs. The parameter's range restrictions are automatically read from the Swagger JSON file, inserted into the database and presented to the user, during "Model Commands - CPS Endpoints matching" process, as described in sub-section [5.3.3](#). Same approach was applied for all other API endpoints. Another milestone with the CPS interfaces was to enable the Cross-Origin Resource Sharing (CORS) support. As the ICT framework establishes cross-domain requests to the devices, special filters need to be applied, in order for the two-way communication channel to work properly. To achieve that, a CORS filter was set on every CPS interface.

6: Deployment and validation

use-cases

This chapter is split into two main parts - Deployment and Use Cases. The Deployment section will give an overview, how each part of this thesis is deployed. It will describe the physical and the technical requirements. The Use Cases section explains, what experiments will be executed into the OMiLAB environment, what results are expected and what kind of CPS will be used as prototypes.

6.1 Deployment

The deployment process is observed from end-user point of view. It consists of three main parts:

- CPS interfaces - how to configure the CPS interfaces and what are their technical requirements
- "smart" models - how to build the models, what are their features and how to export them
- ICT framework - how to import the "smart" models into the platform, how to extract the capabilities from the CPS, how to match the model properties with the CPS capabilities and how to execute the models

6.1.1 CPS interfaces

The first important step in the deployment process is how to prepare the CPS and their interfaces. The user should know the Web/IP address of the device and how to connect to it. In many cases the devices are available only by IP address, through SSH on particular port and only inside the inner OMiLAB network. The user can use PuTTY to connect to the devices. In order for the platform to be able to manage multiple devices, the

CPS need to offer at least some kind of HTTP interface. The HTTP interface should contain all available CPS capabilities as endpoints. It doesn't matter if they are defined as GET, POST or PUT requests. If it is necessary they should also send back responses, in the form of a JSON, Boolean or plain text. Another important condition is that the CPS should offer a single file, describing their API endpoint structure. The required format for this file is Swagger JSON. The file can be manually generated and placed in some public directory inside the CPS, available for external participants. The best option would be to install Swagger OpenAPI on the top of the CPS interfaces and define there the endpoints. The Swagger generates automatically the swagger.json file, which describes the whole structure. The Smart City CPS Integration OMILAB project offers a good example [35]. The second step is to apply a filter for the Cross-Origin Resource Sharing (CORS) on the CPS. This is very important, as the ICT framework is web-based application and by default the cross-domain requests are forbidden. After these two steps are done, the CPS are ready for execution. They should be placed on the desired position, depending how the "smart" models are defined. Last but not least, it should be checked, if they are turned on and if they have enough power.

6.1.2 "smart" models

The "smart" models are implemented as part of the s*IoT modelling framework in ADOxx. As already explained two types of models are needed, to perform an execution: one Environment and one Scenario. In order to build these models, first the s*IoT modelling framework library needs to be imported into the ADOxx platform. The both types of models already exist there and the user is able to create new model of each type. The models can contain different meta-elements, which were explained in details in Section 3.1.3.

To create a new Environment model, only two elements should be considered: CPS Starting Points and Notes. First the user should drag and drop one Note element and set a background image, containing his environment vision. It is highly recommended that the image has clearly defined starting positions of the CPS. It is desirable that the user has already an idea, what processes the CPS are going to execute and plan these steps. Every Environment model can contain as many CPS as needed. The user should put a CPS Starting Point for each CPS. The CPS Starting Point should be placed on the position, where the CPS actions start, no matter if the CPS is going to move or do some static tasks, until the end of the execution. In order to configure a CPS Starting

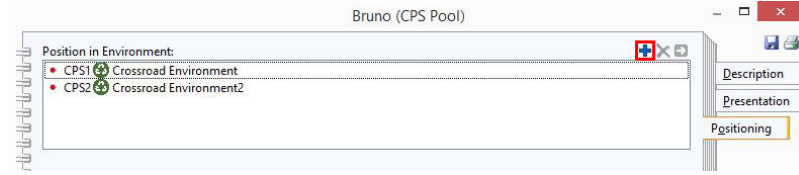


Figure 6.1: CPS Pool Positioning

Point, the user should first set the CPS IP/Web address, with the required port and select the CPS type from the list. These steps need to be applied on every CPS Starting Point. After this is done the model is ready for export.

To create a new Scenario model, many steps should be considered. The elements that are going to be used inside the model are: CPS Pool, CPS Start Event, CPS Task, CPS End Event, CPS Non-exclusive gateway and CPS Exclusive gateway. The user should place a CPS Pool for every CPS participating in the process and additionally one optional CPS Pool for the ICT framework. Each of the CPS Pools except the last one should be connected with the corresponding CPS Starting Point (representing the CPS) from the Environment model. The connection happens through the sub-section Positioning.

As shown on Figure 6.1 the "+" button on the top right corner is used to add or edit a connection. A CPS Pool can be connected to multiple CPS Starting Points from different Environment models (representing for example different devices), but the only important rule is: Every CPS Pool can be connected to exactly one CPS Starting Point from a single Environment model. If the user adds connections from one CPS Pool to multiple CPS Starting Points from the same Environment model, error will occur during the import of the model into the ICT framework.

After the CPS Pools are placed and connected properly, the user can define a desired process with the rest of the elements. As explained in subsection 3.1.2, the Scenario models define a sequence of actions, which can include user decisions or parallels. By definition every process has start and end, so every Scenario model must include only one CPS Start Event and one CPS End Event. As far as CPS Tasks, CPS Non-exclusive gateways (parallels) and CPS Exclusive gateways (decisions) are concerned every Scenario model can be infinitely complicated, including as many of these elements as needed. The general rule is that every time, when there is a need of parallel or decision, open and close element should be placed inside the model, describing where the parallel/decision process starts and ends. The CPS exclusive/non-exclusive gateways have inside-property

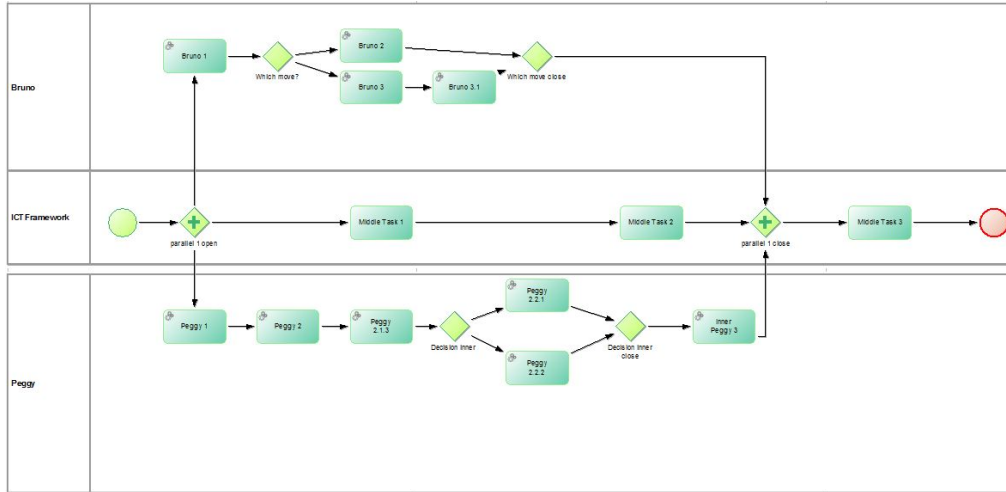


Figure 6.2: Scenario Model Example

	Subject	Predicate	Object
1	Bruno	hasCommand	moveForwardFollowLine
2	moveForwardFollowLine	hasParameter	motorSpeed
3	motorSpeed	hasValue	100

Figure 6.3: Scenario Element Command Triples

”Function” in sub-section Description. It should be set to ”Open” or ”Close”, depending if it is a starting or ending element. An example Scenario model with certain complexity is shown on Figure 6.2.

All CPS Tasks inside a CPS Pool will be performed during execution of the model on the connected CPS Starting Point or more precisely on the connected CPS, which this CPS Starting Point represents.

The CPS Tasks are the main components for the execution process. They contain some sub-sections, which define how exactly they are going to act. The Execution sub-section represents the command and the parameters, which are going to be executed from the CPS on this step. Commands, parameters and their values are described in the form of triples.

As explained in details in Section 3.1.3, the Task type sub-section represents what this particular CPS Task will be used for.

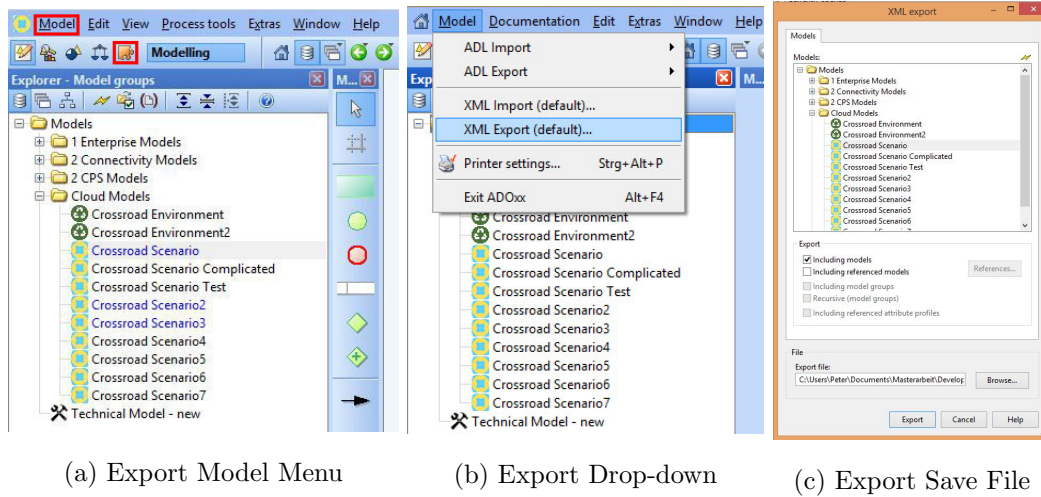


Figure 6.4: Export Model

After the design of the models is complete, they are now ready to be exported from the ADOxx Modelling Toolkit and uploaded into the ICT framework.

A model can be exported through the "Import/Export" button in the main menu (Figure 6.4a). After "Import/Export" is selected, under "Model" drop-down in main menu, there is an item "XML Export Default..." (Figure 6.4b). By clicking on this item, a pop-up will appear. First, the desired model for export should be selected from the sub-window "Models". Under the "File" sub-window, a file should be created or replaced in which the XML will be saved (Figure 6.4c). Clicking the "Export" button will finish the operation and the file will be ready for the next steps.

6.1.3 ICT framework

After the both types of models are correctly exported from the ADOxx platform, the user can now upload them into the ICT framework.

The ICT framework frontend contains a navigation panel with few items - Introduction, Models, Model Upload, CPS Types and CPS Parameters. The single role of each one of them will be explained in details. The "Introduction" item is the default starting page of the app. It contains a manual for the users, how to generate the models, export them, upload them into the ICT framework and how to execute them. The "Model Upload" button forwards the user to a page, where the models should be uploaded (Figure 6.5). As explained in subsection 5.3.2, the platform recognises automatically the type of the uploaded model. In order to transfer a particular Scenario model into the app, first all connected Environment models should be uploaded. If the user tries to upload a Scenario model, before the corresponding Environment models, an error will occur.

The screenshot shows a web application with a dark navigation bar at the top. The navigation bar contains links: "Introduction", "Models" (with a dropdown arrow), "Model Upload" (highlighted in blue), "CPS Types", and "CPS Parameters". On the right side of the navigation bar is a search bar labeled "Search Models". Below the navigation bar, the page title "Model Upload" is displayed. The main content area contains a form with the instruction "Please select Environment or Scenario model". Inside the form, there is a file selection interface with a button labeled "Choose File" and the text "No file chosen". At the bottom of the form is a blue button labeled "Submit".

Figure 6.5: Model Upload

The screenshot shows a web application with a dark navigation bar at the top. The navigation bar contains links: "Introduction", "Models" (with a dropdown arrow), "Model Upload" (highlighted in blue), "CPS Types", and "CPS Parameters". On the right side of the navigation bar is a search bar labeled "Search Models". Below the navigation bar, the page title "Enter path to commands" is displayed. The main content area contains two form sections. The first section is for "CPS Name: mBot" with Host: "http://192.168.75.25:8080". It has an input field with the placeholder text "Path to swagger JSON Endpoints file after host, example: /mBot/api/swagger.json" and a blue button labeled "Get Commands". The second section is for "CPS Name: RoboticArm" with Host: "http://austria.omilab.org". It has an input field with the text "/omirob/dobot1/rest/static/swagger.json" and a blue button labeled "Ready!". To the right of the "Ready!" button is a green circular icon with a white checkmark.

Figure 6.6: CPS Path to Swagger JSON

After an Environment model is uploaded, if its CPS Starting Points contain new CPS Types, still not existing in the platform, the user will be redirected to a new page, where an input field will appear for each new CPS Type (Figure 6.6). The input fields contain short explanation and example how the path to the swagger JSON file should be entered by the user. The idea on this step is to set the path to the file after the IP/Web address, as the this property is automatically read from the model. After clicking the "Get Commands" button, a status will appear. If the commands are read successfully, the button will be disabled, its text will change to "Ready!" and success icon will appear on the right side (Figure 6.6).

If the entered path is wrong or if there is any type of connection problem with the device, the button will be clickable again, its text will change to "Failed! Try again" and a red X icon will be shown on the right side (Figure 6.7). The user can check the connection to the device or correct the path and click the button again. After this step is correctly executed, the Environment model and all its contained CPS types will be available on the platform.

Before uploading the Scenario model, the user should make sure that all connected

Enter path to commands

CPS Name: **mBot**
Host: **http://192.168.75.25:8080**

Failed! Try again



Figure 6.7: CPS Path to Swagger JSON Failed

Introduction Models Model Upload CPS Types CPS Parameters					Search Models
Commands Matching					
Element Name	Pool Name	Commands inside this element	Available CPS Endpoints	Parameters	
Bruno start moving forward	Bruno	Bruno hasCommand moveForwardFollowLine moveForwardFollowLine hasParameter motorSpeed motorSpeed hasValue 100	mBot		
Bruno jump over the gap	Bruno	Bruno hasCommand moveForward moveForward hasParameter motorSpeed	mBot		

Figure 6.8: CPS Commands Matching

Environment models are transferred to the platform in the same way.

After a Scenario Model is uploaded, the user will be redirected to a new page, where he should match the given model commands (triples) from each element to the actual CPS endpoints. A select menu will appear for each element with command and for each unique connected "CPS Type" (Figure 6.8). If a Scenario model is connected to multiple Environment models and for example one CPS Pool to different CPS types, multiple select menus will appear for each single command. A corresponding CPS Endpoints should be selected. If the Endpoint also contains parameters, they will automatically appear in the last right column after the endpoint is selected (Figure 6.9).

After all endpoints are selected and the corresponding parameter values are entered, the save button should be clicked. If everything is done correctly, the models will be ready for execution. The connection between the models is many to many, but during the execution process always one Environment Model and one Scenario Model are selected.

To see the current available models, the navigation drop-down item "Models" should be selected. It contains two options - "Environments" and "Scenarios", respectively for each type of model. The user can select each one of them and will be redirected to a page, where all the models of the selected type are shown. Every model contains its

Position on the load	RobotArm	RoboticArm hasCommand Position X Y Z RoboticArm hasParameter X RoboticArm hasParameter Y RoboticArm hasParameter Z X hasValue 300 Y hasValue -130 Z hasValue 90	RoboticArm 300 X coordinates -130 Z coordinates 90 Y coordinates
Pick up the load	RobotArm	RoboticArm hasCommand grabOn RoboticArm hasParameterValue true	RoboticArm true Grab or Release the item
Position over the load.	RobotArm	RoboticArm hasCommand Position X Y Z RoboticArm hasParameter X	/reset /positionXYZ /grabOn

Figure 6.9: CPS Commands Matching with Parameters

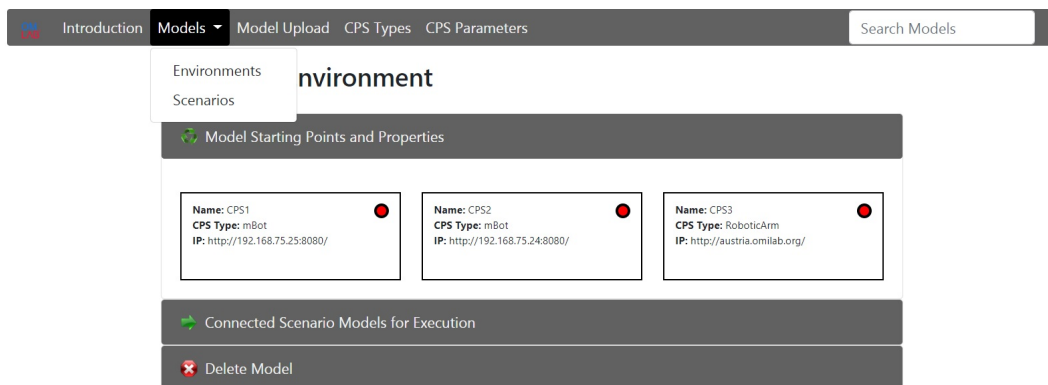


Figure 6.10: Environment Model

name and few properties, depending on the model type. In the last column, there is a button "Details", which redirects the user to the selected model. Each model contains three collapsible sections. The first one contains its properties, the same way as they are described inside the actual model in ADOxx. The second section describes all the available connected models from the opposite type, which are available for execution and the last section serves for deletion of the model (Figure 6.10).

If the "Delete Model" section is selected and the "Delete Model" button is clicked, a confirmation window appears on the screen. The user has to confirm the deletion of the model. The models must be deleted in the same order, as uploaded. This means, that in order to delete a specific Environment model, first all connected Scenario models should be removed from the platform.

#	Name	URL Path	Commands	Delete
1	mBot	/mBot/api	Commands	Delete
2	RoboticArm	/omirob/dobot1/rest	Commands	Delete

Figure 6.11: CPS Types

#	Command	Method
1	/getPorts	get
2	/turnRightFollowLine/{motorSpeed}	get
	/turnLeft/{sleepTime}/{motorSpeed}	get
	/turnLeftFollowLine/{motorSpeed}	get
	/moveForward/{sleepTime}/{motorSpeed}	get

Figure 6.12: CPS Types Commands

If a model is selected from the second section "Connected Models for Execution", the execution process starts. Depending on the CPS participating in the execution process, there are two possible scenarios - the user will be asked to enter Authentication Token for some specific device that requires it or the execution process will start directly. Each step from the Scenario model will be presented on the screen. If the step contains commands, the CPS will perform some action and the ICT framework will go to the next step. If there is some step, which requires user decision, all possible options will be displayed. Each element option will look similar as into the original Scenario model. The user can click on the preferred option and the execution process will continue on the selected path. After the execution process is done, a message "End of Execution" will be displayed.

There is an extra menu item "CPS Types" into the main navigation (Figure 6.11). From there the user is able to see all available CPS, their endpoints and corresponding request methods (Figure 6.12) currently existing on the platform. The user can delete existing CPS Types, imported during the Environment model upload process. This could be necessary for example when a particular CPS has updated endpoints (updated swagger JSON file) and the user needs to load them over again or also in cases where during the Environment model upload, the swagger JSON file hasn't been read correctly.

	Introduction	Models ▾	Model Upload	CPS Types	CPS Parameters	Search Models
---	--------------	----------	--------------	-----------	----------------	---------------

CPS Command Parameters			
#	Name	Description	Details
1	x	X coordinates	Edit
2	z	Z coordinates	Edit
3	grabOn	Grab or Release the item	Edit
4	y	Y coordinates	Edit

Figure 6.13: CPS Parameters

In order to delete particular CPS Type, first all Environment models containing this CPS Type should be deleted.

The last navigation item is "CPS Parameters" (Figure 6.13). These CPS parameters are displayed to the user, during the process - matching of commands to endpoints, by Scenario model upload. All existing endpoint parameters have a description coming from the swagger JSON file. This description can be edited for more comfort. Usually the description of a parameter states, what this parameter is used for. In our case, inside the ICT framework it would be more meaningful, that the parameter describes, which value or what range of values should be entered.

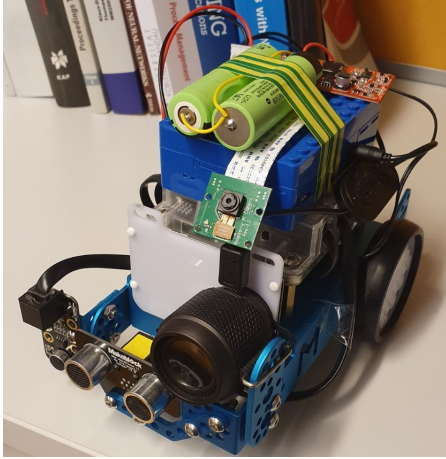
The last important item in the main navigation is the search bar for models. It is placed on the top right corner of the platform. The user is able to search for all types of models. He needs to write few letters from the model name and the relevant results will be shown beneath the search bar. The user can click on the desired model and he will be directly redirected to it.

6.2 Use-cases

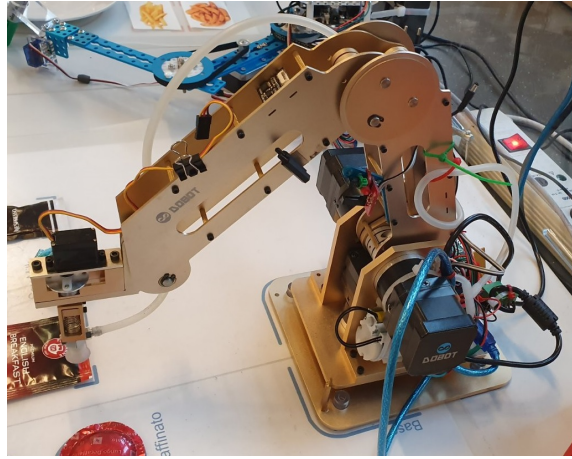
The experiments are executed in the OMiLAB environment. In order to test the capabilities and the performance of this thesis, four experiments are conducted, each with increasing complexity. The experiments are observed from a macro-level, which means that only the input "smart" models, their integration into the platform and their actual physical execution from an user point of view are considered.

The four experiments use one common Environment model, describing the surrounding area, how many and what kind of CPS are used and their starting points. As a prototype for the experiments two types of CPS are used: Makeblock mBot and RoboticArm, which are shown on Figure 6.14.

The Environment model on Figure 6.15 used for the validation contains three CPS



(a) mBot



(b) RoboticArm

Figure 6.14: Prototype CPS devices

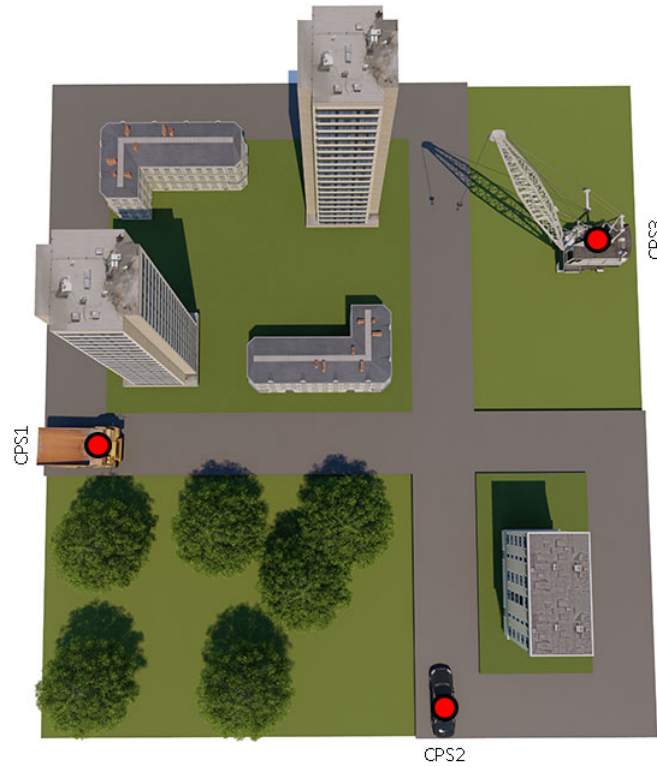


Figure 6.15: Use-Case Environment Model

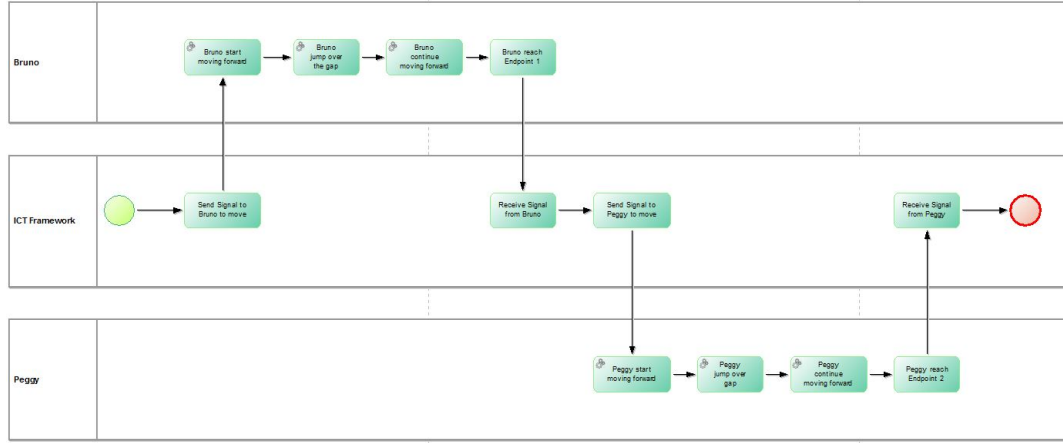


Figure 6.16: Scenario Model 1

Starting Points, representing accordingly two mBots and one RoboticArm. The both mBots are placed on "CPS1" and "CPS2". The RoboticArm is placed on "CPS3". In order to test the ability of the ICT framework to handle many-to-many Scenario-Environment models relationship, a second Environment model, copy of the first one, with switched mBot positions is integrated into the platform.

The CPS interfaces of the devices were configured and their swagger JSON files were prepared. For the validation process, the ICT framework starts with empty database and the relevant CPS types and their properties are read from the devices during the experiment. Both Environment models are uploaded into the platform.

6.2.1 Use-case 1

For the first Use-Case a simple Scenario model is used. Its process is straightforward, without any decisions or parallel executions. It contains the two mBots. Every mBot has to execute three separate steps, which will be performed sequentially. The first mBot moves forward, until it reaches the end of the black line, then it jumps over the gap and continues moving on the next black line until it reaches its end. Then the second mBot executes the exact same process, split on three steps. The model is shown on Figure 6.16.

In order to prepare the model into the platform for execution, the CPS Task commands (triples) are matched with the corresponding mBot endpoints.

The corresponding physical model and the mBots starting points are shown on figure 6.17. Figure 6.18 shows a frame during execution right before the first mBot has reached his endpoint.

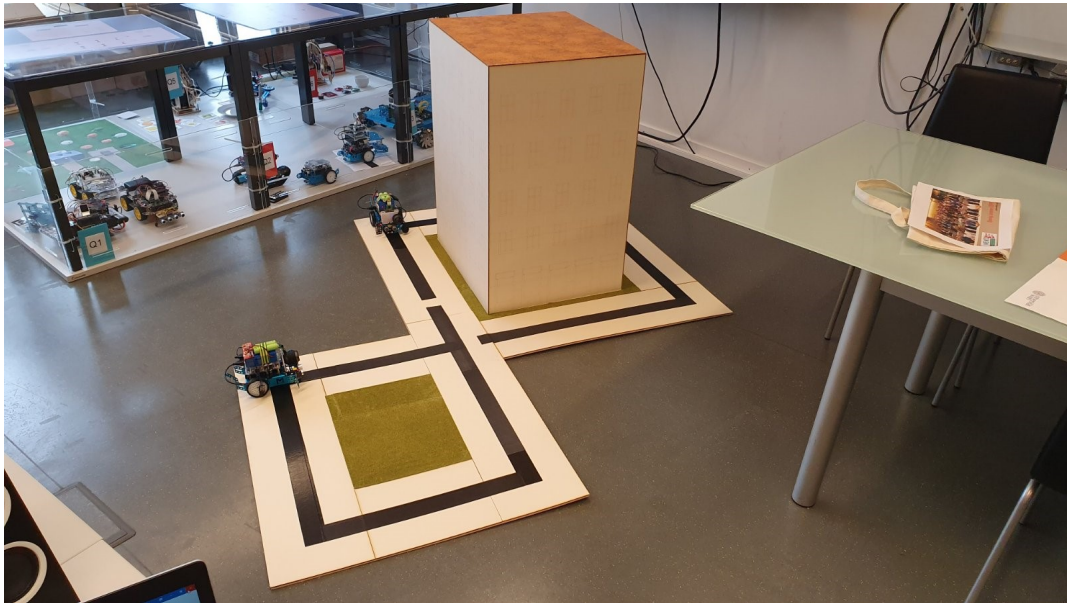


Figure 6.17: mBots Starting Points



Figure 6.18: mBots Execution

Please choose the next element:

Send Signal to
Bruno to move

Send Signal to
Peggy to move

End CPS Exclusive Gateway: "Who goes first?"
Start CPS Task: "Send Signal to Peggy to move"
End CPS Task: "Send Signal to Peggy to move"
Start CPS Task with Command: "Peggy start moving forward"
End CPS Task with Command: "Peggy start moving forward"
Start CPS Task with Command: "Peggy jump over gap"
End CPS Task with Command: "Peggy jump over gap"
Start CPS Task with Command: "Peggy continue moving forward"
End CPS Task with Command: "Peggy continue moving forward"
Start CPS Task: "Peggy reach Endpoint 2"
End CPS Task: "Peggy reach Endpoint 2"
Start CPS Task: "Receive Signal from Peggy"
End CPS Task: "Receive Signal from Peggy"
Start CPS Task: "Send Signal to Bruno to move."
End CPS Task: "Send Signal to Bruno to move."
Start CPS Task with Command: "Bruno start moving forward."
End CPS Task with Command: "Bruno start moving forward."
Start CPS Task with Command: "Bruno jump over gap."
End CPS Task with Command: "Bruno jump over gap."
Start CPS Task with Command: "Bruno continue moving forward."
End CPS Task with Command: "Bruno continue moving forward."
Start CPS Task: "Bruno reach Endpoint 1."

Figure 6.19: Scenario Model 2 Decision Output

6.2.2 Use-case 2

The second Use-Case includes a decision element, giving the user the possibility to decide, which mBot goes first. From scenario point of view, this model executes exactly the same process as the first Use-Case. The both mBots perform the same workflow as described in Use-case 1 sequentially, split on three separate steps until they reach the end of the black line. When the execution process starts, the user is asked to give his input into the platform, which determines the execution order. The decision process and the actual execution of the model from user point of view are illustrated on figure 6.19. The model is displayed on Figure 6.20.

Again each CPS Task command is matched correctly with the corresponding mBot endpoint and the correct parameter values are entered.

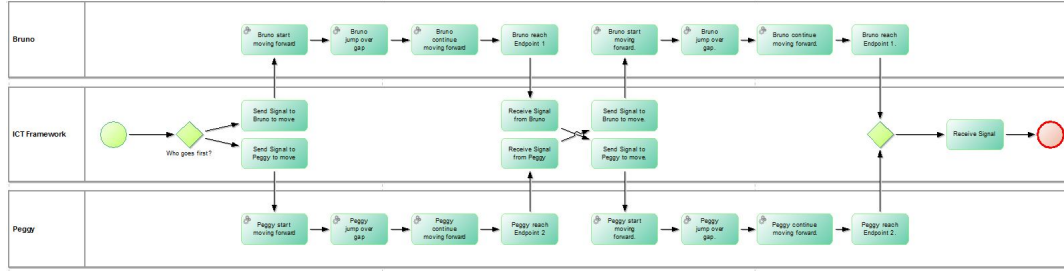


Figure 6.20: Scenario Model 2

6.2.3 Use-case 3

The third Use-Case introduces parallel execution and collision detection. It executes the same process as the previous use-cases, but the both mBots move in parallel. In order to avoid collision, on some point one of the mBots detects (through his distance sensor), that there is an object in close distance in front of him and waits. The second mBot moves further and cleans the route. Then the first one is good to go and continues with his process. The actual execution process of the model and the detected collision are displayed on figure 6.21. From the executed process, it is visible that the CPS Task "Bruno jump over gap using distance sensor" waits until the CPS Task "Peggy jump over gap using distance sensor" is successfully finished. A frame during the collision detection is shown on figure 6.22. The Scenario model 3 is displayed on Figure 6.23.

The CPS Task commands are matched with the corresponding mBot endpoints during Scenario model upload, but the distance sensor functionality is automatically integrated, because of the CPS Task type, as explained earlier.

6.2.4 Use-case 4

The last Use-Case is the most complex one. It includes the RoboticArm and a full-scale process. The both mBots and the RoboticArm execute actions in parallel. The both mBots move forward, reach the end of the black line, the RoboticArm loads the first mBot with luggage, collision detection is performed, the first mBot waits until the second mBot jumps over the gap and when the sensor of the first mBot doesn't detect any obstacles in the front, it also jumps over the gap. Both mBots move on the black line, until they reach its end. This last Use-Case shows an example of complexity with multiple devices that can be reached and how the Scenario model process can be extended. The RoboticArm requires authentication with token, which is realized directly inside the platform as displayed on figure 6.24. A frame, during model execution is shown

Start CPS Start Event: "Start Scenario"
 End CPS Start Event: "Start Scenario"
 Parallel execution started
 Start CPS Task: "Send Signal to Bruno to move"
 End CPS Task: "Send Signal to Bruno to move"
 Start CPS Task: "Send Signal to Peggy to move"
 End CPS Task: "Send Signal to Peggy to move"
 Next step in parallel execution
 Start CPS Task with Command: "Bruno start moving forward using distance Sensor"
 Start CPS Task with Command: "Peggy start moving forward using distance Sensor"
 End CPS Task with Command: "Peggy start moving forward using distance Sensor"
 End CPS Task with Command: "Bruno start moving forward using distance Sensor"
 Next step in parallel execution
 Start CPS Task with Command: "Bruno jump over gap using distance sensor"
 Start CPS Task with Command: "Peggy jump over gap using distance sensor"
 End CPS Task with Command: "Peggy jump over gap using distance sensor"
 Collision detected! Wait to execute again CPS Task with Command: "Bruno jump over gap using distance sensor"
 End CPS Task with Command: "Bruno jump over gap using distance sensor"
 Next step in parallel execution
 Start CPS Task with Command: "Bruno continue moving forward using distance Sensor"
 Start CPS Task with Command: "Peggy continue moving forward using distance Sensor"
 End CPS Task with Command: "Peggy continue moving forward using distance Sensor"
 End CPS Task with Command: "Bruno continue moving forward using distance Sensor"

Figure 6.21: Scenario Model 3 Collision Detection Output

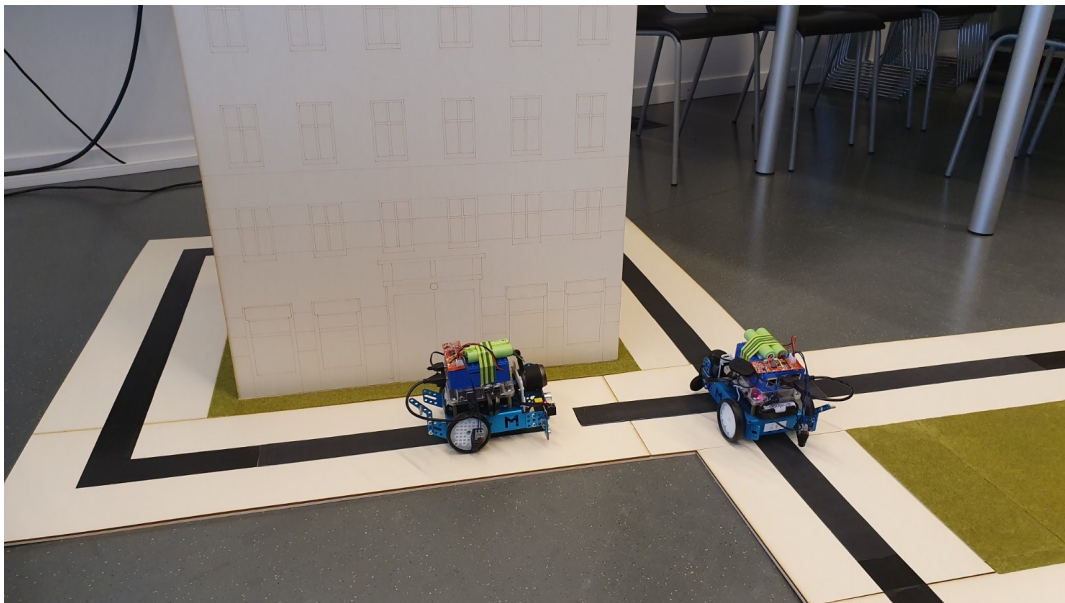


Figure 6.22: Scenario Model 3 Collision Detection

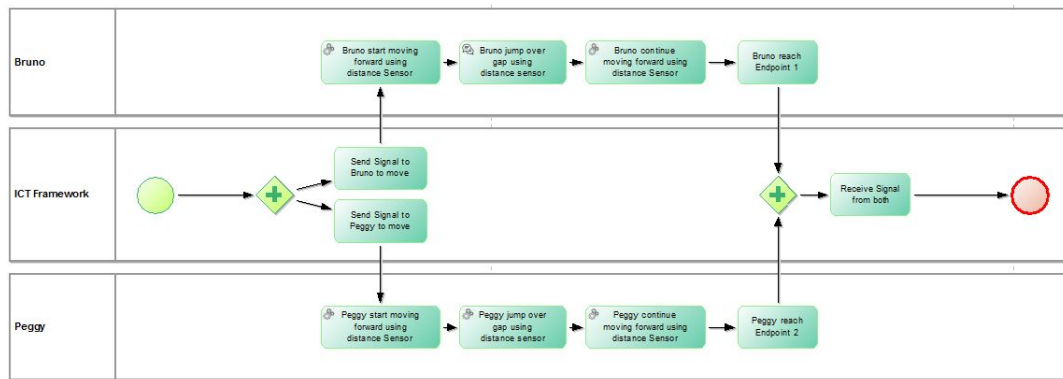


Figure 6.23: Scenario Model 3

Please enter authentication token for the RobotArm:

Enter Auth

Start CPS Start Event: "Start Scenario"

End CPS Start Event: "Start Scenario"

Parallel execution started

Start CPS Task with Command: "Bruno start moving forward using distance Sensor"

Start CPS Task with Command: "Peggy start moving forward using distance Sensor"

Figure 6.24: Scenario Model 4 Authentication

on figure 6.25. The Scenario model 4 is displayed on Figure 6.26.

All of the CPS Task commands, except the distance sensor functionality are matched with the corresponding mBot and RoboticArm endpoints. The correct parameters are also entered inside the ICT Framework, during model upload.

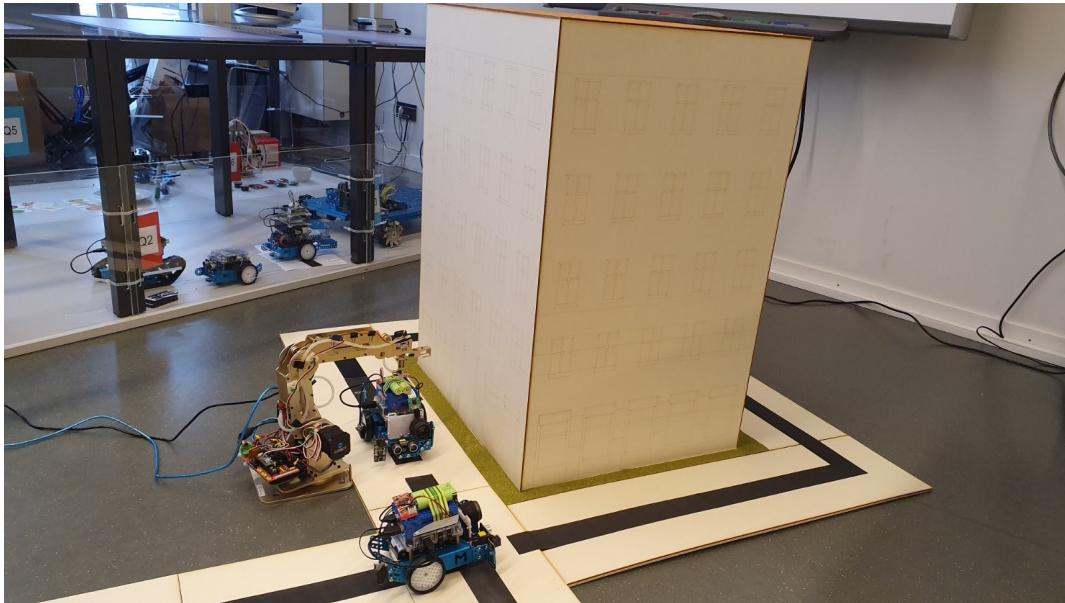


Figure 6.25: Scenario Model 4 frame

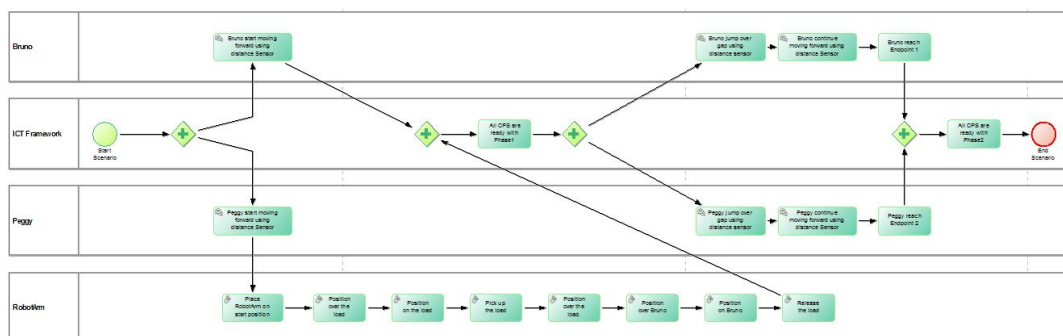


Figure 6.26: Scenario Model 4

7: Conclusion

This final chapter gives a summary what this master thesis covers and tried to achieve. Furthermore, it provides answers to the relevant research questions and overview of the resulting solutions. Additionally, it offers a SWOT analysis, showing clearly what are the strong and weak aspects of this thesis, which opportunities and threats it might encounter. The last section Future Work will cover some of the possibilities to extend the ICT framework, the CPS devices and the conceptual models.

7.1 Summary

The main goal was to provide an answer and realisation of the following questions:

- 1. What kind of architecture is needed, in order to build an automated link between models invented through design thinking approach and CPS run-time environment.
- 2. What components should such architecture contain.
- 3. How to build each of these components as a standalone product and how to link them together.

The thesis provides a solution, which covers these research questions. The resulting model-based framework consists of three main components:

- Design Modelling Environment
- ICT Framework
- Run-time Environment

All these components are connected together as a standalone end-to-end product and realise the link between models invented through design thinking approach and CPS run-time environment. Each component of the product has a specific individual role, but

also contributes to the overall aim of the project. The design modelling environment consists of meta-model and two types of "smart" models. They are composed from meta-artefacts, which on one side provide flexibility and boost the design thinking approach, but on other side contain properties, connectable with the actual CPS run-time capabilities. The run-time environment consists of CPS and their interfaces, which were adjusted to a common standard, extending the standard REST API approach, allowing the CPS to describe their own endpoint structure, so that an external application is able to read their basic properties and capabilities. The main focus of this thesis was on the ICT framework, providing connection between the other two components and offering flexibility and functionality to the users. The ICT framework consists of many different modules with diverse features. The ICT framework is built with JavaScript on both the frontend and the backend side. It consists of a database, where the "smart" models, their properties and the related attributes are stored. It offers a models upload module, responsible for the upload and the transformation of the "smart" models to a JSON structure. The upload module also inserts the models into the database. Moreover, it provides a read CPS capabilities module, which takes care for the extracting and the saving of the CPS capabilities from each device into the database. Furthermore, it supports different protocols and mechanisms to provide two-way communication channel between the platform and the devices. The ICT framework realises the connection between the "smart" model abstract element commands and the actual CPS capabilities. It offers a matching virtual capabilities interface, which supports the user in the matching process and stores the selected commands into the database. In addition, the platform generates virtual copies of the uploaded models, allowing them to keep similar to the initial structure and virtual representation for more user convenience. The ICT framework manages to read and execute the "smart" models, providing different features and decision possibilities during execution for the users. It offers a collision detection mechanism concept, allowing the devices to collaborate with each other. This concept can be easily extended for all kind of CPS and sensors. Last but not least, the ICT framework provides a frontend, which is responsive, reliable and fast, offering many interesting frontend features, like model search module, user navigation panel and many dynamic elements, which give the user more satisfaction by browsing. The ICT framework is compatible with any kind of device or platform.

This paper represents a step towards the digital transformation of the enterprise models. It is fundamental and can be used in many different areas like smart manu-

facturing processes, household automation, autonomous driving. It offers an innovative approach for building of procedures with "smart" devices of every purpose. As a result, it provides an automated and effective architecture, which can be extended and adjusted according to the specific user needs.

During the process of research and implementation it was found, that there exist more than one option how to build such architecture. The solution can go in different directions - one possibility would be to extend the "smart" conceptual models and the CPS run-time environment, decommissioning the gap between them, allowing to build similar architecture with less software dependence. Another possibility would be to build such software tool, which covers the whole process of building the meta-model, the "smart" models and connecting them with CPS run-time environment.

7.2 SWOT analysis

The SWOT analysis stays for Strengths, Weaknesses, Opportunities, and Threats. It gives a summary about the different aspects of the introduced architecture. It tackles the advantages and the disadvantages used by the current approach. The SWOT analysis showed on Figure 7.1 can also be used to consider, which processes can be improved and how to extend the architecture in the future. These possibilities will be described in much more details in the next section "Future work".

7.3 Future work

The future work on this solution is concerned with expanding all available aspects towards higher level of interactivity, more convenience for the users and better management of the CPS. There are three main pillars, which will be considered:

- Conceptual models and elements
- ICT framework
- CPS devices

7.3.1 Conceptual models and elements

The both types of "smart" models can be extended with new interactive meta-elements, providing more functionality and possibilities.

Strengths	Weaknesses
- The design modelling environment consists of two types of models, with different roles, The Environments describe the participating CPS, the surrounding area and the conditions, the Scenarios describe particular workflow of actions, executed by one or more CPS - The framework allows N to N connection between the both types of models, which means that in one Environment can be executed unlimited many different Scenarios and one Scenario can be executed in many different Environments - The two types of models together fulfill the business needs and the CPS run-time requirements - The Scenarios offer parallel execution of activities, from different CPS, allowing them to collaborate with each other through sensors - The Scenarios offer real-time decisions by the user during the execution process - The ICT framework offers automatic model type recognition by upload - The ICT framework offers database to store models, CPS types and capabilities and the relevant connections between them - The ICT framework offers semi-automatic CPS capabilities read mechanism - The ICT framework offers responsive, reliable and fast frontend - The ICT framework offers models search, overview and execution features - The run-time environment offers unified connection common standard for CPS interfaces, allowing external participants to make request and get responses - The run-time environment allows the CPS to describe their own API structure in a single file	- The Environments offer only two elements - Environment picture and CPS Starting Point for each CPS, describing its properties - The Scenarios offer only basic tasks, parallels and decisions - The models can be executed only from the ICT framework, not from the ADOxx itself, where the models were created - The collaboration between CPS in the ICT framework is based only on sensor data - The ICT framework doesn't offer genuine HTTP/REST interface, for all possible CPS devices (with different authentication methods) and capabilities - The ICT framework can't communicate directly with ADOxx - The ICT framework doesn't match automatically the model properties with the CPS capabilities - The CPS interfaces support only simple HTTP requests, it is not distinguishable if some request is a command with action or sensor data - The CPS and their interfaces offer only basic actions
Opportunities	Threats
- The both model types can be extended with new meta-elements - The Environments can have new meta-element, representing static objects - The Environment CPS Starting Points can be extended with new properties - The Environment picture (describing the surrounding area) can be transformed into 3D object or replaced by online-stream - The Scenarios can be extended with sub-processes, timers, new gateways and "AdoScript", allowing them to communicate with the ICT framework - The ICT framework can offer functionality, allowing it to communicate with ADOxx - The ICT framework can offer many different collaboration mechanisms for the CPS - The ICT framework HTTP requests towards CPS can be extended to an actual interface, for all possible CPS devices (with different authentication methods) and capabilities - The ICT framework can match automatically the model properties with the CPS capabilities - The CPS interfaces can be extended with further logic for commands - sensors differentiation	- As the CPS devices are so different from each other in design and capabilities, there could be a case, where the ICT framework needs to be generally extended only for one particular type of CPS - As the solution is service oriented architecture, it is possible, that another architecture directly integrated into a modelling tool, with the same functionality could be more successful on some point - There could be a new type of model, which fulfills generally better the business needs and the CPS run-time environment requirements

Figure 7.1: SWOT Analysis

A new meta-element, representing static objects can be introduced to the Environment models. An example in the Use-Case model could be a traffic light element, which gives indication to the mBots, when they should move and when they should stop and wait. This static meta-element can acquire many different forms, depending on the current requirements in a particular model. If an Environment model contains two CPS of type NaoBot communicating together, the static meta-element could be a docking station where they charge or a chair, where they sit. In another Environment model the static meta-element can also be a barrier, which opens and closes, depending on different circumstances. This element can be later connected with the CPS sensors. Such element would give to the model much more interactiveness and interesting capabilities. Another future possibility is also to extend the current existing elements. The CPS Starting Points can contain more different CPS types and the Note element can be transformed to 3D structure, representing the Environment model in 3D format, much more similar to the actual physical model, where the different experiments are executed. There is also a possibility to replace the Note element with online stream. The Environment model could represent real-time video of the used physical model. Another benefit of this approach is that the actual execution will be traceable inside the ADOxx modelling tool and every action executed by the CPS will be visible inside the tool.

The future development of the Scenario models can go in a similar direction. As the Scenario models mostly represent a process, they can be extended with many different elements. If a standard BPMN model is considered, it contains much more meta-elements with various capabilities. One good example is the "Sub-Process" element. It contains another process inside a single element, which can be used in many cases to simplify the model structure from a user point of view. Another great example is the "Timer" element. It would tell to a CPS, that during the execution some actions should be performed after a specific time period. It is also a good option to include new types of gateways, like "Event-based Gateway". This specific gateway determines the further model execution, based on some previous event, like for example CPS action or sensor output. Extending the Scenario models with such meta-elements, will give much more freedom to the designer to implement a process that would fulfill his vision and ideas.

Another possibility to extend the "smart" models is to add "AdoScript" code inside them. It should be possible for the user to track the whole execution process inside a specific Scenario model. The ICT framework could also be extended with functionality towards direction requests and responses to ADOxx modelling tool. With "AdoScript"

the Environment models could also request the corresponding endpoints from the CPS on design time. That would allow the designer to see all available commands during the modelling process and enter more accurate triples into the Scenarios.

7.3.2 ICT framework

The ICT framework is the main component of this thesis and as such it offers a variety of opportunities for further development. One direction could be the collision detection mechanisms. They can be extended with further logic, if the CPS interfaces provide their sensor data as a single output. The execution of the model won't be concerned with separate requests and responses from the sensor endpoints. The data can be part of the standard execution of each element and specific cases can be applied if some sensor output goes beneath or above any threshold. In other words the sensor mechanisms can be developed as a dynamic component. The whole process should be optimized right from the start. During the phase of reading the commands from the swagger JSON file, there must be a clear definition how to separate the sensor data from the commands endpoints. The swagger JSON structure logic should have a clear differentiation about these two parts.

Another important direction is to extend the HTTP requests to an actual interface. As the main idea of this thesis is to be used for all kind of CPS, with much different capabilities, designs and run-time environment, it is important to build an interface with various possibilities. In order to consider all options, an interface should include the GET, POST, PUT and even probably DELETE request methods. The requests to the devices and the coming responses can have different forms - Boolean, JSON, XML, plain text and so on. Another aspect is that different CPS may require different authentication methods. Some of them can even offer an interface with username and password. Further work is also necessary in this direction. The authentication method of the devices should be readable from the swagger JSON file and different approaches, corresponding to the CPS needs should be considered.

As previously mentioned, the ICT framework can be also extended to provide requests and responses towards the ADOxx modelling tool. The execution interface can be transformed to actually show the changes and the user oriented content in the ADOxx tool itself. This way the user will have the comfort to track the whole process on the same place, where he designs his model and to make immediate changes if necessary.

Last but not least, there is a great potential for improvement in the Scenario model

commands - CPS endpoints matching process. With further logic, this process may not be required at all and actually the Scenario model commands can be automatically matched to the right endpoints, depending on particular patterns. The Scenario model commands can be categorized and whenever a new similar command enters, it should be placed to an existing category or a new category will be created.

7.3.3 CPS devices

Currently the CPS interfaces support simple HTTP requests. A good idea could be to split their interfaces in direction sensor-based requests and actual commands. If the CPS interfaces provide the data from all available sensors attached to them as a single output, the ICT framework can process this data and use it for calculations, validation and further actions. Another great opportunity is to extend the CPS devices with more complicated actions. The RoboticArm could have image recognition ability, which would allow it to find objects, without the need of specific coordinates. The input could be for example an object description.

7.4 Final comments

This master thesis provides a product, in the form of an end-to-end architecture, which fulfills the requirements to generate automated link between models invented through design thinking approach and CPS run-time environment. During the process of research and implementation, I came to a conclusion, that there are many different ways, how to build such architecture. On one side the models play in general an important role, supporting industry and people to express their abstract vision and ideas. The models offer a generic approach, understandable by business and IT. The models can be composed in a way that makes them machine readable, which is crucial for the IT. On the other side the CPS are smart devices, which need machine readable instructions, in order to be managed and to be able to execute some actions. The CPS should offer some kind of interface, allowing external parties to connect to them and to execute these instructions. These are the two mandatory components, which most probably can't be replaced in the whole architecture. For all of the other components, there are plenty of possibilities. The models can have a diverse structure, with various artefacts and functionalities. There isn't one perfect straight-forward way how to build them or how they should look like. Exactly this flexibility makes the modelling process so interesting and useful. The connection gap between the models and the CPS can also be realised

in many different ways. There isn't one perfect way how to accomplish this process, but there are some requirements, which should be taken into consideration: performance, availability, cost, further development possibilities and hardware/software requirements. All these requirements shape a product, his life-cycle and considering them, brings a significant business value. The general idea of this master thesis was very interesting for me and I am really grateful that I had the opportunity to contribute in such innovative and developing area.

Acronyms

CPS	Cyber-Physical Systems
PC	Personal Computer
ICT	Information and Communications Technology
UML	Unified Modeling Language
MDA	Model-Driven Architecture
OMG	Object Management Group
MDE	Model-Driven Engineering
MDD	Model-Driven Development
MDSD	Model-Driven Software Development
DSL	Domain-Specific Language
MBT	Model-Based Testing
XML	Extensible Markup Language
HTML	HyperText Markup Language
s*IoT	Semantic Internet of Things
IoT	Internet of Things
IP	Internet Protocol
BPMN	Business Process Model and Notation
API	Application Programming Interface
HTTP	HyperText Transfer Protocol

HTTPS	HyperText Transfer Protocol Secure
SQL	Structured Query Language
REST	Representational State Transfer
AJAX	Asynchronous JavaScript and XML
JSON	JavaScript Object Notation
CSS	Cascading Style Sheets
W3C	World Wide Web Consortium
DOM	Document Object Model
NPM	Node Package Manager
I/O	Input/Output
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
JS	JavaScript
ES6	ECMAScript 6
MVC	Model–View–Controller
app	Application
rem	root ephemeral unit
CORS	Cross-Origin Resource Sharing

List of Figures

2.1	Three Layer Framework	3
3.1	Model-based Framework	7
3.2	Meta-model	9
3.3	CPS Starting Point	10
3.4	CPS Pool Positioning Connections	11
3.5	Non-exclusive Gateway (Parallel)	13
3.6	Architecture Overview	15
3.7	Run-time Environment	16
4.1	NodeJS processing model [21]	21
4.2	Handlebars Block Expression	24
4.3	Handlebars Helper	25
5.1	Software Architecture Overview	31
5.2	SQLite Database	32
5.3	Express Routing	33
5.4	Handlebars Block Expression	34
5.5	Bootstrap Grid	36
5.6	Upload Model	37
5.7	Insert Environment Model	39
5.8	Request CPS Endpoints	40
5.9	Insert CPS Types	41
5.10	Insert Scenario Model	42
5.11	Render Parameters for Endpoint	44
5.12	Fill Parameters-div	45
5.13	Environment Models	45
5.14	Fetch all Environment Models	46

5.15 Sort Scenario Model Elements	47
5.16 Recursive Scenario Model Elements Sequence	48
5.17 Recursive Scenario Nodes Sequence	51
5.18 NodeGroup1	52
5.19 NodeGroup2	52
5.20 SimpleNode part 1	54
5.21 SimpleNode part 2	55
5.22 SimpleNode part 3	55
6.1 CPS Pool Positioning	63
6.2 Scenario Model Example	64
6.3 Scenario Element Command Triples	64
6.4 Export Model	65
6.5 Model Upload	66
6.6 CPS Path to Swagger JSON	66
6.7 CPS Path to Swagger JSON Failed	67
6.8 CPS Commands Matching	67
6.9 CPS Commands Matching with Parameters	68
6.10 Environment Model	68
6.11 CPS Types	69
6.12 CPS Types Commands	69
6.13 CPS Parameters	70
6.14 Prototype CPS devices	71
6.15 Use-Case Environment Model	71
6.16 Scenario Model 1	72
6.17 mBots Starting Points	73
6.18 mBots Execution	73
6.19 Scenario Model 2 Decision Output	74
6.20 Scenario Model 2	75
6.21 Scenario Model 3 Collision Detection Output	76
6.22 Scenario Model 3 Collision Detection	76
6.23 Scenario Model 3	77
6.24 Scenario Model 4 Authentication	77
6.25 Scenario Model 4 frame	78
6.26 Scenario Model 4	78

7.1 SWOT Analysis	82
-----------------------------	----

Bibliography

- [1] National Science Foundation, “Cyber-physical systems: Enabling a Smart and Connected World”, https://www.nsf.gov/news/special_reports/cyber-physical/.
- [2] Dimitris Karagiannis; Vimal Kunnunmel, “A Conceptual Modeling Framework for Evaluation of Cyber-Physical Systems based on Applied Category Theory and Metamodeling”, <http://ceur-ws.org/Vol-2234/paper6.pdf>.
- [3] National Science Foundation, “Cyber physical systems: Connecting computation and the physical world”, <https://www.nsf.gov/cise/news/cps-perspective.jsp>.
- [4] OMiLAB, “OMiLAB-Rob Experiments”, <http://austria.omilab.org/psm/omirob>.
- [5] László Monostori, “Cyber-physical Production Systems: Roots, Expectations and R&D Challenges”, <https://www.sciencedirect.com/science/article/pii/S2212827114003497>.
- [6] Dimitris Karagiannis; Michael Walch, “Service-Driven Enrichment for KbR in the OMiLAB Environment”, in *Serviceology for Services: 5th International Conference, ICServ*, 2017.
- [7] Dimitris Karagiannis; Michael Walch, “How to connect design thinking and cyber-physical systems: the s*IoT conceptual modelling approach”, in *52nd Hawaii International Conference on System Sciences, HICSS*, 2019.
- [8] Dimitris Karagiannis; Harald Kühn, “Metamodelling Platforms”, in *E-Commerce and Web Technologies, Third International Conference, EC-Web*, 2002.
- [9] BOC Asset Management GmbH, “Introduction to ADOxx”, <https://www.adoxx.org/live/introduction-to-adoxx>.

- [10] Dimitris Karagiannis; Dominik Bork, *Meta-Modelling as a Concept: The Conceptualisation of Modelling Methods*, Tech. rep., University of Vienna, 2015.
- [11] Stephen J. Mellor; Marc J. Balcer; Ivar Jacobson, “Executable UML: A Foundation for Model-Driven Architecture”, *0201748045*, 2002.
- [12] Object Management Group, “Model-Driven Architecture - The architecture of choice for a changing world”, <https://www.omg.org/mda/>.
- [13] Douglas C. Schmidt, “Model-Driven Engineering”, <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.106.9720&rep=rep1&type=pdf>.
- [14] B. Selic, “The pragmatics of model-driven development”, <https://ieeexplore.ieee.org/abstract/document/1231146/authors#authors>.
- [15] Mozilla, “Form data validation”, https://developer.mozilla.org/en-US/docs/Learn/HTML/Forms/Form_validation.
- [16] Cisco Visual Networking Index, “Cisco Annual Internet Report (2018–2023) White Paper”, <https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html>.
- [17] Paulo R. M. de Andrade; Adriano B. Albuquerque; Otávio F. Frota; Robson V. Silveira; Fátima A. da Silva, “Cross platform app: a comparative study”, <https://arxiv.org/abs/1503.03511>.
- [18] Mozilla, “JavaScript”, <https://developer.mozilla.org/en-US/docs/Web/JavaScript>.
- [19] Aishwarya A. Kadam; Chandani M. Patil; Harshada R. Chaudhari; S. P. Chavhan, “Web Application Development by Using MEAN Stack”, http://www.ijstmr.com/wp-content/uploads/2017/04/IJSTMR_V2I4_13106.pdf.
- [20] Github, “Github Octoverse Languages”, <https://octoverse.github.com/#languages>.
- [21] Microsoft, “Getting Acquainted with Node.js on Windows Azure”, <https://docs.microsoft.com/en-us/archive/blogs/usisvde/getting-acquainted-with-node-js-on-windows-azure>.

- [22] Stack Overflow, “Developer Survey Results”, <https://insights.stackoverflow.com/survey/2020>.
- [23] NodeJS, “NodeJS”, <https://nodejs.org/en/>.
- [24] Hacker Noon, “How Netflix and Paypal did product transformation using Node.js?”, <https://hackernoon.com/how-netflix-and-paypal-did-product-transformation-using-node-js-22074e13caad>.
- [25] J. O’DELL, “How LinkedIn used Node.js and HTML5 to build a better, faster app”, <https://venturebeat.com/2011/08/16/linkedin-node/>.
- [26] Hacker Noon, “Walmart Strives to be an Online Retail Leader with Node.js”, <https://hackernoon.com/walmart-strives-to-be-an-online-retail-leader-with-node-js-9484a4ff51f7>.
- [27] Sebastian Rupley, “Node.js: An Open Source Tool On the Rise at eBay”, <https://www.ebayinc.com/stories/news/node-js-an-open-source-tool-driving-innovative-development-at-ebay/>.
- [28] Victoria Pimentel; Bradford G. Nickerson, “Communicating and Displaying Real-Time Data with WebSocket”, <https://ieeexplore.ieee.org/abstract/document/6197172>.
- [29] ExpressJS, “Express - Fast, unopinionated, minimalist web framework for Node.js”, <http://expressjs.com/>.
- [30] Yehuda Katz, “HandlebarsJS”, <https://handlebarsjs.com/>.
- [31] Ben Frain, “Responsive Web Design with HTML5 and CSS3”, <https://books.google.at/books?hl=en&lr=&id=uLVrCgAAQBAJ&oi=fnd&pg=PP1&dq=css3&ots=hxNbYTCY3K&sig=qtxAKjypXpYvDlTwzNeYS76WIXc#v=onepage&q=css3&f=false>.
- [32] Mark Otto; Jacob Thornton; Chris Rebert; XhmikosR; Patrick H. Lauke; Gleb Mazovetskiy; Johann-S; Andres Galante; Martijn Cuppens, “Bootstrap”, <https://getbootstrap.com/>.
- [33] M. Kunal Suri, “Modeling the Internet of Things in Configurable Process Models”, <https://tel.archives-ouvertes.fr/tel-02059346/document>.
- [34] Marco Brambilla; Eric Umuhoza; Roberto Acerbis, “Model-driven development of user interfaces for IoT systems via domain-specific components

and patterns”, <https://jisajournal.springeropen.com/track/pdf/10.1186/s13174-017-0064-1>.

- [35] OMiLAB, “Smart City CPS Integration”, <https://austria.omilab.org/psm/content/XSmartCity/info>.