Master's thesis

Master's Programme in Computer Science

Managing Variability in Robot Cooperation Software

Tomi Laurinen

November 25, 2021

Faculty of Science
University of Helsinki

Supervisor(s)

Prof. Tomi Männistö, Ph.D. Niko Mäkitalo, Ph.D. Anna Kantosalo, M.Sc. Simo Linkola

Examiner(s)

Prof. Tomi Männistö, Ph.D. Niko Mäkitalo

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Tomi Laurinen

Työn nimi — Arbetets titel — Title

Managing Variability in Robot Cooperation Software

Ohjaajat — Handledare — Supervisors

Prof. Tomi Männistö, Ph.D. Niko Mäkitalo, Ph.D. Anna Kantosalo, M.Sc. Simo Linkola

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	November 25, 2021	51 pages

Tiivistelmä — Referat — Abstract

Tämä tutkielma esittelee Robot Configurator -ohjelman, jolla voi konfiguroida Cooperative Brain Service -robottiyhteistyöohjelman käynnistykseen liittyviä asetuksia. Muuttuvuuden konfigurointi on ongelma, jossa yritetään valita validi yhdistelmä komponentteja muodostamaan jokin kokonainen tuote. Se on sovellettavissa myös ohjelmistoihin, sillä on olemassa lähestymistapoja missä valmis ohjelmainstanssi muodostetaan erilaisista ennalta valmistetuista komponenteista.

Cooperative Brain Service on Creative and Adaptive Cooperation between Diverse Autonomous Robots (CACDAR) -nimisen Helsingin yliopiston tutkimusprojektin pääasiallinen tuotos. Projektin päämääränä on mahdollistaa autonominen yhteistyö monenlaisten robottien välillä. Tämän vuoksi Cooperative Brain Service -ohjelma on suunniteltu niin, että uusille roboteille ja niiden toiminnoille lisätään tuki uusien moduulien kautta. Käyttöön tulevat moduulit määritellään ohjelman käynnistyksen yhteydessä annettavan JSON-tiedoston kautta. Ongelmana on, että tämän JSON-tiedoston muokkaaminen vaatii muun muassa tietämystä siitä, millaisia moduuleja Cooperative Brain Service -komponentissa on sillä hetkellä sisäisesti toteutettuna. Robot Configurator -ohjelman tarkoituksena on mahdollistaa näiden JSON-tiedostojen luominen käyttäjäystävällisemmällä tavalla, jotta pystyisimme CACDAR-projektissa tekemään tehokkaammin kokeiluja erilaisilla robottiyhteistyöskenaarioilla.

Tämän tutkielman pääasiallinen kontribuutio on Robot Configurator, jonka toiminnan selitän ja dokumentoin monipuolisesti arkkitehtuurikuvauksen kautta. Lisäksi esittelen luomani lähestymistavan Cooperative Brain Servicen käynnistykseen liittyvän muuttuvuuden mallintamiseen.

ACM Computing Classification System (CCS)
Software and its engineering → Software organization and properties → Software system structures → Software system models → Entity relationship modeling
Software and its engineering → Software organization and properties → Software system structures → Abstraction, modeling and modularity

Avainsanat — Nyckelord — Keywords

Software architecture, variability management, software as a service, multiple views

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Software Systems specialisation line

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Tomi Laurinen

Työn nimi — Arbetets titel — Title

Managing Variability in Robot Cooperation Software

Ohjaajat — Handledare — Supervisors

Prof. Tomi Männistö, Ph.D. Niko Mäkitalo, Ph.D. Anna Kantosalo, M.Sc. Simo Linkola

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	November 25, 2021	51 pages

Tiivistelmä — Referat — Abstract

This thesis introduces Robot Configurator, a software for performing variability configuration for a robot cooperation software named Cooperative Brain Service. The variability configuration problem concerns selecting appropriate combinations of components to form a valid complete product. It is applicable to the realm of software as well, as there are approaches where an instance of software is formed from different pre-made components.

Cooperative Brain Service is the main product of Creative and Adaptive Cooperation between Diverse Autonomous Robots (CACDAR), a University of Helsinki research project. One of the main principles of the project is enabling cooperation between various types of robots. In Cooperative Brain Service, this is taken into account by having support for any new robots and actions be added as new modules. In the current implementation, the modules to be loaded are determined during startup, through a manually written JSON configuration file. The problem is, managing such JSON files requires beforehand knowledge of what modules there are implemented in Cooperative Brain Service. As it is crucial to be able to flexibly experiment with different cooperation scenarios in CACDAR, I design Robot Configurator as a tool to assist in the creation and management of these JSON configuration files.

The main contribution of this thesis is Robot Configurator, for which I provide an architecture description that describes and documents it from multiple stakeholder perspectives. Additionally, a novel approach to modeling variability involved in the initialization of Cooperative Brain Service is also introduced.

ACM Computing Classification System (CCS)

Software and its engineering → Software organization and properties → Software system structures → Software system models → Entity relationship modeling

Software and its engineering → Software organization and properties → Software system structures → Abstraction, modeling and modularity

Avainsanat — Nyckelord — Keywords

Software architecture, variability management, software as a service, multiple views

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Software Systems specialisation line

Contents

1 Introduction **1**

2 Background and research problem **4**

2.1 The CACDAR project 4

2.2 Research problem 6

2.3 Research questions 8

2.4 Research method 8

2.5 Artifacts produced 9

3 Designing Robot Configurator **12**

3.1 Capturing stakeholder requirements 12

3.2 Variability modeling 12

3.2.1 Research on Variability in Software Product Lines 13

3.2.2 Capturing variability in Cooperative Brain Service 15

3.3 Configurator software design 16

4 Models of variability in Cooperative Brain Service **18**

4.1 Capability Configuration and Robot Models 18

4.1.1 Robot Platform 19

4.1.2 Capabilities 19

4.1.3 Robot Template 21

4.1.4 Mapping Scheme 21

4.2 Scenario Configuration and Environment Models 22

4.2.1 Ontology Object 22

4.2.2 Environment Knowledge Set 23

5 Robot Configurator architecture **24**

5.1 Architecture description through viewpoints 24

5.2 Context view 25

5.3 Functional view 26
5.3.1 Functional components 27
5.3.2 User workflow 31
5.4 Information view 35
5.4.1 Models of Variability 35
5.4.2 Robot Description 35
5.4.3 Configuration Formats 37
5.5 Development view 39
5.5.1 Configurator API 40
5.5.2 Configurator GUI 40
5.5.3 Launch Server 40
5.5.4 Launch Client 41
5.6 Deployment view 41
5.6.1 Server-side deployment 42
5.6.2 Robot-side deployment 44
5.6.3 Client-side requirements 44

6 Discussion **45**

7 Conclusions **47**

Bibliography **49**

1 Introduction

In this thesis I document the circumstances and design processes behind a configurator software named *Robot Configurator*, and also provide a multi-perspective description of its architecture. The purpose of *Robot Configurator* is to streamline the configuration of settings for a robot cooperation software named Cooperative Brain Service.

The fundamental concept of configuration originates from *product line configuration*, a decades-old practice tracing back to industrial product design concerns (Peltonen et al., 1994). Different customers often have their own specific needs in terms of what a product should be like, which is a problem when significant efforts would be required to design a new product for each and every one of them. A so-called *product line* approach can be used as a more efficient solution (Brownsword and Clements, 1996). It works as thus: first, a wide variety of interchangeable components are designed meticulously for some type of product. Then, new versions of the product can be tailored for customers by selecting valid combinations of suitable components, in a process called *configuration*. Notably, this approach is not limited manufacturing industries; it is very much applicable to software also.

Variability refers to different kinds of abstract optional or alternative elements in a product line. In the context of software, Galster et al. (2011) describe variabilities as the different ways a software can be adapted to various contexts in a pre-planned manner. A typical variability is the presence of some function. With correct combinations of components, the function becomes implemented in the product.

At times, a *configurator* tool may be introduced to help choosing combinations of components in such a way that desired variabilities are implemented (Myllärniemi et al., 2012). However, creating a successful configurator often also requires proper *models of variability* to make its scope of configuration clear. Such models can define constraints that limit what components can coexist in a configuration, for example.

As for configuration in the field of software, there are multiple ways in which a configurable software architecture can be implemented: the entire software can be built from existing smaller software components, or alternatively the software itself may have a range of components available which it loads at runtime. In University of Helsinki robotics cooperation research project CACDAR, we are using the latter approach. However, this has also made

the usefulness of having a configurator system apparent to us, for reasons subsequently described.

The primary artifact being designed in CACDAR is a robot cooperation software, named Cooperative Brain Service, that can be run preferably on a wide variety of robots if reasonably possible. Yet, robotics has long been a field marked by a certain dichotomy: while great advancements are being made, the solutions and technologies themselves tend to be very fragmented between different players of the industry (Gates, 2007). Separate robots have often their own underlying software implementations, so the cooperation software cannot be designed with any single platform in mind. Similarly, we cannot make any assumptions of what kinds of cooperative actions the robots could be capable of, other than potentially anything.

However, what we can do is frame this as a software configuration problem. To make Cooperative Brain Service work on as many robots as possible, it is designed to be *modular*: the base functionality is very generic, and support for new robot development platforms or capabilities is added through new modules as needed. From the configuration perspective, these abstract concepts of robot platforms and capabilities are effectively our models of variability; capabilities can even require the presence of some other capabilities as constraints. The code modules accordingly are our components, because they implement the robot platforms and capabilities in the software.

When an instance of the cooperation software is initialized on a robot, the platform and capability modules that are to be loaded are determined through a manually written configuration file provided as a startup parameter. Unfortunately, there are flaws to this approach: not only is having to manually modify the configuration files cumbersome, but it also requires insider knowledge of what software modules there are currently implemented to know what options there are select from. In other words, though we essentially have a product line like solution where the platform and capability modules are the components, our options for doing configuration with it are severely lacking.

As a result, I designed a configurator system, named *Robot Configurator*, to assist in the creation and management of configuration files used in Cooperative Brain Service. The stakeholders of this software are us CACDAR researchers, and the purpose of the software is to make robot cooperation experimentation with Cooperative Brain Service more straightforward for us.

Robot Configurator consist of four separate components with their own responsibilities. The first component is Configurator API, which contains all configuration logic and

data models. The second is CONFIGURATOR GUI, a Graphical User Interface that allows the user to select appropriate platforms, capabilities and other settings for robots. The remaining two components are LAUNCH SERVER and LAUNCH CLIENT, whose purpose is solely to enable sending configurations to Cooperative Brain Service instances of robots over a network. Figure 1.1 illustrates what components there are interacting in a full *Robot Configurator* setup. Additionally, as the basis of configuration logic in *Robot Configurator*, I also contribute the models used to represent variability in Cooperative Brain Service. These are separated into two categories: Robot models and Environment models.

To start off this thesis, I give more background on the CACDAR project and the configurator approach in Chapter 2. To frame the research of this thesis as design science, the design reasoning behind the development of artifacts is documented in Chapter 3. The models designed to represent variability in Cooperative Brain Service are introduced in Chapter 4. An extensive description of the configurator's architecture from the perspectives of different stakeholder interests is given in Chapter 5. Finally, the significance of this thesis is discussed, both in terms of what has been achieved so far in Chapter 6, and what could be the direction for further research in Chapter 7.

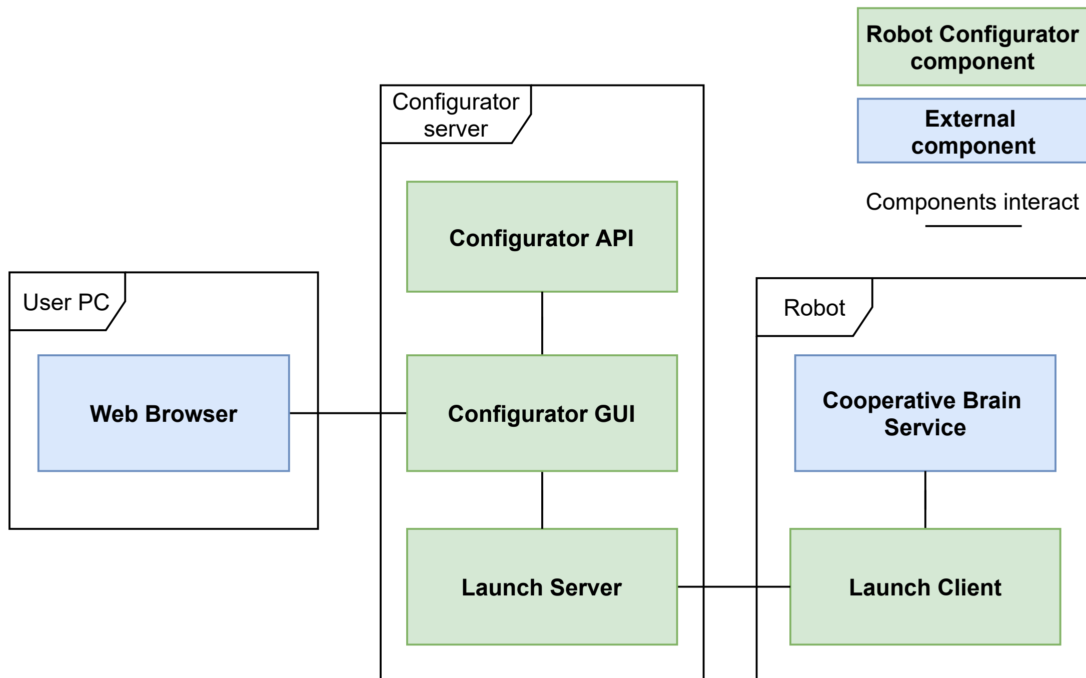

Figure 1.1: Components involved in a complete *Robot Configurator* setup, under the environments they are typically run on.

.

2 Background and research problem

To make clear the inspirations and purposes of this thesis, I explain what the motivations and goals of the CACDAR project are, and describe the circumstances why it warrants the creation of a configurator tool.

2.1 The CACDAR project

Creative and Adaptive Cooperation between Diverse Autonomous Robots (CACDAR) is a University of Helsinki research project aiming to design peer models and create a framework for collaboration between diverse types of robots (Mäkitalo et al., 2021). Interests of the project range from reducing the amount of robots needed for separate tasks, to possibly even facilitating the emergence of some form of creativity within the cooperation efforts.

To better illustrate the type of problems CACDAR project addresses, we have a package delivery scenario as an example. The scenario features two robots with distinct roles: a delivery robot trying to deliver a package to a building, and a cleaning robot native to the building. The delivery robot arrives into the building, where it runs into a problem: it does not have a map of the building. It is only given the name of the target location, but it has no further information about where this location could actually be in the building. To proceed, the delivery robot broadcasts a message asking for help, which the cleaning robot receives. The cleaning robot pauses its current cleaning task, and goes to help the delivery robot. The cleaning robot, knowing already the map of the building, then leads the delivery robot to its target location. At the same time, the delivery robot builds its own map of the building using its radar sensors. After successfully delivering the package, the delivery robot then navigates out of the building using the map it made so far. Should there be obstacles such as doors, it may have to ask for help in opening those on the way back.

The primary goal of CACDAR is to develop a framework that facilitates this sort of flexible, independent cooperation between robots. The gist of this undertaking is in the autonomy of the robots: every participating robot runs its own independent instance of the cooperation software, making it an autonomous agent with its own model of the

surrounding world. In other words, there is no centralized server or decision-making algorithm. To enable robots to reason about their cooperation capabilities in contrast to their peers, a Cooperation Ontology of concepts abstract enough to function as a common language between robots is being created in CACDAR research, separate from this thesis.

Moreover, the framework is also meant to enable swift and flexible robot cooperation experimentation using three different *worlds* of abstraction: simplified 2D world, 3D world that simulates real world, and the real world itself. The idea is to create a feedback loop between the worlds: when we implement a new cooperative action, we may first verify that it works at least on an abstract level, namely the 2D world. When it works, we may then see if this is also the case in our 3D world, and lastly the real world environment.

To put it another way, if there is trouble implementing some cooperative action in the 2D world, there may be fundamental issues with the logic. But when we are implementing the same action for 3D worlds and the real world, we can more certain that any issues faced pertain to the pecularities of those worlds, rather than the fundamental idea, if we have already a successful implementation in the logically pure 2D world.

The implementation of the robot cooperation framework, and the main software artifact of CACDAR, is named Cooperative Brain Service. Cooperative Brain Service is intended to be agnostic of the development platform of the robot, supporting potentially any type of robot that possesses communication capabilities. To make it possible for Cooperative Brain Service to accommodate to any robot's capabilities and limitations at all abstraction levels, it consists of three components that are mainly generic in design. Only Task Runtime, the component responsible for the concrete execution of cooperative actions, requires that support for new platforms and capabilities is added as modules. The remaining two components, Planner and Knowledge Manager, have no such concerns. The Planner component performs decision making at an abstract level using Cooperation Ontology concepts, and Knowledge Manager stores various knowledge also in a platform-agnostic manner. As a separate component from Cooperative Brain Service, we also have the Coop Communication Service that is used for messaging between Cooperative Brain Service instances. Figure 2.1 illustrates what artifacts there are already made in CACDAR, and also what are the new artifacts created in this project.

Also, we use Turtlebot3 (*TurtleBot3 Features* 2021) as our main research robot model in CACDAR, which is why it is occasionally mentioned in the examples of this thesis. Robot Operating System 2 (*ROS 2 Design* 2021) is also used in the examples, as it is the development platform we use with Turtlebot3.

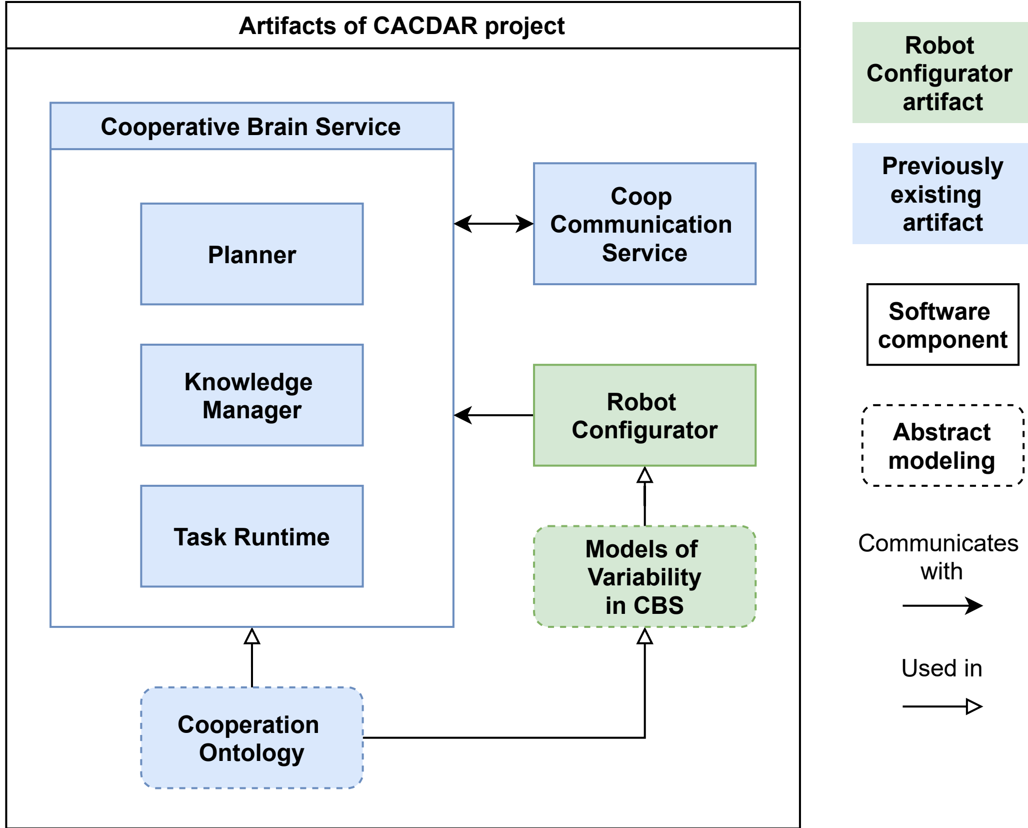

Figure 2.1: All current artifacts of CACDAR project, with the artifacts produced in this thesis marked in green. CBS is short for Cooperative Brain Service.

2.2 Research problem

Due to the need to support potentially any types of robots, the Task Runtime component of Cooperative Brain Service employs a modular design where functionality can be extended through new code modules. The code modules to be loaded, among other information, are defined in a JSON (JavaScript Object Notation) file (Pezoa et al., 2016) the Cooperative Brain Service loads at startup. While this makes switching the modules to be loaded technically a matter of rewriting several lines in a JSON file, there is a certain issue from the user perspective: the user cannot know what modules there are available without direct access to the module folders of Task Runtime. This heavy burden of knowledge on the user would make the software cumbersome to use in the long run. As Cooperative Brain Service is specifically meant to enable easy robot experimentation in different environments and abstractions, some form of more user-friendly solution is definitely warranted. This leads to the primary *research problem* of this thesis: how can we design a variability configuration tool that successfully simplifies the deployment of Cooperative Brain Service?

The modular approach to initializing software is explored in research on *variability in software architectures*. The current implementation which Cooperative Brain Service uses corresponds to a variation realization mechanism called *adaptation on startup* (Bachmann and Bass, 2001). Section 3.2.1 features a more in-depth look into these methods of realizing variability, and various approaches to modeling variability also.

The combinations of modules and other information are called *configurations*. One solution to managing a variable software is developing a system that assists in finding valid configurations, an approach known as a *configurator* tool (Myllärniemi et al., 2012). Notably, a configurator also needs proper models of the variability it configures, to set it clear what is legal to include in a configuration. Thereby, we will define both models of variability involved in initializing Cooperative Brain Service, and create a new configurator system, titled *Robot Configurator*, to assist in creating the JSON configuration files user for Cooperative Brain Service. The previously introduced Figure 2.1 pictures how these newly created artifacts fit with the existing CACDAR artifacts.

A distinction should be made that the main function of a configurator can be considered to be in giving feedback on validity of configurations. However, the purpose of *Robot Configurator* is also to streamline the use of Cooperative Brain Service for us CACDAR researchers. Therefore, I look into user interface solutions suitable for the purpose.

An another concern is preventing the user from accidentally sending incompatible configurations that cause errors in Cooperative Brain Service. Even though Task Runtime itself is capable of running the code logic, it is of not much use if the robot being controlled is not compatible and cannot communicate with the code being run. Therefore *Robot Configurator* needs to feature some form of validation process to ensure incompatible modules are not loaded for a robot inadvertently.

Moreover, the architecture of *Robot Configurator* is also to be documented comprehensively. As the usability of Cooperative Brain Service is of high importance for the larger CACDAR project, it is appropriate to describe of how exactly the *Robot Configurator* streamlines the use of Cooperative Brain Service. However, it is also vital to document how *Robot Configurator* is maintained and developed further, to ensure it can be used in the future too.

2.3 Research questions

As discussed in this chapter, we have decided to use a configurator tool to simplify the usage of Cooperative Brain Service. Therefore, the overall research problem and main purpose of this thesis can be summarized with one question: How do we design a variability configuration tool that successfully simplifies the deployment of Cooperative Brain Service?

In addition, we may define several *research questions* to further clarify the principal concerns of this thesis.

- RQ1: How can the configurator tool streamline Cooperative Brain Service configuration for the user?
- RQ2: How do we validate that a configuration is compatible with a robot?
- RQ3: How do we comprehensively document the architecture of *Robot Configurator*?

2.4 Research method

Because this work involves modeling variability and creating a whole new configurator component, I have chosen *design science* (Brocke et al., 2020) as the appropriate research methodology. Design science research concerns innovating novel artifacts that solve real-world problems, all while building design knowledge on the matter. The intention is to not to just document the artifact itself, but to also knowledge learned from its creation that could be useful for future research. This documentation of knowledge is effectively design science's version of research data collection.

The Design Science Research Methodology process described by Peffers et al. (2007) can be used to illustrate how different Chapters and Sections of this thesis correspond to existing design science methodology. The process consists of six phases: (1) Problem identification and motivation, (2) definition of objectives for a solution, (3) designing and development, (4) demonstration, (5) evaluation, and (6) communication. Also, as this six-part process is meant to be iterative, the knowledge learned in (6) could be utilized in the development of further artifacts.

(1) The research problem of this thesis is explained and motivated in detail in Sections 2.1 (The CACDAR project) and 2.2 (Research problem).

(2) Definition of objectives: The objectives of this research are defined in Sections 2.2 (Research problem), 2.3 (Research questions) and 3.1 (Capturing stakeholder requirements).

(3) Designing and development: The design and development processes are discussed in all of Chapter 3 (Designing Robot Configurator).

(4) Demonstration: The usage of *Robot Configurator* from user perspective is demonstrated in 5.3.2 (User workflow).

(5) Evaluation: The evaluation of *Robot Configurator* is discussed in Chapter 6 (Discussion).

(6) Communication: Chapters 3 (Designing Robot Configurator), 4 (Models of variability in Cooperative Brain Service) and 6 (Discussion) present what work this research is based on and what new knowledge is contributed. For the stakeholders, as in the CACDAR researchers meant to use the software, Chapter 5 (Robot Configurator architecture) communicates how the artifact itself works. I borrow from existing architecture description practices to employ a comprehensive approach.

2.5 Artifacts produced

Both concrete and abstract artifacts are produced in this thesis. The concrete artifacts are the components of *Robot Configurator*: Configurator API, Configurator GUI, Launch Server and Launch Client, listed in Table 2.1.

The abstract artifacts are models for capturing the variability involved in starting up Cooperative Brain Service, labeled under two categories: Robot Model and Environment Model. However, several of these models do not originate from this thesis, but the primary CACDAR research instead. Table 2.2 can be used as a reference to separate what models originate from this thesis, and what is applied from existing CACDAR research. The artifact Tables also show how the names of the artifacts are highlighted differently depending on the category, e.g. Configurator API for Configurator Components.

On the other hand, I have not created any artifacts based on formal software variability modeling methods in this thesis. As Chapter 3 discusses, the configuration validation problem is trivial to the point of not being interesting to formalize, and the variability models themselves are still entirely subject to change and improvement. I have therefore decided to focus on the architectural aspects of design instead, as seen in Chapter 5, a documentative artifact.

Configurator Component	Responsibility	Section
Configurator API	Contains models of variability, performs validation of configurations.	5.3.1
Configurator GUI	User interface for creating configurations, communicates with Configurator API to receive models and perform validation, complete configurations can be sent to Launch Server.	5.3.1
Launch Server	Middleman server for passing configurations on to Launch Clients.	5.5.3
Launch Client	Robot-side client, receives configurations to launch Cooperative Brain Service with.	5.5.4

Table 2.1: The *Robot Configurator* component artifacts created in this thesis.

Robot Model	Responsibility	Section
Robot Platform	The software implementation of a robot (ROS for example).	4.1.1
Task	Abstract concept of activity, common between robots. Implemented by combinations of **Actions**. Not tied to any **Robot Platform**.	4.1.2
Action	More specific than a **Task**, the same **Action** can be implemented by different **Logics**. Belong to **Robot Platforms**.	4.1.2
Logic	The actual implementation of an **Action**, corresponds to a code module implemented in Task Runtime, uses **Mappings**.	4.1.2
Mapping	**Robot Platform** specific interface used to communicate with a robot in a **Logic** code module.	4.1.2
Robot Template	Represents a robot model (TurtleBot3 for example) as a list of **Mappings**, used in configuration validation. Specific to a **Robot Platform**.	4.1.3
Mapping Scheme	Rules that determine how **Mappings** are mapped, specific to a **Robot Platform**.	4.1.4

Environment Model	Responsibility	Section
`Ontology Object`	Locations, objects or even other robots in the environment, used in reasoning about the environment.	4.2.1
`Environment Knowledge Set`	Environment-specific knowledge used in **Logics**, provided to robots to simplify setting up cooperation scenarios.	4.2.2

Table 2.2: The model artifacts used in *Robot Configurator*. Artifacts created in this thesis are marked in light green, and artifacts from other CACDAR research are in light cyan.

3 Designing Robot Configurator

Design science research is set apart from plain system building mainly by the act of documenting knowledge that could be useful for further similar studies. This chapter documents what design processes were involved in creating the artifacts of this thesis.

3.1 Capturing stakeholder requirements

The basis of design science lies in creating solutions for stakeholder needs. In the case of this thesis, the stakeholders are the members of the CACDAR research project, including myself. I obtained the stakeholder requirements for *Robot Configurator* in weekly CACDAR meetings, through discussions with the other researchers.

Three fundamental functions required of *Robot Configurator* shaped up in the discussions: (1) it is to capture software variability involved in the deployment parameters of Cooperative Brain Service, (2) provide an intuitive method for the user to configure variables with, and (3) enable checking that the variables chosen form a valid configuration. The first (1) of these is a function concerning abstract designs, while the latter two (2) (3) require concrete software design. I will therefore discuss the design processes of *Robot Configurator* as two separate categories: the abstract modeling of variabilities, and the concrete design of the configurator.

Also, during the design process of *Robot Configurator*, I received regular feedback in the weekly discussions on how I should develop the *Robot Configurator* further. This helped me decide what the Configurator API model formats should be like, and what elements the Configurator GUI should feature.

3.2 Variability modeling

To form the basis for *Robot Configurator*, an appropriate approach for modeling the variability in Cooperative Brain Service was needed. For this, I explored what kinds of solutions have already been researched in the field, and also looked into what sort of variability Cooperative Brain Service itself features.

3.2.1 Research on Variability in Software Product Lines

The product line approach was introduced in Chapter 1, but it should be explored further how it relates to software in particular. There are multiple definitions of what a Software Product Line, or Family, is. One such definition, by Bosch (2000), defines that a "Software Product Line consists of a product line architecture and a set of reusable components that are designed for incorporation into the product line architecture. In addition, the product line consists of the software products that are developed using the mentioned reusable assets". Applying this definition, Cooperative Brain Service and its components, particularly Task Runtime and the different cooperation action logic modules loaded within it, can be considered a Software Product Line, along with all of the possible runtime instances of Cooperative Brain Service. However, it should be noted that a Software Product Line typically already contains most of the components necessary for creating new products. Yet in our case, component reuse is more limited, as the introduction of new robot platforms and actions usually require creating new modules for Task Runtime.

Furthermore, if the Software Product Line is designed in a manner where models can be loaded during runtime, it can be considered a Dynamic Software Product Line (Hallsteinsen et al., 2008). Cooperative Brain Service is technically not a Dynamic Software Product Line presently, as the logic modules are loaded only during startup. However, the current architecture of Task Runtime would allow the easy implementation of runtime module switching for any logic module not currently running, if it would come up as a purposeful feature to have.

In research that classifies different methods for realizing variability, the current implementation used in Task Runtime is designated as *adaptation on startup* by Bachmann and Bass (2001). In adaptation on startup, the software is designed so that it can handle all possible variants, and configuration files are used to tell the software how it should adapt. If applied to the taxonomy of variability techniques by Svahnberg et al. (2005) instead, the current approach would belong under the category *Condition on Constant*: the Cooperative Brain Service configuration parameters determine what modules are loaded in Task Runtime on startup, but the modules cannot be changed anymore at runtime. However, if we were to modify Task Runtime so that the loaded modules can be switched dynamically even when it is running, the approach would change to *Condition on Variable*. In the terms of Bachmann and Bass (2001), this would be *Adaptation during normal execution*: adding and removing modules as the program is running.

Nonetheless, the module loading logic of Task Runtime was implemented long before work on this thesis and *Robot Configurator* began. The implementation concerns of Task Runtime are not the focal point of this thesis; it is more appropriate to look into what options there are for modeling variability in Cooperative Brain Service.

Over time, numerous methods have been explored in software variability modeling research. Many of these methods belong to so-called *feature modeling* paradigm (Kang et al., 1990; Czarnecki et al., 2002; Asikainen et al., 2006). In feature modeling, the variability in a software architecture is captured as *features*. The definition of feature varies between the methods: it can be "an end-user visible characteristic of a system" (Kang et al., 1990), or an "a distinguishable characteristic of a concept (e.g. system, component, etc.) that is relevant to some stakeholder of the concept" (Czarnecki et al., 2002). The *feature model* then is what captures the variability of a system as the total of these features. As an alternative approach to feature modeling, the variability of a software can be modeled also as *components* of a product line (Asikainen et al., 2003).

Of particular interest to me is that both feature and component modeling feature *constraints* between elements. In fact, both Forfamel (Asikainen et al., 2006) and Koalish (Asikainen et al., 2003) models can be translated to Weighted Constraint Rule Language (Simons et al., 2002), which is used for systematic reasoning about validity. If a configuration made by a user meets the constraints set in the configuration model, the configuration can be deemed internally valid. As per stakeholder requirement (3), this validity checking is of great interest to us.

I however decided to look into a more recent implementation in variability managing: the Kumbang language (Asikainen et al., 2007), which synthesises both feature and component modeling approaches to software variability as Forfamel (Asikainen et al., 2006) and Koalish (Asikainen et al., 2003) respectively. Kumbang had been applied successfully in previous projects concerning configuration (Myllärniemi et al., 2012; Tiihonen et al., 2018), and it could be translated into the Constraint Satisfaction Problem format used by the Choco solver (Prud'homme et al., 2016). In other words, the combination of Kumbang and Choco solver was a good candidate for covering both variability modeling and configuration validation concerns of *Robot Configurator*. For example, the platforms of robots and cooperative actions could be mapped as features in Kumbang. But before looking further into these methods, I started to outline what variability the initialization of Cooperative Brain Service would involve.

3.2.2 Capturing variability in Cooperative Brain Service

In practice, the process of modeling variability involves identifying so-called *variation points* (Bachmann and Bass, 2001) relevant to the problem at hand. In the context of robotics, these variation points can be both various general capabilities, like whether the robot has a camera, and more specific attributes like the pixel count of the camera. This meant that I would need concepts that describe properties of robots in a manner that is not tied to any specific platform, but still informative enough to be useful. Fortunately, we had already explored this matter together in the CACDAR project, prior to my work on this thesis, which is why there already were applicable concepts in use in Cooperative Brain Service: capabilities of robots, modeled in four layers: **Tasks**, **Actions**, **Logics** and **Mappings**. Describing the state of the example robot having a camera would fit in the lowest level of these abstractions.

As mentioned in stakeholder requirement (3), being able to check whether a configuration is valid was also a major motivator behind using the configurator approach. However, after the decision to use the capability models as the basis of variability modeling, it became apparent to me that the problem of checking whether a robot is capable of running a configuration is essentially trivial in terms of complexity. This meant that specifically using formal variability modeling methods, such as the combination of Kumbang and Choco, for this simple problem did not appear very purposeful nor interesting in terms of scientific contribution. Accordingly, the focus of this thesis instead moved on to providing more extensive architectural descriptions, as they would be also be of use to stakeholders.

Moreover, though I decided to not apply any formal software variability modeling method in the scope of this thesis, the option of trying to apply Kumbang or something similar may be worth considering in the future, particularly if the variability models were to be extended so that the configuration problem increases in complexity.

There was also still more to the startup parameters of Cooperative Brain Service than just determining the capability modules. Namely, Cooperative Brain Service is also given various data such as knowledge and initial goals on startup, to better control what robots do in demos. From the modeling perspective, this meant we needed concepts for describing environmental information that had significance in robot cooperation.

As the the robot itself and the environment were essentially different domains, I separated configuration into two types, both with their own models: configuration related to the composition of the robot, and configuration related to the current cooperation environ-

ment. Named Capability Configuration and Scenario Configuration respectively, Chapter 4 will describe more about these configuration types.

3.3 Configurator software design

Here I describe design choices pertaining to the software design of *Robot Configurator* itself, and how I ended up separating it into four different artifacts. They are Configurator API, Configurator GUI, Launch Server and Launch Client, highlighted as seen here to make them easier to distinguish.

From early on in the work on this thesis, it was intended for the design of the *Robot Configurator* to have the configurator interface and the configurator itself be separate components. An initial prototype Graphical User Interface was made using Qt (Nord and Chambe-Eng, 2021) for Python, as a multi-platform desktop application. However, in the CACDAR meetings it was clarified that it would be preferable to be able to host the GUI over network, and make it usable through internet browsers. This would enable on-the-fly creation and validation of configurations for any cooperative robot connected to the network of the configurator, as a "configurator-as-a-service" type of approach (Myllärniemi et al., 2012).

Accordingly, the design evolved into a system that has two primary components: a browser-based Graphical User Interface as the frontend, and the configurator itself as a REST (Fielding and Taylor, 2000) backend service, containing all models and rules of variability. These two services became to be known as Configurator GUI and Configurator API respectively.

The user interface, Configurator GUI, was implemented using Vue.js (You, 2021), due to its reputation as an easy to use JavaScript framework. While the framework itself was relatively simple to use, there was some initial challenge to using it due to my lack of prior experience in frontend development in general. Nevertheless, all of the intended features were implemented successfully in the end.

The implementation of configuration validation was also an important design point. There were two notable benefits motivating it: firstly, sending a broken configuration to Cooperative Brain Service can lead to an early failure at best, and unexpected behavior at worst. By validating configurations beforehand, it is distinctly easier to keep track of such risks. Secondly, the existence of a proper validation process helps guide the user in selecting a

valid combination of parameters, especially when the user is given feedback on what went wrong with the configuration parameters. Having validation be done through its own *solver* service was considered, inspired by the configurator-as-a-service approach (Myllärniemi et al., 2012). However, as it became apparent that the computational complexity of the validation as virtually trivial in comparison, the validation logic was implemented simply as a REST endpoint within Configurator API backend instead. This was convenient also in the sense that all models of variability, including the constraints used in validation, were already present in the backend.

Moreover, I also needed to implement components for communication between the configurator and robots, namely to make it possible to send configurations. Contrary to how Cooperative Brain Service is about promoting peer-to-peer autonomous cooperation, a centralized server was selected as the approach. This is because it would not be sharing information between robots, but simply setting their configurations. However, trying to incorporate the server into Configurator GUI, a Vue.js application, turned out to not be very elegant nor practical. Therefore the middleman server was made its own separate component. For communication I used Socket.IO (Rauch, 2021), a socket communication component I was already familiar with. The resulting component was named Launch Server, due to its role in launching any connected robots by sending configurations to them. Respectively, Launch Client was created as the robot-side Socket.IO endpoint that would launch Cooperative Brain Service upon receiving a configuration. Lastly, Configurator GUI was also added a Socket.IO client, as the client logic was readily supported by Vue.js.

An additional mention goes to the design of *configuration formats*, as in contrast to the variability models. These are the formats in which the configurations are exchanged between the configurator components and sent to the Cooperative Brain Service. As it was not immediately apparent what sorts of configuration formats would be suitable, it was natural to design them hand-in-hand with the *Robot Configurator*, particularly with the Configurator GUI.

In the Architecture Description (Chapter 5), I give a more detailed, multifaceted description of all of these components, configuration formats included.

4 Models of variability in Cooperative Brain Service

In Chapter 3, I separated configuration in *Robot Configurator* into two types: Capability Configuration and Scenario Configuration. In this chapter, I introduce these configuration types, and also describe variability model artifacts under categories that correspond to the two configuration types: Robot Models and Environment Models.

It should also be emphasized that I am not describing the actual formats of created configurations in this chapter, but the models of data that form the basis for the creation of configurations in *Robot Configurator*. This creation of configurations happens in the views explained in Section 5.3.1. The data formats of created configurations are more of an implementation concern, and are thus described in the Information view of the architecture description, Section 5.4.

4.1 Capability Configuration and Robot Models

As mentioned in Chapter 2, the Task Runtime component of Cooperative Brain Service loads the capabilities of a robot as code modules. I have decided to separate configuration concerning the composition of the robot itself as its own configuration type, Capability Configuration, and have the relevant models be called *Robot Models*.

As the basis of Robot Models, I have utilized several concepts from existing CACDAR research, namely the **Robot Platform** and the four-layered capabilities, as they are involved in the initialization of Cooperative Brain Service. Moreover, I also contribute additional models Robot Template and Mapping Scheme, which currently are only to be used in logic specific to *Robot Configurator*.

It should be stressed that unlike Environment Model, which already defines everything belonging to an environment in a single instance, the Robot Models are separate pieces: they are used for different purposes in *Robot Configurator* for the purpose of creating a coherent Capability Configuration, which finally is the complete definition of a robot.

To make them stand out better, capability-related models are highlighted in **bold**.

4.1.1 Robot Platform

Robot Platform refers to the underlying software level implementation of a robot which determines how the robot is interfaced with. Examples of **Robot Platforms** would be Robot Operating System (ROS) (Quigley et al., 2009), a leading open-source robotics development platform, and its successor ROS2 (*ROS 2 Design* 2021). Considering how platforms can have very different paradigms regarding how controlling and developing a robot works, we cannot make any assumptions about reusability of components or operational code between them. For the abstract decision making parts of Cooperative Brain Service, this is not an issue. But in Task Runtime, which is responsible for running the cooperation activities, support for different platforms is realized by having each platform be implemented through its own code module.

As for what significance **Robot Platforms** have in *Robot Configurator*, the subsequent explanations of **Actions**, **Mapping Schemes** and **Robot Templates** will mention how they are tied to **Robot Platforms**.

4.1.2 Capabilities

Instead of describing a robot's physical configuration for instance, CACDAR takes an approach more appropriate for our cooperation context: four different levels of abstractions of what a robot is currently capable of. These four layers are **Tasks**, **Actions**, **Logics** and **Mappings**, in order from abstract to concrete. Strictly speaking, there is no overarching concept of "capability" in Task Runtime logic, but I will use it as a catch-all term for these four concepts.

It must be also emphasized that these capability concepts are not contributions of this thesis; they originate from earlier CACDAR research done in tandem with the development of Cooperative Brain Service, and are heavily tied to the logic of the Task Runtime component in particular. Two of these concepts, **Task** and **Action** originate from DOLCE + DnS Ultralite (Gangemi, 2017) ontology, which is used in developing the Cooperation Ontology in the main CACDAR research. **Logic** and **Mapping**, on the other hand, are concepts we invented for use in Task Runtime specifically.

Task. **Task** is the abstract concept of activity, used in Cooperation Ontology as a common language between robots. Thus, it is also agnostic of the Robot Platform. A single

Task is implemented by some number of **Actions**. Also, different combinations of **Actions** can accomplish the same **Task**. The entire package delivery illustrated in Chapter 2 would be an example of a **Task**, named simply Delivery.

Action. An **Action** describes some interaction or feature a robot is capable of. The goal of an **Action** can potentially be implemented in many different ways, which is why its implementation is made a separate concept, **Logic**. **Actions** also need to distinguish what **Robot Platforms** they are intended for. Multiple platforms can be defined in case there is much similarity in between, such as with ROS and ROS2, but this may be subject to change.

As an example of what is currently implemented in Task Runtime, the Delivery **Task** consists of **Actions** AskHelp, SendGoalId and Follow, in that order. AskHelp asks nearby guide robots for help, SendGoalId sends the ID of the desired location to a guide that agreed to help, and Follow has the robot following the guide robot to the location. On the other hand, nothing prevents separating these three **Actions** into their own **Tasks**.

Logic. A **Logic**, shortened from Action Implementation Logic, is the name of the actual set of instructions that make a robot act out an **Action**. Essentially, a **Logic** corresponds to a similarly named code file loaded in Task Runtime. A **Logic** usually requires **Mappings**, and can also contain **Logic Settings**. **Logic Settings** are parameters specific to the **Logic** that affect how it is carried out, and can be adjusted in the CONFIGURATOR GUI for any **Logics** selected for the Capability Configuration. Taking the Follow **Action** as an example, there are two options for a **Logic**: follow_with_poses and follow_with_bluetooth. The former has the following done through QR codes and comparisons of rotational poses, while the latter uses Bluetooth signal strength to let the follower estimate how far away the guide is.

Mapping. A **Mapping** defines some interface of the robot, necessary for the implementation of a **Logic**. What the **Mappings** actually are like depends entirely on the **Robot Platform**. Let us use ROS2 as an example. In case of ROS2, **Mappings** include Topics, Services and Action Clients*. Topics are used for communicate with ROS2 robots by subscribing to Topics that provide information of the robot's state, and publishing to Topics that control the robot's behavior. Services and Action Clients are also types

*The Action Client of ROS2 is separate from our concept of **Actions**.

of abstractions for communication with the robot. As an example, a Navigation Action Client communicates with a Navigation Server by sending goal coordinates to navigate to, after which the server gives the client status updates of the navigation process. In our follow_with_poses **Logic** made for ROS2, "odometry" topic is listened to for rotation information, and "cmd_vel" topic is used to send movement commands to the robot so that it tries to keep the QR code in its view.

Because **Tasks** can consist of multiple **Actions**, and an **Action** can have multiple implementation **Logics**, they can together be represented as a tree structure. This capability tree is employed in CONFIGURATOR GUI, as described in Section 5.3.1.

4.1.3 Robot Template

To make it possible to reason whether a robot meets all **Mapping** requirements of a Capability Configuration, there needs to be a definition of a robot to compare against. One desirable approach could be that of determining a robot's **Mappings** through some type of program that analyzes the robot automatically. Unfortunately, implementing such a solution in a robust enough manner would most likely necessitate a considerable amount work, and thus is not currently reasonable within this thesis. As a more pragmatic solution, I introduce **Robot Templates**. A **Robot Template** lists the **Mapping** names of a robot, and can also optionally contain **Robot Settings** to be adjusted in the CONFIGURATOR GUI similarly to **Logic Settings**. **Robot Settings** may involve limiting the robot's move and turn speeds, for instance. A **Robot Template** is also tied to the **Robot Platform** of the robot.

4.1.4 Mapping Scheme

Even within the same **Robot Platform**, there may be multiple ways in which **Mappings** are mapped. Taking ROS2 Topic **Mappings** as example, a Camera topic may be mapped differently between real world and 3D simulation even for the same type of robot. Yet on the other hand, it is also possible for different robots to have identical mappings. Hence, I have made the intentional decision of not including **Mapping** definitions in a **Robot Template**. Instead, they are provided as their own **Mapping Schemes**, which are specific to a **Robot Platform**, but separated from **Robot Templates**.

4.2 Scenario Configuration and Environment Models

The purpose of a robot is to interact with the world surrounding it. To set up various experiments reasonably in CACDAR, we need to be able to give the robot information of its surroundings, and also set various parameters such as the robot's goals and settings based on the current environment. In contrast to Capability Configuration, where the domain is strictly the robot's own composition, I introduce a separate type of configuration for the "external" world as Scenario Configuration. The "scenario" in the name refers to how these parameters are crucial in setting up cooperation demo scenarios.

To configure scenarios, we need a standardized way to describe the surrounding environment the cooperation scenario takes place in. For this purpose I define *Environment Models*, extended from "Environment Description", a prototype draft from earlier CACDAR research. These consist of lists of two types of objects, `Ontology Objects` and `Environment Knowledge Sets`, emphasized in the manner shown. The original Environment Description model only contains a list of `Ontology Objects`. However, specific to this thesis, I chose to extend it with a list of `Environment Knowledge Sets`, as a means of supporting environment-specific information for **Logics** in a practical way. As this new model no longer only describes the environment as `Ontology Objects`, but contains additional information for convenience in configuration, I will name it "Environment Model" for it to be in line with "Robot Model". Notably, unlike the Robot Model concepts which act as separate pieces, a single Environment Model instance contains everything describing the environment.

4.2.1 Ontology Object

The concept of `Ontology Object` as used in this thesis is an early draft from CACDAR research on Cooperation Ontology. They can be locations, objects or even other robots, determined by a type field, and their purpose is to be used for reasoning about the environment in the decision making components of Cooperative Brain Service. However, currently only `Ontology Objects` of type "Location" have a role in the CONFIGURATOR GUI of *Robot Configurator*: the user can define locations where **Tasks** are to be done. It is then up to the Planner component of Cooperative Brain Service to interpret this information.

4.2.2 Environment Knowledge Set

Knowledge of the environment the robot operates in is beneficial if not crucial for many **Logics**. Ideally, robots would learn about the environment cumulatively as much as possible through their own observations. However, to simulate knowledge difference between robots, and to add an element of control that helps in creating purposeful scenarios, there has to be some method to provide information to robots beforehand.

Taking the Package Delivery scenario as an example, the Guide robot needs knowledge of the different locations in the environment, and the Delivery role needs to have some idea of a target location. Inspired by this need, a Scenario Configuration can contain initial environment knowledge to be included in a robot's knowledge base at launch.

To make it possible to include such environment knowledge other than writing by hand, I introduce pre-made `Environment Knowledge Sets`. `Environment Knowledge Sets` are included within Environment Models, and can be viewed and chosen in the Configurator GUI. In the description of the Package Delivery demo for example, we have two `Environment Knowledge Sets`: one named Helper, which includes the environment locations in the forms of IDs and coordinates, and another named Delivery, which only features the ID of the target location. On the other hand, if no such information is provided, the Guide and Delivery **Actions** cannot be performed due to the absence of location information required for their **Logics** to function.

5 Robot Configurator architecture

In this chapter, an established software architecture documentation methodology by Rozanski and Woods (2012) is introduced first. The architecture of *Robot Configurator* is then described through multiple perspectives, or *viewpoints*, presented in the methodology.

5.1 Architecture description through viewpoints

Describing software architecture in a meaningful manner can be a challenging task, as software design is a significantly multifaceted endeavor. To adequately document the design of *Robot Configurator* while avoiding well-known mistakes, I borrow from established work on architecture description practices, and use the guidebook *Software systems architecture: second edition* (Rozanski and Woods, 2012) as the basis for architecture descriptions in this thesis. The book claims to take a reasonable stance to the realities of architecture description, and particularly the multi-perspective approach it describes is useful for my documentation purposes.

First of all, Rozanski and Woods (2012, p. 23) emphasize that attempting to explain all the different design choices of differing abstraction levels at once has historically proven to be not very effective. Rather, software may be described through multiple *architecture views*. A view describes the software, or a part of it, to some perceived group of *stakeholders*. The idea is to address the *requirements* of the group using concepts understood by it. Here, an important distinction to make is that the term *stakeholders* refers not only to the direct end users of the software, but all people involved with the software in some way, such as developers and maintainers.

Nonetheless, it can be a nontrivial task to determine what concepts and models are the most meaningful to different stakeholder groups. This is why there are established models for common types of views, called *viewpoints* (Rozanski and Woods, 2012, p. 24). A viewpoint is essentially a template that describes a certain type of view and what should be included in it. The Functional viewpoint (Rozanski and Woods, 2012, p. 277), for example, requires a description of the functional elements and interfaces of the architecture in a manner all stakeholders should be able to understand.

In this chapter, the architecture of *Robot Configurator* is described using viewpoints of Rozanski and Woods (2012, p. 255). Of them, I chose to use Context, Functional, Information, Deployment, and Development viewpoints. The Concurrency and Operational viewpoints I left out: there is very little concurrency in *Robot Configurator* and essentially no race conditions or competition for resources, and the latter is meant for systems where failure would have critical implications, but ours is meant strictly for explorative research and thus has no such concerns.

5.2 Context view

Context viewpoint (Rozanski and Woods, 2012, p. 257) treats all inner functionality and components of the software architecture as a black box, and is only concerned with what other, external software it interacts with. The idea is to call attention to the environment and purposes the software is developed for. Context viewpoint is meant to be useful to anyone concerned with the software's place in the larger picture, and thus is best kept short and simple.

The role of *Robot Configurator* is to provide configurations, essentially launch parameters, for Cooperative Brain Service. Therefore as far as the software context goes, the sole software component *Robot Configurator* interacts with is Cooperative Brain Service, as pictured in Figure 5.1. Multiple robots' Cooperative Brain Services can connect to a single instance of *Robot Configurator*. Any connected robots can then be launched with saved configurations from the user interface of *Robot Configurator*. Configurations can also be downloaded as JSON files (Pezoa et al., 2016) and provided directly to Cooperative Brain Service in launch, without having to connect to *Robot Configurator* at all.

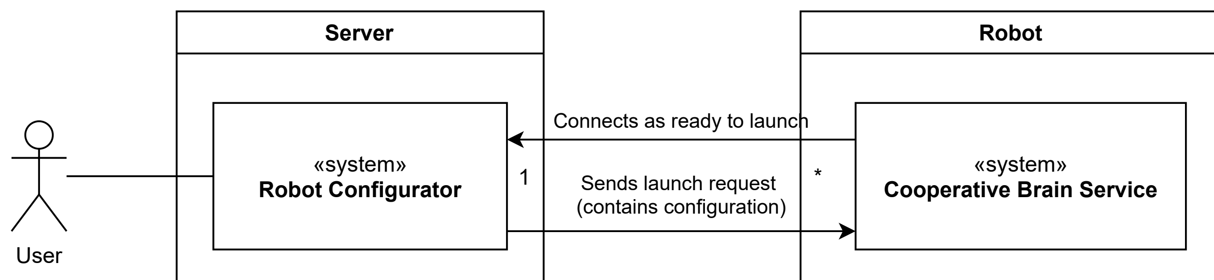

Figure 5.1: The context of *Robot Configurator*. There can be any number of Cooperative Brain Services connected to a single *Robot Configurator* (signified by the * symbol)

5.3 Functional view

Functional viewpoint (Rozanski and Woods, 2012, p. 277) is described as the cornerstone of almost all architecture descriptions. Its purpose is very straightforward: describe what the software actually does, in a manner any potentially interested parties can understand. Anyone working with Cooperative Brain Service, the main artifact of CACDAR, evidently is the primary audience. However, as a research project, the software should be understandable to anyone interested in the research of CACDAR. Therefore, I will aim to keep this view comprehensible even without beforehand knowledge of the concepts.

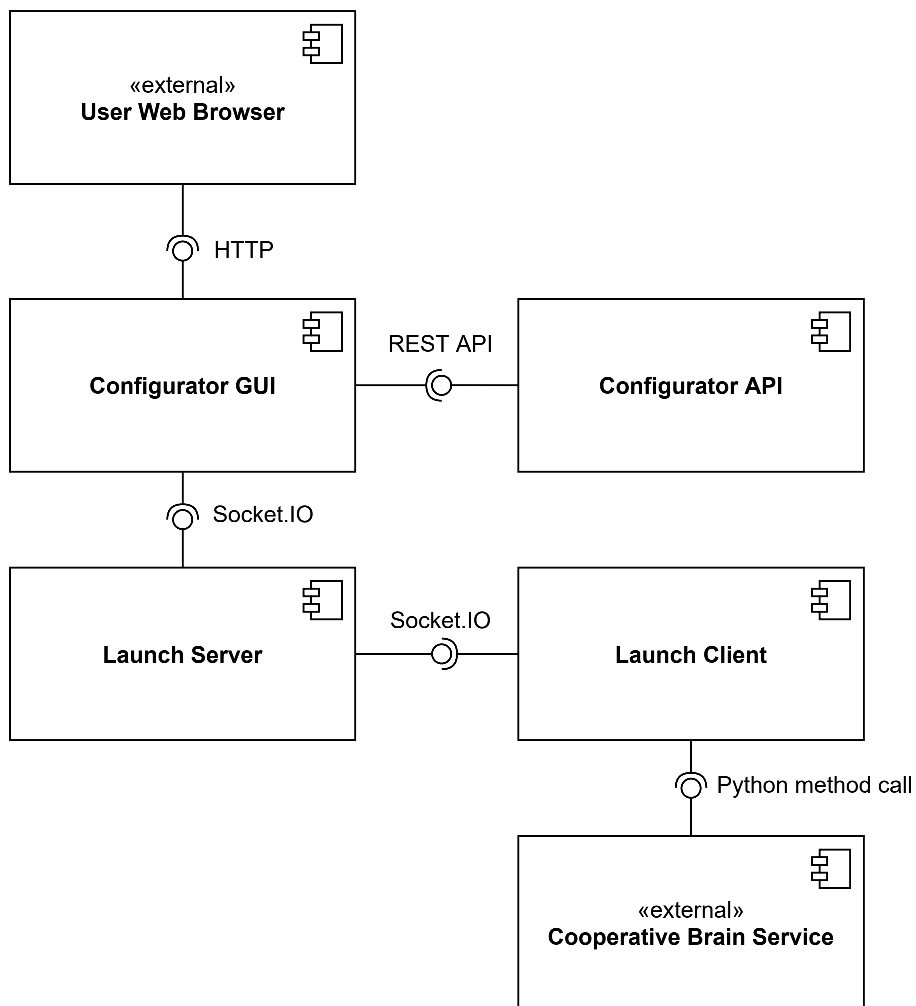

Figure 5.2: Functional components of *Robot Configurator* and the interfacing methods between them.

5.3.1 Functional components

As a whole, the purpose of *Robot Configurator* is to guide in creation of configurations necessary for launching Cooperative Brain Service, the robot cooperation software of CACDAR. In practice, it consists of four separate *Robot Configurator* components with distinct functional purposes. These are Configurator API, Configurator GUI, Launch Server and Launch Client.

Configurator API

The essence of a configurator system lies in models that describe variation points, and validation that checks that all the constraints between variables are met. In *Robot Configurator*, these models are as described in Chapter 4. The role of Configurator API is to contain these configuration concerns in a single backend component, as a RESTful API. As thus the two functions, or endpoints, provided by this API answer these two concerns.

The first REST endpoint is named "/available-variables". It returns all variability models currently included as a JSON object. The format of this object is specified in Information view. The Configurator GUI queries this endpoint at launch, and uses the model data to populate its user interface.

The second endpoint, "/configuration-validation", concerns validation naturally. It expects a Scenario Configuration as a JSON string as input, and returns a JSON object containing message "VALID" on successful validation, or a list containing errors and inconsistencies met otherwise. The validation process itself checks whether all **Mappings** of the selected Capability Configuration are included in the **Robot Template** chosen. If the validation fails, the error returned lists what **Mappings** are missing from the **Robot Template**.

Through these endpoints, the Configurator API component both contains and supplies the models for creating configurations, and also enables reasoning about the internal consistency of created configurations.

Configurator GUI

As the frontend of the configurator, Configurator GUI is a browser-based Vue.js GUI application that provides various views and menus for the user to create a complete configuration with.

All data used by Configurator GUI is received by querying the "/available-variables" REST endpoint of Configurator API on startup. If the query fails, Configurator GUI will keep querying at specific intervals until it succeeds.

To make the user experience more intuitive, the functionality of Configurator GUI is separated into three primary Vue views: CAPABILITY CONFIGURATION, SCENARIO CONFIGURATION, and CONNECTED ROBOTS.

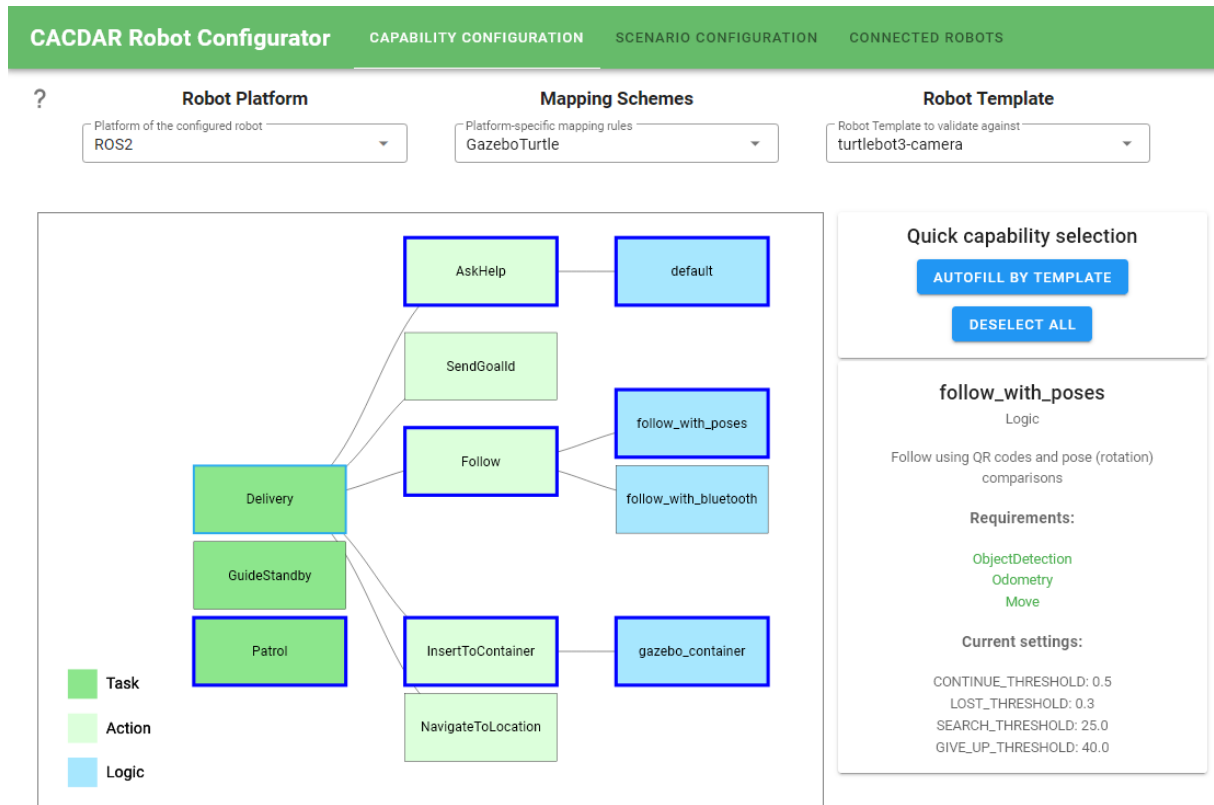

Figure 5.3: CAPABILITY CONFIGURATION allows selecting the capabilities of a robot. Pictured: The follow_with_poses **Logic** of Follow **Action** has just been selected.

The CAPABILITY CONFIGURATION view (Figure 5.3) enables the user to create Capability Configurations by selecting the desired capabilities, meaning **Tasks**, **Actions**, **Logics** and **Mappings**, with the help of a D3 (Bostock, 2012) Scalable Vector Graphics tree.

In particular, a **Robot Template** to validate against can also be selected. If a **Robot Template** is chosen, the Capability Configuration must be successfully validated against the **Robot Template** before the Capability Configuration can be saved.

When validation is initialized, Configurator GUI sends the Capability Configuration to

the "/configuration-validation" endpoint of CONFIGURATOR API. If validation fails, error messages are shown on what **Mappings** of the Capability Configuration are not supported by the selected **Robot Template**, and the user can edit the Capability Configuration accordingly.

Created Capability Configurations can be saved and downloaded as JSON files. At least one saved Capability Configuration is needed for creating a Scenario Configuration, which is in turn necessary for launching a connected robot.

Figure 5.4: SCENARIO CONFIGURATION allows setting various scenario-specific parameters. Pictured: Camera-equipped TurtleBot3 being configured to do Delivery **Task** in GazeboDemo1.

The SCENARIO CONFIGURATION view (Figure 5.4) enables the user to set various

information specific to the scenario the robot is meant to be run in, as an extra layer on top of a saved Capability Configuration.

There are two types of settings to configure: Robot settings and Scenario settings. Robot settings can contain settings specific to a **Robot Template** if one is implemented by the Capability Configuration chosen, and also for **Logics** in the Capability Configuration if they contain settings to adjust. Scenario settings lets the user set Goals to be done in the scenario, and also other initial knowledge through a free-form JSON field that can optionally be filled using `Environment Knowledge Sets`.

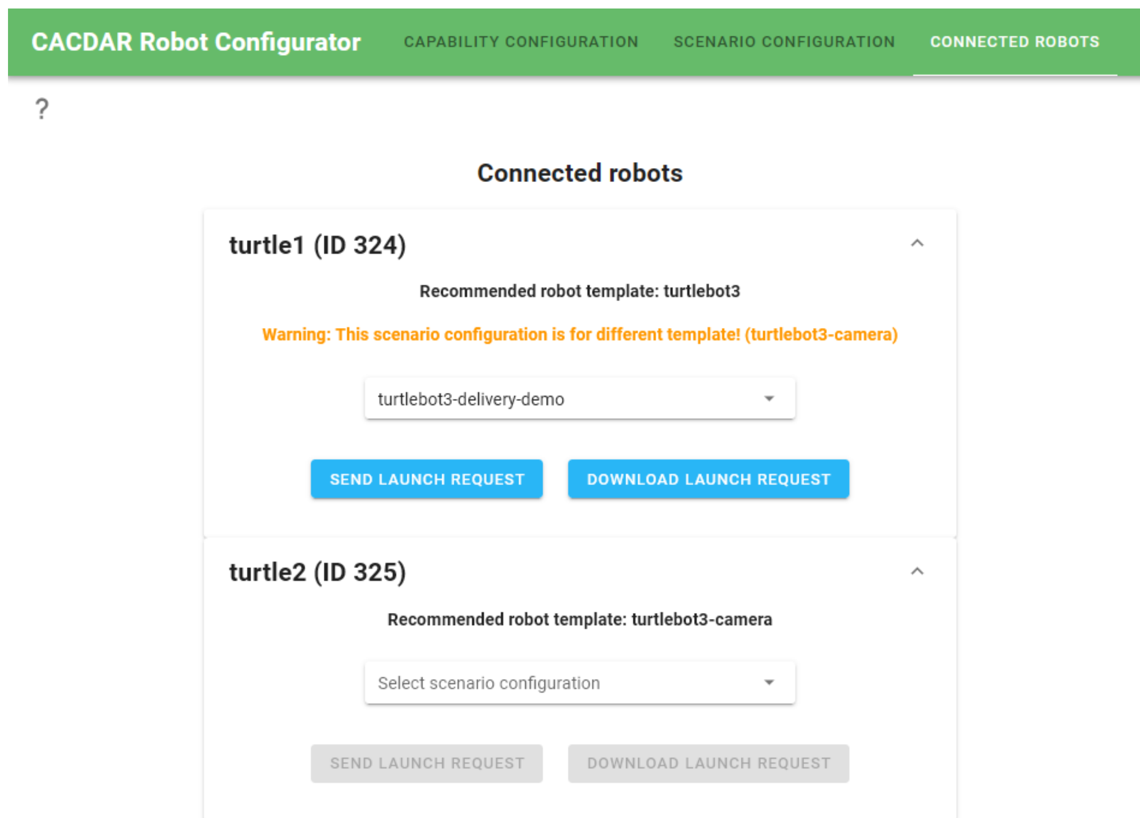

Figure 5.5: CONNECTED ROBOTS shows all robots standby for a Launch Request.

Lastly, the CONNECTED ROBOTS view (Figure 5.5) shows all robots currently connected to the LAUNCH SERVER. Each connected robot describes some of its properties: an ID, a name, and the name of a recommended **Robot Template**. A Launch Request, which contains a Scenario Configuration, can be sent to any connected robot as long as there is a Scenario Configuration to include. However, a warning message is shown if the **Robot Template** the Scenario Configuration selected implements differs from that of the recommended one.

Through these views, the CONFIGURATOR GUI both provides a user interface for creating configurations, and also acts as a hub for launching robots' Cooperative Brain Services.

Launch Server

The LAUNCH SERVER is a Socket.IO application, serving as the middleman between the Configuration GUI and LAUNCH CLIENTs running on the robots. As this middleman component is not really significant for the overarching functionality of the program, and other views will cover its implementation details. Nonetheless, it does have its function in creating the CONNECTED ROBOTS hub for launching Cooperative Brain Services.

Launch Client

The LAUNCH CLIENT is the robot side Socket.IO client. On startup, LAUNCH CLIENT loads a Robot Description chosen. When LAUNCH CLIENT connects to LAUNCH SERVER through Socket.IO, it sends its Robot Description parameters at the same time. LAUNCH CLIENT also has the role of launching its respective Cooperative Brain Service, using a Launch Request sent from CONFIGURATOR GUI through LAUNCH SERVER.

5.3.2 User workflow

The purpose of *Robot Configurator* is to enable the user to create complete Cooperative Brain Service configurations, namely Scenario Configurations, from scratch in a guided manner. All user input, through the entire basic workflow process of creating a configuration and launching a robot, happens strictly within the CONFIGURATOR GUI. Yet in terms of functionality, the process encompasses the whole configurator. By walking through the user actions in order, I can explain the functions happening at the same time. However, it should be noted that several elements of functionality described are tied to the models of variability in Cooperative Brain Service, explained in Chapter 4. Figure 5.6 also visualizes what happens in the different *Robot Configurator* components in the user input process.

Let our initial situation be one where all four *Robot Configurator* components have been successfully launched. The user launches the CONFIGURATOR GUI on their web browser, which first fetches the models of variability from CONFIGURATOR API. Now, to launch a robot, a complete Scenario Configuration is needed to tell Cooperative Brain Service

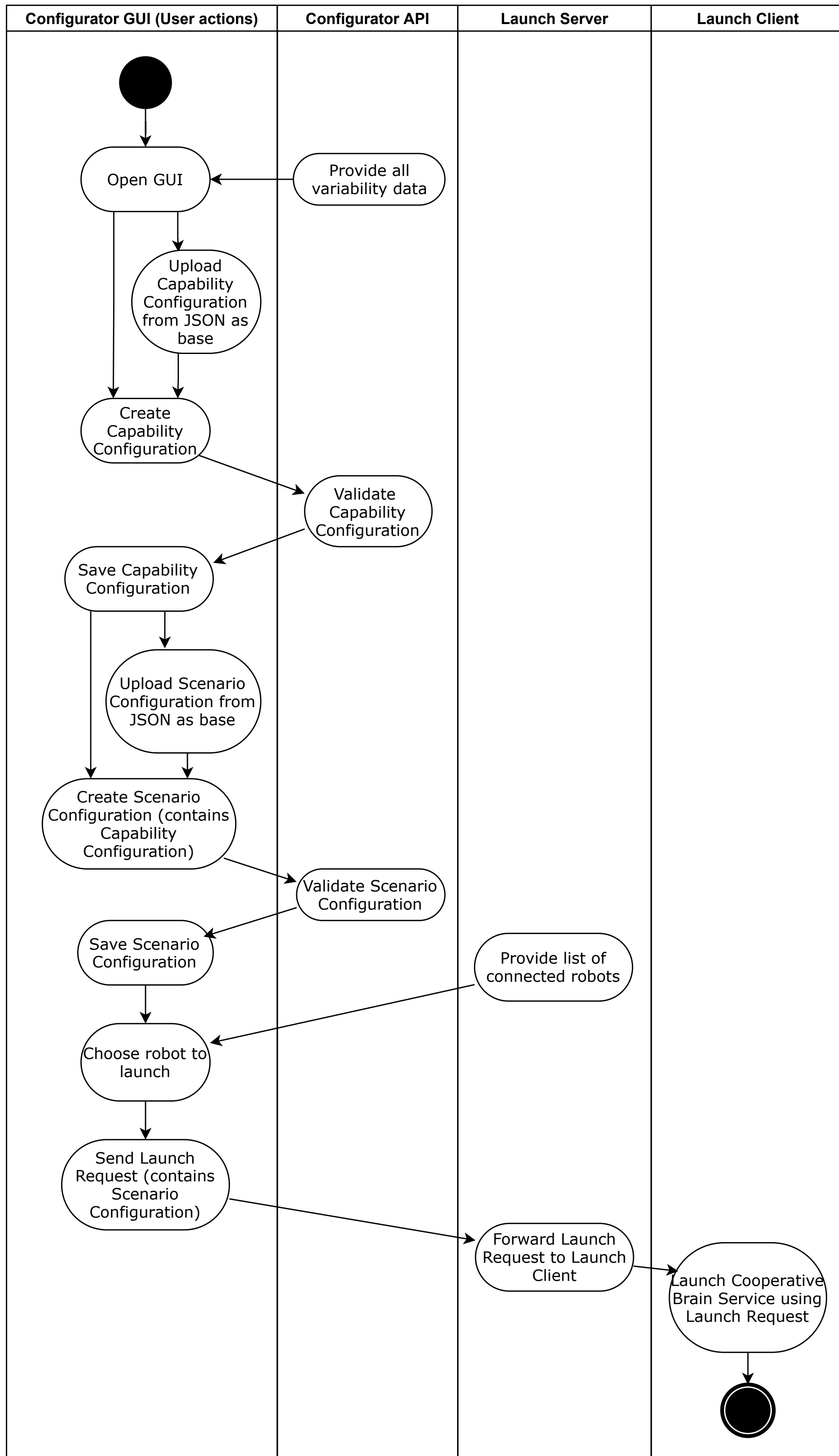

Figure 5.6: Activity in *Robot Configurator* components during CONFIGURATOR GUI user workflow.

what capabilities and knowledge we want to initialize the robot with. But before we can create a Scenario Configuration, we need to first create and save at least one Capability Configuration in the CAPABILITY CONFIGURATION view of Configurator GUI.

In the CAPABILITY CONFIGURATION view (Figure 5.3), the first steps in creating a Capability Configuration are choosing a **Robot Platform** and a **Mapping Scheme** specific to the **Robot Platform**. After this, an interactive Scalable Vector Graphics tree appears that allows selecting what capabilities to include in the Capability Configuration.

The capability tree includes **Tasks**, **Actions** and **Logics** fetched from the Configurator API, based on what **Actions** belong to the current **Robot Platform**. Importantly, only **Logics** can be directly toggled as included or excluded by the user. In turn, selecting a **Logic** also highlights the **Action** it implements. When all **Actions** pertaining to a **Task** are highlighted, the **Task** also becomes highlighted, signaling its inclusion in the current Capability Configuration.

Optionally, a **Robot Template** can be chosen to be implemented by the Capability Configuration. However, this means that the Capability Configuration cannot be saved until it is validated successfully in Configurator API against the **Robot Template**. Validation checks if the **Robot Template** contains all **Mappings** required by the Capability Configuration. If the validation is unsuccessful, Configurator API returns error messages that are shown to the user, so they can adjust the Capability Configuration accordingly.

Saving Capability Configurations adds them to a list within the Configurator GUI application. This list only persist for as long as the application is open, and there is currently no support for storing Capability Configurations on the server side. To alleviate this limitation, saved configurations can be downloaded as JSON files from the Configurator GUI. When a saved configuration JSON is uploaded to Configurator GUI, the fields are populated accordingly. Additionally, the CAPABILITY CONFIGURATION view has an alternate, **Robot Template** based method for quick population. If a **Robot Template** is chosen, the AUTOFILL BY TEMPLATE button becomes available. Clicking it causes all of the **Logics** that have their **Mapping** requirements fulfilled by the **Mapping** list of the **Robot Template** be automatically chosen in the capability tree.

With at least one Capability Configuration saved, the user can move to the SCENARIO CONFIGURATION view (Figure 5.4) next. At bare minimum, this view requires the user to select a saved Capability Configuration to include. However, if the intention is to launch the robot with any scenario-specific settings and goals, the user is likely interested

in setting those with the help of this view. The settings come in two categories: Robot settings and Scenario settings.

The Robot settings originate from the chosen Capability Configuration. If the Capability Configuration chosen implements a **Robot Template**, settings specific to that robot can be adjusted. In case of Turtlebot3, its move speed can be adjusted, for example. Moreover, **Logics** included in the Capability Configuration can also contain settings to adjust; the robot may use a custom move speed during the follow_with_poses **Logic**, which can be adjusted here in Robot settings.

Scenario settings lets the user set goals and also other initial knowledge for use in **Logics**. To set Scenario settings, the user may first select an Environment Name. Selecting an Environment Name allows adding `Environment Knowledge Sets` from the corresponding Environment Model to the robot's initial knowledge of the environment. Also, any number of so-called *Goal Tasks* can be added to the Scenario Configuration. A Goal Task consist of a **Task** and optionally a Location. Their purpose is to define the initial **Tasks** a robot is meant to do. The **Task** for Goal Task can be chosen from among any of those in the CONFIGURATOR API data, but choosing one that is not supported by the current Capability Configuration will cause validation to fail. A Location can also be chosen, but only if an Environment is selected, as they are fetched from the `Ontology Objects` of the Environment Model. The Location is meant to be interpreted by the decision making components of Cooperative Brain Service. Moreover, the Goal Tasks are order sensitive: the robot attempts to carry them out in the order they are given.

In reality, the Goal Task is a simplification instead of a real format used: in the inner logic of CONFIGURATOR GUI they are mapped to *Goal Configurations*, a different format, when saving the Scenario Configuration. The Goal Configuration format is briefly introduced in Section 5.4.3.

With a Scenario Configuration created, a robot can finally be launched. The CONNECTED ROBOTS view shows all robots connected to LAUNCH SERVER through a LAUNCH CLIENT. Any connecting or disconnecting robots appear or disappear respectively in real time. Connecting robots provide three descriptive parameters: Robot ID, Robot Name and Recommended Robot Template. Of these, Recommended Robot Template has a specific purpose in the CONFIGURATOR GUI: a warning text appears if the user chooses a Scenario Configuration whose Capability Configuration implements a **Robot Template** different from the recommended one. When a robot is chosen for launch, a Launch Request is created, which contains the robot's Robot ID and Robot Name along

with the Scenario Configuration. The Launch Request is then sent to the robot's Launch Client through Launch Server. The robot's Launch Client then launches the Co-operative Brain Service of the robot with the contents of Launch Request as parameters.

5.4 Information view

The Information viewpoint (Rozanski and Woods, 2012, p. 302) concerns manipulation and storage of information within the architecture. Considering how the variability models are tied to this configurator's design and configuration formats affect directly the data being produced, this viewpoint is of use to stakeholders interested in this software beyond raw functionality.

All data models in this architecture use JSON format. Notably, no database whatsoever is used in any of the components. The Configurator API models are instead saved as JSON files in the program's folder structure, but this is a concern to be covered in Deployment view (Section 5.6) instead.

Models of *Robot Configurator* can be designated under two fundamentally differing primary categories: models of variability that are provided by Configurator API, and configurations created in Configurator GUI. However, there is also Robot Description, which does not quite fall under either. I will mostly focus on the formats of created configurations here, as they are not covered elsewhere.

5.4.1 Models of Variability

The models of variability are described already in Chapter 4. The concrete formats of the models, meant to cover all necessary variability, can be seen in Figure 5.7. Figure 5.8 shows the format Configurator GUI receives the variability models in.

5.4.2 Robot Description

A Robot Description JSON is provided to Launch Client on startup. It describes the Robot ID, Robot Name and Recommended Robot Template of a robot. The purpose is to show this information in the Configurator GUI. The robot's Robot ID and Robot Name fields are also used in forming a Launch Request together with a Scenario Configuration.

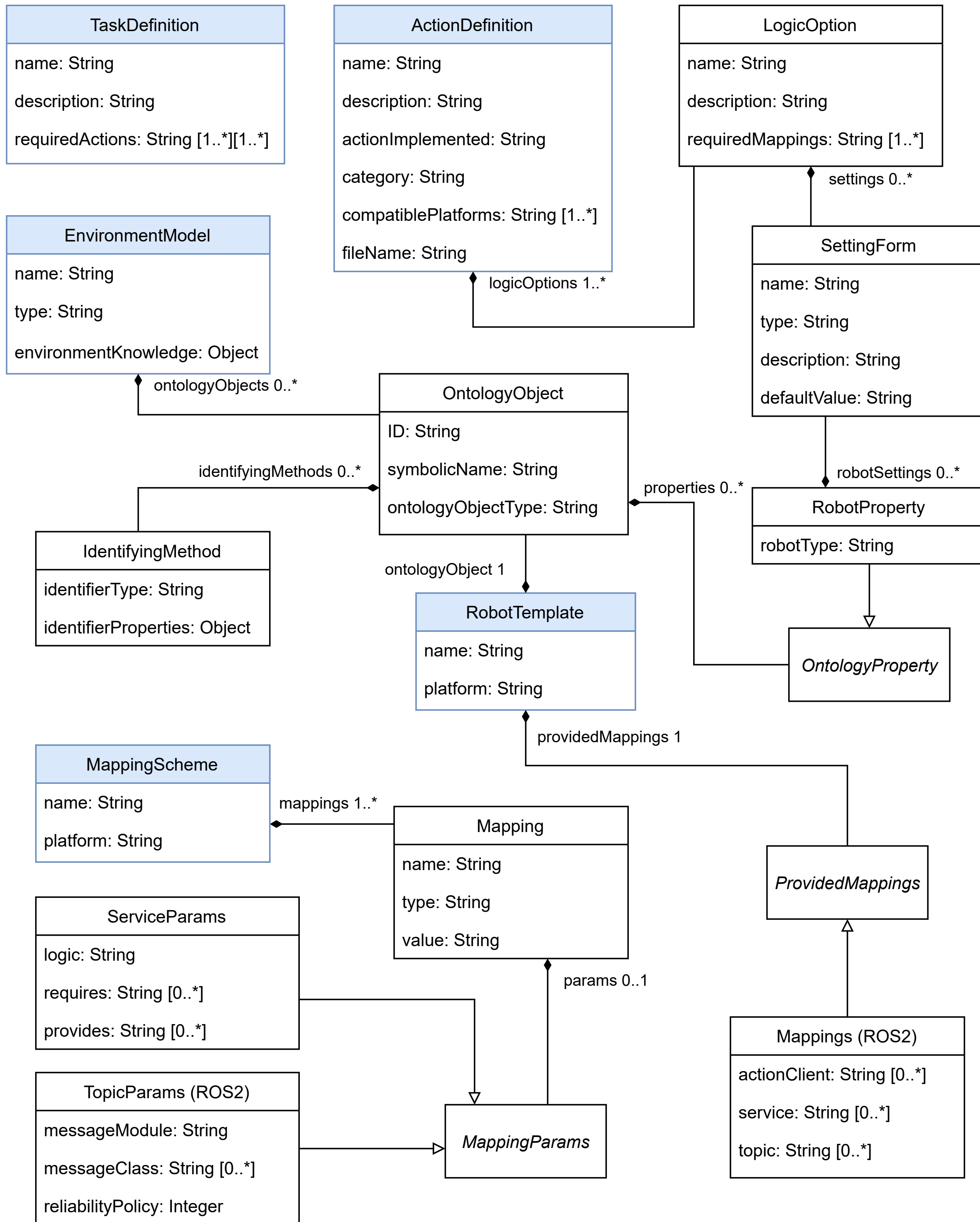

Figure 5.7: Formats of models contained in CONFIGURATOR API. Highlighted in blue are the root elements, returned by a GET endpoint in the format shown in Figure 5.8.

ModelData
platforms: String [1..*]
mappingSchemes: MappingScheme [1..*]
robotTemplates : RobotTemplate [0..*]
taskDefinitions : TaskDefinition [0..*]
actionDefinitions : ActionDefinition [1..*]
environmentModels : EnvironmentModel [0..*]

Figure 5.8: Data format returned by the "/available-variables" GET endpoint of CONFIGURATOR API.

5.4.3 Configuration Formats

As opposed to the models of variability contained in CONFIGURATOR API, these models are the actual formats of configurations produced in CONFIGURATOR GUI through user input. Figure 5.9 shows the structures and relations of these formats.

Capability Configuration format

A Capability Configuration is created in the CAPABILITY CONFIGURATION view of the CONFIGURATOR GUI, and having at least one Capability Configuration is a prerequisite to creating a Scenario Configuration. A Capability Configuration has a Name for the sake of separating it from other saved configurations in a user-controlled manner. A Platform Name is also included to tell Task Runtime which **Robot Platform** module to load when initiating the robot.

The Robot Template Name is included for the purpose of validation. When the configuration is sent to the CONFIGURATOR API for validation, this field determines the **Robot Template** to be validated against. If no Robot Template Name is provided, validation will be omitted. In other words, if a Capability Configuration has a Robot Template Name, it means the configuration promises to successfully implement that corresponding **Robot Template**.

The Mapping Type field is used solely to keep track of which **Mapping Scheme** to choose if the Capability Configuration JSON is uploaded to the CONFIGURATOR GUI. Lastly, **Tasks**, **Actions**, **Logics** and **Mappings** are included as they are chosen by the user in the tree element of the CAPABILITY CONFIGURATION view.

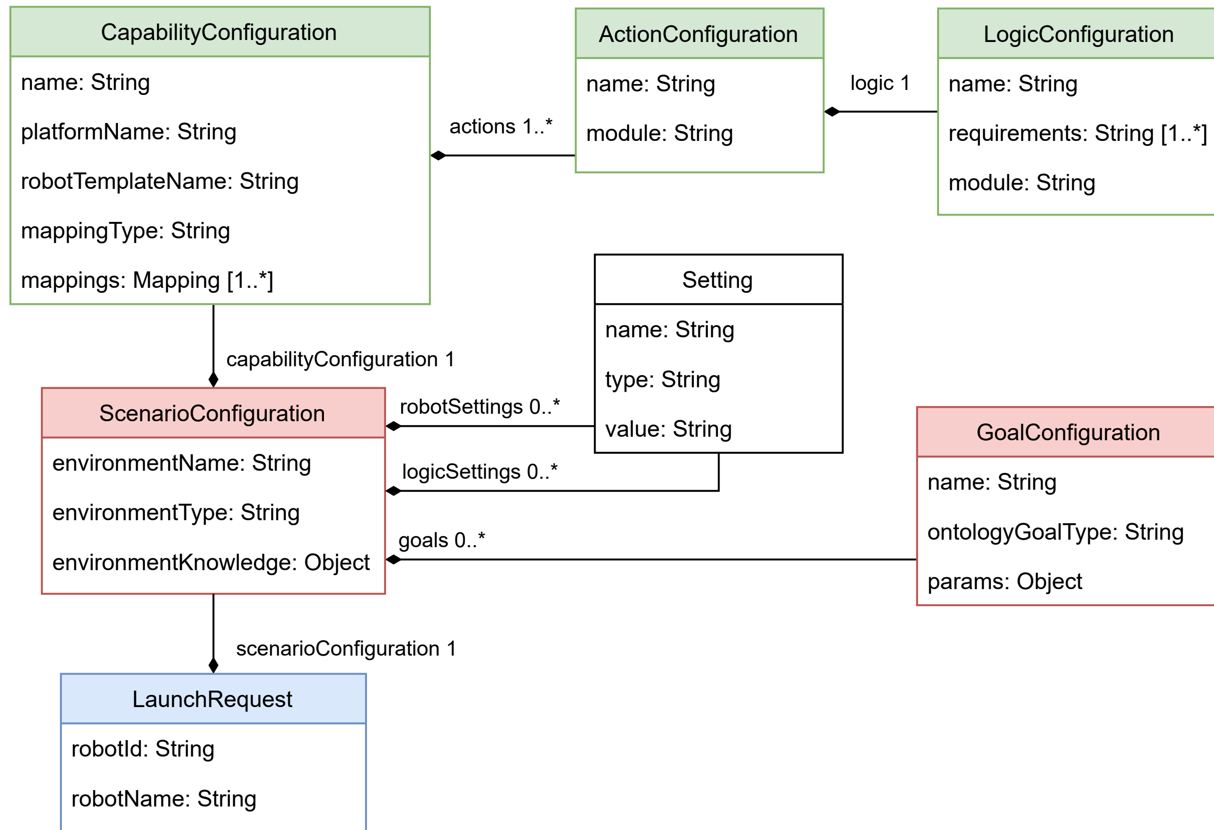

Figure 5.9: The formats of the configuration files created in Configurator GUI, color coded for visual clarity between concerns of the different formats.

Scenario Configuration format

Whereas Capability Configuration defines strictly capabilities of a robot, Scenario Configuration adds a layer of environment- and scenario-specific information on top of a Capability Configuration. As a valid Scenario Configuration contains everything there is necessary to configure, it is almost ready to be used in launching a robot.

The fields specific to Scenario Configuration are Environment Name, Environment Type, Robot Settings, Logic Settings, Environment Knowledge, and Goals.

The Environment Name parameter is included only to make it possible for Configurator GUI logic to choose the correct Environment when uploading Scenario Configuration JSON in the Configurator GUI.

The Environment Type parameter, in contrast, has distinctive use in Task Runtime for handling nonstandard occasions in the initialization of capabilities. For example, specific to ROS2, having an Environment Type "Gazebo" will result in a "namespace" being added

to certain ROS2 Topics as a necessary special measure. This type of behavior is implemented entirely within a Platform Module, but should be avoided if possible due to being hard to follow in terms of program logic.

Robot Settings are settings related directly to the robot's own functionality, such as movement speed. Logic Settings are settings specific to the **Logics** of the Capability Configuration contained in the Scenario Configuration.

Environment Knowledge is a JSON object of to be provided to the robot's knowledge base in initialization, crucial for setting up demo scenarios. Environment Knowledge can be written manually as JSON in the GUI, or appended to from `Environment Knowledge Sets` of Environment Models.

Goals is a list of *Goal Configurations*, a format that is specific to Cooperation Ontology and not utilized in any *Robot Configurator* logic. In effect, Goal Configurations contain the initial **Tasks** accompanied with Location information for the Planner component of Cooperative Brain Service to interpret on startup. Goal Tasks created in the SCENARIO CONFIGURATION view (covered in Section 5.3.2) are mapped into this format when saving the Scenario Configuration.

Launch Request

A Launch Request is the ultimate end product of the *Robot Configurator*, containing all settings necessary to initialize a robot's Cooperative Brain Service. It only adds fields Robot ID and a Robot Name on top of a Scenario Configuration. The Robot ID is what is used to differentiate robots in the cooperation ontology, while the Robot Name is included mainly for the convenience of the user.

5.5 Development view

The Development viewpoint (Rozanski and Woods, 2012, p. 366) emphasizes that documenting the development environments is generally important for the upkeep of a system. *Robot Configurator* features multiple components of various concerns and technologies, and CONFIGURATOR GUI particularly has a somewhat sizeable codebase. For these reasons, some degree of explanation is certainly warranted in case of further development efforts by other developers. The languages used are JavaScript for the three server-side components, and Python3 for the LAUNCH CLIENTS on robots. Technologies used include

Vue.js (You, 2021) and Socket.io (Rauch, 2021). I will describe the components one by one, as each one uses its own particular technologies.

5.5.1 Configurator API

Created in plain JavaScript, Configurator API provides REST endpoints for Configurator GUI to query. No JavaScript web application framework was used, as the sparse amount of REST endpoints did not to necessitate it.

The folder structure of Configurator API contains the various models of variability as JSON files, which are loaded into Configurator API on startup. This is discussed in the Deployment view (Section 5.6) instead.

5.5.2 Configurator GUI

The Configurator GUI is implemented as a browser-based Vue.js (You, 2021) application. Vuetify.js (Leider and Leider, 2021), a Material Design framework, is used for UI elements and design. D3 (Bostock, 2012), a data manipulation library, is used in tandem with Scalable Vector Graphics to create the tree interface found in the Capability Configuration view. As per the design principles of Vue, the page consists of Vue components nested in different ways. Each component consists of its own HTML, methods and data fields. Data fields can be passed to other components, but mutating them anywhere except in the owner component is against Vue's principles.

For some component files with extensive amounts of methods, the methods are put into separate implementation files categorized under the views they are responsible for.

5.5.3 Launch Server

The server that acts as a middleman between the Configurator GUI and Launch Clients, made using Socket.IO for JavaScript. It is essentially a very basic Socket.IO implementation, to the point of not warranting any detailed technical description beyond the basics described in the Socket.IO documentation (Rauch, 2021). The main point of note is that the Configurator GUI connects to the server using a different endpoint than the Launch Clients.

5.5.4 Launch Client

Launch Client is a basic Python3 Socket.IO client that connects to the Launch Server. Target server IP address and Robot Description folder are read from a configuration file, making it easy to change them as necessary. Upon receiving a Launch Request, the Launch Client runs the main method of Cooperative Brain Service, also a Python3 program, as a Python import.

5.6 Deployment view

The Deployment viewpoint (Rozanski and Woods, 2012, p. 380) is concerned with the physical and technical circumstances involved in successfully deploying and maintaining the software system. An applying stakeholder would be anyone who intends to set up *Robot Configurator* by themselves or modify the model data contained, instead of only accessing an already running Configurator GUI instance. Especially because the model data is to be changed as the modules implemented in Task Runtime change, this view can be useful to anyone doing research in CACDAR.

While *Robot Configurator* itself does not have noteworthy performance requirements by modern-day hardware standards, it does involve running multiple separate software components over different machines. These components also require certain technologies and dependencies to be installed, so there is a plenty of incentive to include this type of view description. Also, adding new data to Configurator API will be covered in this view, which in particular is a concern of rather great importance.

As illustrated in Figure 5.10, deploying the entire *Robot Configurator* involves launching three components, Configurator API, Configurator GUI, and Launch Server, as separate processes on a server system. The components can be launched in any order. Each robot intending to use *Robot Configurator* is to run its own instance of Launch Client, which in turn launches a Cooperative Brain Service instance when provided a valid configuration.

Additionally, being a web service, Configurator GUI has to be used through a web browser with JavaScript support. Though the user PC is pictured as a separate entity, Configurator GUI can naturally be accessed on the server machine itself should it support graphical web browsing.

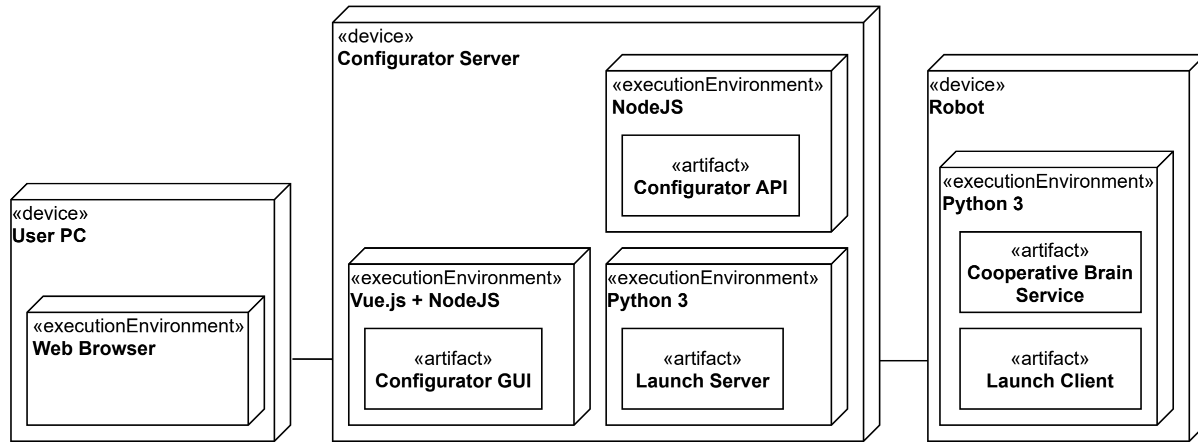

Figure 5.10: How the components of *Robot Configurator* are to be deployed on the different devices.

5.6.1 Server-side deployment

The server system must have a working installations of JavaScript and Vue.js (You, 2021). The Configurator API, Configurator GUI, and Launch Server all have to be launched separately, e.g. in separate terminal windows. The READMEs of each component contain the specific launch commands used for this.

A major deployment concern of this system lies in the models of variability included in Configurator API, described in Chapter 4. Only on startup, Configurator API loads its model data from JSON files, and this is currently the only way to provide them. Therefore, adding or editing models requires access to these files. Moreover, be it a robot template or **Action** definition, it is necessary to know what the JSON format is like and where the files are to be placed.

The files are stored in a folder structure that mimics that of the models seen in Chapter 4. E.g. the "platforms" folder contains its own folder for each **Robot Platform**, ("ROS1" and "ROS2"), which in turn contain subfolders for "robot-templates" and "mapping-schemes". These subfolders contain the JSON files defining **Robot Templates** and **Mapping Schemes** respectively, particular to that platform.

Adding a new **Action** is a matter of creating a definition file for it in the "action_definitions" folder. **Tasks**, on the other hand, have to be added separately to "task_definitions" folder. The definitions are made separate because there is no direct correlation between the two: a **Task** is implemented by a list of **Actions**, but the same **Action** can be used in multiple **Tasks**.

An **Action** definition defines which **Robot Platforms** it supports, and also contains

definitions for all **Logics** it is implemented by. In other words, a **Logic** definition is included in the definition file of the **Action** it belongs to. A **Logic** definition mainly lists the required **Mappings**, and optionally also contains **Logic Settings**. These **Logic Settings** can be adjusted when doing Capability Configuration in the CONFIGURATOR GUI.

For the exact formats of the **Task** and **Action** definition files, Figure 5.7 can be used as a reference. As the figure shows, **Task** and **Action** definitions are kept separate from each other.

To illustrate better how these **Capability** definitions are added, let us assume a user trying to add a new **Action** for opening doors, named OpenDoor. To add the **Action**, the user simply has to create a JSON definition file for it in the "action_definitions" folder. However, the **Action** definition must also contain at least one **Logic** definition. If we imagine an environment with automatic doors that can be opened by sending signals, an appropriate **Logic** could be "open_by_sending_signal". This **Logic** definition then would in turn define whatever **Mappings** are needed to make the robot send the signal to the door.

The OpenDoor **Action** and its **Logic** were added successfully as a result. However, they are not of much use if there are no **Tasks** that would use the OpenDoor **Action**. Let us suppose there is an existing **Task** definition file in "task_definitions" folder for Patrol **Task**, and that this **Task** has previously been defined to be implemented only using **Action** NavigateToLocation. We may now add a new alternative list of required **Actions** that consist of both OpenDoor and NavigateToLocation. As a result, it becomes possible to have Patrol **Tasks** where the robot may also open doors on its patrol route.

It should also be briefly mentioned how Socket.IO is used for communication between CONFIGURATOR GUI and LAUNCH SERVER. CONFIGURATOR GUI first connects to LAUNCH SERVER, and then waits for messages through two Socket.IO endpoints: "robotConnected" and "robotDisconnected". Any LAUNCH CLIENTS that connect to LAUNCH SERVER send a Robot Description, which contains data identifying the robot. LAUNCH SERVER then forwards this data to CONFIGURATOR GUI through the "robotConnected" socket. If any LAUNCH CLIENTS disconnect from LAUNCH SERVER, LAUNCH SERVER informs CONFIGURATOR GUI of this through the "robotDisconnected" socket.

5.6.2 Robot-side deployment

Each robot runs its own Launch Client. Though it is not run on the configurator server, we consider Launch Client a part of *Robot Configurator* due to it being entirely tied to its functionality.

Launch Client itself uses Cooperative Brain Service as a Python import, which is why the Launch Client is typically put in the Cooperative Brain Service folder. A Python 3 installation of version 3.7 or above is a necessity to run both the Launch Client and Cooperative Brain Service.

Upon running, Launch Client tries to read a "client-settings.json" file from its folder. This file informs the Launch Client of the server IP to connect to, and the folder name for Robot Descriptions. The Launch Client also needs a name of a Robot Description as a launch parameter, which it searches for in the folder defined in client settings. By default, this folder is named "robot-settings". If a valid Robot Description is found, Launch Client attempts to connect to Launch Server using the server IP in the client settings file.

Though it is not directly a concern of this *Robot Configurator*, it should be mentioned that on the Cooperative Brain Service side, support for the platform of the robot must be implemented within the Task Runtime component as **Robot Platform** and **Action** modules.

5.6.3 Client-side requirements

As Configurator GUI is a web service, it can be simply connected to through network access. However, using it does require a graphical web browser modern enough to run Vue.js and Scalable Vector Graphics manipulated through D3 (Bostock, 2012). Otherwise, no guarantees can be made about the page working properly. Also, Configurator GUI most likely will not work on mobile platforms, as it is not tested on those nor is it developed with such platforms in mind.

6 Discussion

As explicated in Chapter 2, the purpose of this thesis is to create and document *Robot Configurator*, a system for streamlining the use of Cooperative Brain Service by providing an intuitive way to configure its initialization parameters. Let us first reflect on how the designs documented in this paper answer the research questions set in Section 2.3.

RQ1: How can the configurator tool streamline Cooperative Brain Service configuration for the user? For this purpose, I have specifically included a Graphical User Interface component in *Robot Configurator*, as Configurator GUI. Configurator GUI features views that guide the user in creating both Capability Configurations and Scenario Configurations, and it also allows launching robots with the created configurations from a hub-like view. Section 5.3.1 gives a detailed explanation of the Configurator GUI, and Section 5.3.2 explains the basic workflow from the perspective of a user. Nevertheless, it is still possible that the user interface or even some non-critical functionality of Configurator GUI could have room for improvement, as development was done with priority on implementing functionality necessary for the basic workflow first. However, I believe further Configurator GUI development is better done after getting more feedback of it in use, separate from this thesis.

RQ2: How do we validate variability configurations made? The validation in *Robot Configurator* concerns whether a robot is actually capable of performing the **Actions** it is designated to have in a Capability Configuration. **Robot Templates** are introduced to represent robots' capabilities as lists of **Mappings**. Validating whether a type of robot can run a Capability Configuration is now rather straightforward, as long as we have a **Robot Template** representing the robot: the validation checks whether all of the **Mappings** required by the Capability Configuration are included in the **Robot Template**. Sections 5.3.1 and 4.1.3 concern this topic.

RQ3: How do we comprehensively document the architecture of *Robot Configurator*? By "comprehensively" I refer to how the documentation should cover not just bare functionality, but also how the software can be maintained and developed further. For this purpose,

I have chosen to use a viewpoint-based architecture documentation framework (Rozanski and Woods, 2012), where the viewpoints correspond to different types of concerns, maintenance (as deployment) and development included. This architecture description of *Robot Configurator* is presented in Chapter 5.

Though it is always possible further requirements will come up as the *Robot Configurator* is put to use, all the current documented requirements were answered as described in this thesis. Therefore, it could be said *Robot Configurator* meets the requirements set to it at the time being. That being said, though *Robot Configurator* has been evaluated to work as intended in manual use, as per the basic workflow described in Section 5.3.2, it has not been formally tested in any way. This is because it was not seen as necessary at the time. If there is interest, further work could involve creating test cases from stakeholder requirements, and using those tests to evaluate that the system is working as intended.

One point that also warrants discussion is the development of the CONFIGURATOR GUI frontend. Creating the CONFIGURATOR GUI ended up being an undertaking of unexpected scale and complexity, not helped by the lack of previous frontend development experience. In short, getting all the GUI views implemented required a significant work effort for what might not be seen as the most compelling part of this thesis research. The initial lack of know-how also led to some difficulty in creating a sensible code structure for the CONFIGURATOR GUI, especially in terms of the data models and flow. However, at least this experience will surely be beneficial should an opportunity arise to work with an another Vue.js project in the future.

Finally, it must be conceded that though a degree of design knowledge was documented, the problem domain itself of this thesis, deployment parameters of autonomous cooperative robots, is rather niche. Accordingly, the research focus was on answering stakeholder concerns over trying to present generalisable scientific contribution. Even so, Chapter 3 is dedicated to discussing the design of *Robot Configurator*, and the models of variability described in Chapter 4 are essentially a novel approach to modeling properties of robots and environments from the cooperation-oriented perspective of Cooperative Brain Service.

7 Conclusions

The *Robot Configurator* was implemented as defined by the requirements, and is already integrated for use with Cooperative Brain Service. In the primary research of CACDAR, it had become apparent that there are many elements particularly in simulated 3D worlds and the real world that can cause experimenting with robots to be challenging (Mäkitalo et al., 2021). Therefore, it is greatly beneficial to have a tool that can streamline setting up various experiments. The new models introduced can potentially also be used as-is or as inspiration for other elements of the whole CACDAR research project, or even other future robot research.

As for what direction the *Robot Configurator* could go next, one idea would be that of making the Cooperative Brain Service support changing configurations in real time, as opposed to the current approach which determines the modules loaded at startup. In fact, *Robot Configurator* being hosted as a service already hints towards this sort of real-time approach. As for how to actually implement this, it could either be as simple as changing only the parameters specific to Scenario Configuration, which would be comparatively straightforward, or changing even the Capability Configuration and thus swapping the **Logic** modules, which would require careful procedures to make this work during runtime. Either way, most of the work in doing these changes would be on the Cooperative Brain Service side. From *Robot Configurator*'s perspective, the way configurations are sent could stay practically the same. As a somewhat tangential idea, the CONNECTED ROBOTS view could be updated to show more about robots' current statuses.

An another relevant idea for future research could be developing further the models of variability featured in Chapter 4. In particular, they could be applied to some existing software variability modeling methodology such as Kumbang (Asikainen et al., 2007). However, this proposal would appear more acute if the complexity of the configuration problem were to increase in some manner first.

In my view, however, currently the best next step for *Robot Configurator* would be to put it into proper practical use in the experiments of CACDAR, and then improve it based on the feedback received.

Bibliography

Asikainen, T., Mannisto, T., and Soininen, T. (2006). “A unified conceptual foundation for feature modelling”. In: *10th International Software Product Line Conference (SPLC'06)*, pp. 31–40. DOI: 10.1109/SPLINE.2006.1691575.

Asikainen, T., Männistö, T., and Soininen, T. (2007). “Kumbang: A domain ontology for modelling variability in software product families”. English. In: *ADVANCED ENGINEERING INFORMATICS* 21.1, pp. 23–40. ISSN: 1474-0346.

Asikainen, T., Soininen, T., and Männistö, T. (Aug. 2003). “A Koala-Based Ontology for Configurable Software Product Families”. In:

Bachmann, F. and Bass, L. (2001). “Managing variability in software architectures”. In: *ACM SIGSOFT Software Engineering Notes* 26.3, pp. 126–132.

Bosch, J. (2000). *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. USA: ACM Press/Addison-Wesley Publishing Co. ISBN: 0201674947.

Bostock, M. (2012). *D3.js - Data-Driven Documents*. URL: http://d3js.org/.

Brocke, J. v., Hevner, A., and Maedche, A. (Sept. 2020). “Introduction to Design Science Research”. In: pp. 1–13. ISBN: 978-3-030-46780-7. DOI: 10.1007/978-3-030-46781-4_1.

Brownsword, L. and Clements, P. (1996). *A Case Study in Successful Product Line Development*. Tech. rep. CMU/SEI-96-TR-016. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=12587.

Czarnecki, K., Østerbye, K., and Völter, M. (Jan. 2002). “Generative Programming.” In: pp. 15–29.

Fielding, R. T. and Taylor, R. N. (2000). “Architectural Styles and the Design of Network-Based Software Architectures”. AAI9980887. PhD thesis. ISBN: 0599871180.

Galster, M., Männistö, T., Avgeriou, P., and Weyns, D. (June 2011). “First International Workshop on Variability in Software Architecture (VARSA 2011)”. In: *Software Architecture, Working IEEE/IFIP Conference on*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 280–281. DOI: 10.1109/WICSA.2011.44. URL: https://doi.ieeecomputersociety.org/10.1109/WICSA.2011.44.

Gangemi, A. (2017). *DUL: the DOLCE + DnS Ultralite ontology*. URL: http://ontologydesignpatterns.org/wiki/Ontology:DOLCE+DnS_Ultralite (visited on 11/17/2021).

Gates, B. (Feb. 2007). “A Robot in Every Home”. In: *Scientific American* 296, pp. 58–65. DOI: 10.1038/scientificamerican0208-4sp.

Hallsteinsen, S., Hinchey, M., Park, S., and Schmid, K. (2008). “Dynamic Software Product Lines”. In: *Computer* 41.4, pp. 93–95. DOI: 10.1109/MC.2008.123.

Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A. (Jan. 1990). “Feature-Oriented Domain Analysis (FODA) feasibility study”. In:

Leider, J. and Leider, H. (2021). *Vuetify — A Material Design Framework for Vue.js*. URL: https://vuetifyjs.com/en/ (visited on 09/02/2021).

Mäkitalo, N., Linkola, S., Laurinen, T., and Männistö, T. (2021). “Towards Novel and Intentional Cooperation of Diverse Autonomous Robots: An Architectural Approach”. In: *Companion Proceedings of the 15th European Conference on Software Architecture*. URL: http://ceur-ws.org/Vol-2978/casa-paper1.pdf.

Myllärniemi, V., Ylikangas, M., Raatikainen, M., Pääkkö, J., Männistö, T., and Aaltonen, T. (2012). “Configurator-as-a-Service: Tool Support for Deriving Software Architectures at Runtime”. English. In: *International Workshop on Variability in Software Architecture(VARSA), Helsinki, August, 2012*. ACM, pp. 151–158. ISBN: 978-1-4503-1568-5.

Nord, H. and Chambe-Eng, E. (2021). *Qt - Cross-platform software development for embedded & desktop*. URL: https://www.qt.io/ (visited on 10/08/2021).

Peffers, K., Tuunanen, T., Rothenberger, M. A., and Chatterjee, S. (2007). “A Design Science Research Methodology for Information Systems Research”. In: *Journal of Management Information Systems* 24.3, pp. 45–77. DOI: 10.2753/MIS0742-1222240302.

Peltonen, H., Männistö, T., Alho, K., and Sulonen, R. (Jan. 1994). “Product Configurations - An Application for Prototype Object Approach.” In: pp. 513–534.

Pezoa, F., Reutter, J. L., Suarez, F., Ugarte, M., and Vrgoč, D. (2016). “Foundations of JSON schema”. In: *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, pp. 263–273.

Prud’homme, C., Fages, J.-G., and Lorca, X. (2016). *Choco Solver Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. URL: http://www.choco-solver.org.

Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software*.

Rauch, G. (2021). *Socket.IO*. URL: https://socket.io/ (visited on 09/21/2021).

ROS 2 Design (2021). URL: https://design.ros2.org/ (visited on 11/17/2021).

Rozanski, N. and Woods, E. (2012). "Software systems architecture: second edition". In: *ACM SIGSOFT Softw. Eng. Notes* 37.2, p. 36. URL: http://dblp.uni-trier.de/db/journals/sigsoft/sigsoft37.html#Ra12a.

Simons, P., Niemelä, I., and Soininen, T. (2002). *Extending and Implementing the Stable Model Semantics*.

Svahnberg, M., Gurp, J. van, and Bosch, J. (July 2005). "A Taxonomy of Variability Realization Techniques: Research Articles". In: *Softw. Pract. Exper.* 35.8, pp. 705–754. ISSN: 0038-0644.

Tiihonen, J., Raitahila, I., Raatikainen, M., Felfernig, A., and Männistö, T. (Sept. 2018). "Generating Configuration Models from Requirements to Assist in Product Management: Dependency Engine and its Performance Assessment". English. In: *Proceedings of the 20th Configuration Workshop*. Ed. by F. Alexander, J. Tiihonen, L. Hotz, and M. Stettinger. Vol. Vol-2220. CEUR Workshop Proceedings. Germany: CEUR-WS.org, pp. 69–76.

TurtleBot3 Features (2021). URL: https://emanual.robotis.com/docs/en/platform/turtlebot3/features/ (visited on 11/22/2021).

You, E. (2021). *Vue.js - The Progressive JavaScript Framework*. URL: https://vuejs.org/ (visited on 09/21/2021).