Department of Informatics
Technical University of Munich

Master's Thesis in Informatics

Evaluating the Impact of Knowledge Aggregation with Subjective Logic on the Overall Systems Adaptivity in Multi-Agent Self-Adaptive Cyber-Physical Systems

Tamer Temizer

Department of Informatics

Technical University of Munich

Master's Thesis in Informatics

Evaluating the Impact of Knowledge Aggregation with Subjective Logic on the Overall Systems Adaptivity in Multi-Agent Self-Adaptive Cyber-Physical Systems

Evaluierung des Einflusses der Wissensaggregation mit Subjektiver Logik auf die Adaptivitat des Gesamtsystems in Multiagenten Selbstadaptiven Cyber-Physical Systems

Author:	Tamer Temizer
Submission Date:	February 15, 2022
Supervisor:	Prof. Dr. Alexander Pretschner
Advisor:	M.Sc. Ana Petrovska

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Garching, February 15, 2022 Tamer Temizer

Acknowledgments

First and foremost I want to thank my supervisor Ana Petrovska for her constant support and understanding throughout the thesis. Her genuine willingness to assist and guide made this thesis a very pleasant and fulfilling experience. I want to thank Malte Neuss, Sebastian Bergemann and the others that have contributed on laying the groundwork this thesis is built upon. I want to thank Nour Bouzid for her major contributions in the development and debugging of the system.

I also want to thank my family and friends for their continuous support and encouragement. Without their support, this thesis would not be possible. Finally, I want to thank Dr. Hande Bekmez for her love and keeping me sane throughout the thesis.

Abstract

The role of cyber-physical systems (CPSs) in the modern world is immense. As they operate in the physical world, they encounter both internal and external uncertainties constantly. A successful CPS should compensate for these uncertainties to operate in a robust and reliable way. Self-Adaptive Cyber-Physical Systems (SACPS) adapt during run-time and manipulate their internal state to overcome internal and external uncertainties.

Even though the research fields of adaptivity and self-adaptive systems (SASs) are growing rapidly, there is a lack of consistent formal definitions of what system adaptivity means. This gap in the literature led to research that aims to formally define system adaptivity and a model problem that can be used evaluate multi-agent self-adaptive cyber-physical systems (MA-SACPS). The predecessors of this thesis implemented and simulated this model problem. One interesting aspect of this implementation is the Knowledge Aggregation (KA) between the different agents using Subjective Logic (SL). This functionality aims to close another gap in the SAS literature on knowledge and uncertainty representation, and reasoning under uncertainty to update the knowledge during run-time.

Another recent advance in the research fields of adaptivity and SASs is the introduction of an adaptivity metric called the Q-Score that can be used to measure the adaptivity of a system. A predecessor of this thesis implemented a wrapper called the evaluation framework to calculate this adaptivity metric.

This thesis aims to bring together these two components to evaluate the impact of Knowledge Aggregation with Subjective Logic on the overall system's adaptivity in simulated MA-SACPS. This is done by merging the two different subsystems implemented by the predecessors of this thesis. One major contribution of this thesis is creating this merged system. There were several challenges in merging these systems that were solved in the scope of this thesis.

Merging these subsystems allowed for a sophisticated, new experimental tooling system that can automate and improve the experimentation process. The new tooling allows automating the configuration, execution, storage, and evaluation steps and reduces the previously required manual labor greatly. The new tooling also allows distributing the execution of the experiments to greatly reduce the long computational time required to explore the vast parameter space.

After the implementation for these two main contributions were done, the evaluation of KA with SL started. Several experiments were designed to thoroughly evaluate the impact of KA with SL from an adaptivity perspective. We found KA with SL improves the adaptivity and the performance of the system significantly in certain scenarios. These scenarios are presented and validated through experimental results.

Contents

1 Introduction

1.1 Context

Cyber-physical systems (CPS) are becoming more and more impactful in today's world. These systems have to operate in a robust and predictable way when faced with internal or external uncertainties which are plenty in the real world. Internal uncertainties can be malfunctions or unplanned changes in the agent. The external uncertainties can be unforeseeable changes in the environment (context) the CPS operates in. Self-adaptive cyber-physical systems (SACPS) or self-adaptive systems (SAS) can adjust their behaviour in run-time to compensate for these uncertainties. In this work we study benefits and drawbacks of Knowledge Aggregation (KA) with Subjective Logic (SL) for Multi-Agent Self-Adaptive Cyber-Physical Systems (MA-SACPS) from the perspective of system adaptivity.

1.2 Motivation

To understand the motivation for this work, one has to look at the gaps in the state-of-the-art literature on adaptivity and self-adaptive systems. The first gap this work is motivated from, is the fact that there is no consistent definitions of system adaptivity in the literature. Having consistent definitions are a must to discuss whether a system is self-adaptive or not. Petrovska [37] introduces a framework and formalisms to close this gap. This work is built upon that framework and formalisms. The definition of adaptivity that we are using for this thesis can be seen in Section 2.1.

Furthermore, Petrovska [37] proposes a model problem from the CPS domain to explore system adaptivity. The model problem was implemented in the predecessors of this thesis [34, 40]. This implementation includes many configureable internal and external uncertainties. The self-adaptive agents try to compensate for these uncertainties using Knowledge Aggregation (KA) with Subjective Logic(SL). We are using this implementation of the model problem to explore another gap in the SAS literature. This gap is the lack of solutions for knowledge and uncertainty representation in the adaptation logic and reasoning under uncertainties to update the knowledge during run-time. The model problem is explained in detail in Section 3.1.

A Master's thesis by Bergemann [5] implements a wrapper called the evaluation framework around the implementation of the model problem to measure adaptivity using Petrovska's [37] definitions of adaptivity metrics. The details of the previous work can be found in Chapter 3.

The motivation for this thesis is to evaluate the impact of KA with SL in our model problem from an adaptivity perspective by merging the evaluation framework with the existing implementation. This thesis offers the first extensive evaluation of the system from an adaptivity perspective. The previous works and how this thesis fits in the bigger picture can be seen from Figure 1.1.

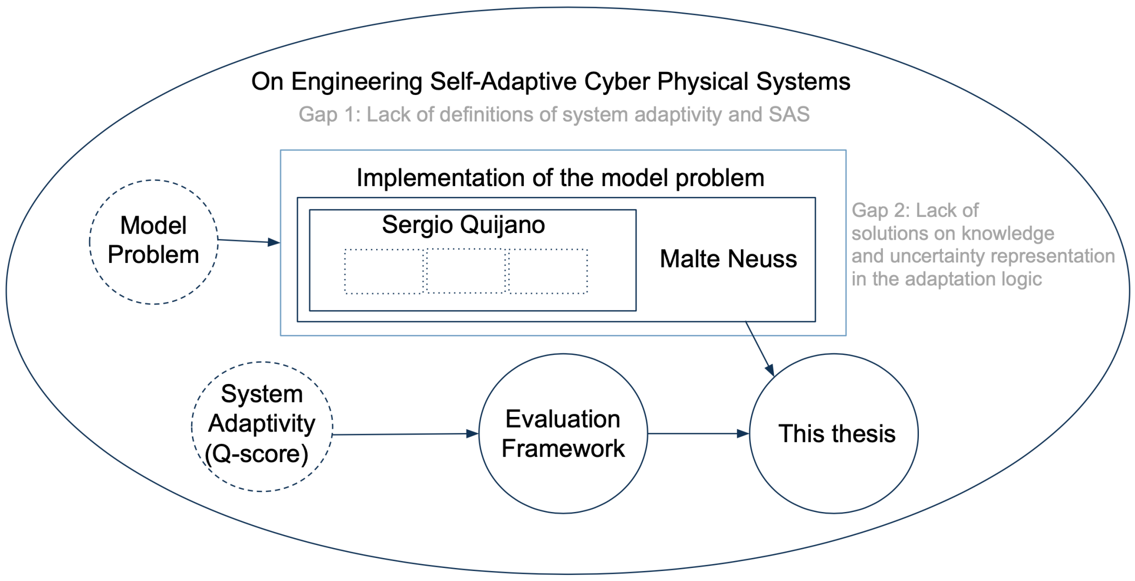

On Engineering Self-Adaptive Cyber Physical Systems
Gap 1: Lack of definitions of system adaptivity and SAS
Implementation of the model problem
Model Problem
Sergio Quijano
Malte Neuss
Gap 2: Lack of solutions on knowledge and uncertainty representation in the adaptation logic
System Adaptivity (Q-score)
Evaluation Framework
This thesis

Figure 1.1: *The figure aims to show how this thesis fits in the bigger picture of closing the gaps in SAS literature. The main overarching theme between all of the predecessors of this thesis is the formalisms of system adaptivity and the model problem proposed by Petrovska [37]. This implementation of the model problem is done by various projects and theses of the same group. In parallel the measure of system adaptivity (Q-score) was proposed by Petrovska [37] and implemented in the evaluation framework by Bergemann [5]. This thesis combines both systems to evaluate the impact of knowledge aggregation with SL on the model problem from a lens of adaptivity.*

1.3 Aims and Contribution

The primary contribution of this thesis is to evaluate the impact of knowledge aggregation with subjective logic from the perspective of self-adaptivity for the ROS-based implementation of multi-agent self-adaptive cyber-physical system. Even though there were several attempts at evaluating the system in the past, this is the first work that evaluates the adaptivity property of the system to this depth.

In order to evaluate the impact of the knowledge aggregation from a lens of adaptivity, the previously created ROS-based implementation of the multi-agent self-adaptive cyber-physical system and the evaluation framework had to be merged. Merging the two subsystems revealed several bugs and limitations which shifted the focus of the thesis to fix those. Merging the subsystems, fixing the bugs and creating a much more robust and reliable system was one of the major contributions of this thesis.

During the scope of this thesis, a completely new experimental tooling system was introduced. This new tooling containerizes the merged system with all of its dependencies to create a stand-alone Docker container. The new tooling automates and simplifies the whole process of parallelization, configuration, execution, debugging, storage and evaluation of the results. This containerization further allows running the simulation in parallel in one or many computers at the same time. This new tooling can be seen as another major contribution of this thesis.

Secondary contributions of this thesis also includes introducing a new metric for the calculation of the adaptivity score.

1.4 Outline

The thesis is structured as the following. Chapter 2 will introduce the background information used throughout the thesis. Chapter 3 will explain the previous work this thesis is built upon. Related works in the literature are presented in Chapter 4. Chapter 5, one of the two most crucial chapters in this thesis, explains the experimental infrastructure and implementation details. Chapter 6 showcases the results of our evaluation and goes in detail what they mean. Chapter 7 concludes the thesis by mentioning the limitation, threats to validity and possible future works.

2 Background

This chapter explains the necessary background knowledge used throughout the thesis. Two major concepts used in the implementation and evaluation of our system is adaptivity and knowledge aggregation with Subjective Logic. Section 2.1 will provide the necessary definitions regarding adaptivity and self-adaptive systems. Subjective Logic in general and how it is used for knowledge aggregation will be explained in Section 2.2.

2.1 Adaptivity and Self-Adaptive Systems

Cyber-physical systems (CPS) often have to handle and compensate for the uncertainties in their environments. These systems need to be able to adapt to different scenarios in order to achieve their business goals successfully. Cyber-physical systems that can adapt to unknown scenarios and learn from them are called Self-Adaptive Systems (SAS) or Self-Adaptive Cyber-Physical Systems (SACPS).

A system can be converted to a SAS by two commonly used methodologies. These are called control-based self-adaptation [2, 12] and architecture-based self-adaptation [9, 14] [47]. This thesis and its predecessors [5, 34, 40] use architecture-based self-adaptation. Architecture-based self-adaptation commonly divides the system into two main elements, namely the adaptation logic and the managed element. The system interacts with the context, which is often the environment the CPS exists in. This distinction can be seen from Figure 2.1.

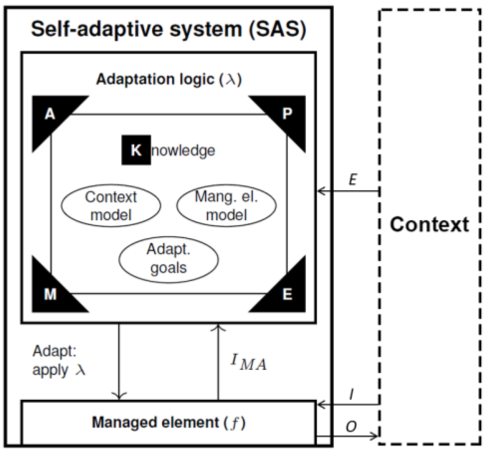


Figure 2.1: *The figure shows the different components of the MAPE-K approach for self-adaptation. The adaptation logic consists of the closed-feedback loop of MAPE and the shared knowledge. M stands for Monitoring, A stands for Analyzing, P stands for Planning and E stands for Executing. K or the Knowledge is shared between all of the components. The managed element can be seen as the CPS, while the context is the environment in which this CPS exists in. Adopted from [46]*

As seen in Figure 2.1, the adaptation logic is based on the *MAPE-K* closed feedback loop. MAPE-K stands for **M**onitor, **A**nalyze, **P**lan and **E**xecute. These four components form the feedback loop that achieves self-adaptation while the **K**nowledge is shared between all of the components [27] [30] [29]. The managed element is the CPS that is desired to be self-adaptive. The operating context, or the context for short is the part of the environment that the CPS can interact with. The self-adaptivity of a CPS depends greatly on the context it exists in [37]. The CPS interacts with the context during its operations and need to compensate for the changes in the context. This behaviour makes self-adaptivity challenging.

This thesis and its predecessors use master-slave pattern of the MAPE-K adaptation logic [50]. A visualization of the master-slave pattern of the MAPE-K approach can be found in Figure 2.2. In the master-slave layout, the managed element monitors the context and execute commands from the adaptation logic. The adaptation logic analyzes the information coming from the monitor, combines it with the shared knowledge and plans the necessary actions.

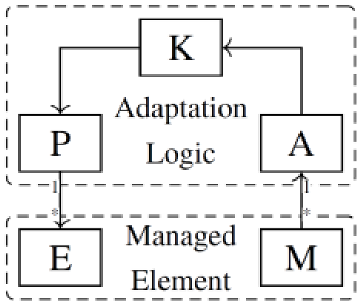


Figure 2.2: *The figure shows the flow of the Master-Slave pattern of the MAPE-K feedback loop. Adopted from [48]*

For our model problem described in Section 3.1, the monitoring is done via LiDAR sensors and the managed element is a dirt cleaning robot. The context is a simulated room with dirts spawning in random places. Different components of the MAPE-K feedback loop is realized via centralized and distributed ROS nodes.

2.1.1 Definition

First step of determining whether a system is self-adaptive or not is defining adaptivity itself. However in the literature, there is a lack of consistent definitions and formalisms regarding adaptivity and self-adaptive systems. Petrovska [37] introduces a formal framework that discusses the definition of adaptivity and the conditions in which a system can be considered self-adaptive. This thesis and its predecessors use those formalisms in order to implement and evaluate the adaptivity of our ROS-based implementation of the multi-agent self-adaptive cyber-physical system (MA-SACPS). This section explains the definition of adaptivity using Petrovska's [37] framework.

Kugele et al. [28] argues that it is challenging to classify the entirety of a system as autonomous or non-autonomous. The authors argue that the autonomy is defined for individual functions. Petrovska [37] adopts the same principle for self-adaptive systems. The author argues that the adaptivity of a system can be defined for a specific system function sf. System function sf maps the input $i \in I$, system's internal state $\sigma \in \Sigma$ and context $c \in \mathbb{C}$ to the output $o \in O$ and new internal state $\sigma' \in \Sigma$ as shown in Equation 2.1 [37].

$$sf : I \times \Sigma \times \mathbb{C} \rightarrow O \times \Sigma, f(i,\sigma,c) = (o,\sigma') \tag{2.1}$$

The author defines behaviour of the system in a given time frame as a sequence of system functions $(sf_0, sf_1, ...sf_t...)$. Furthermore, the author defines a *Quality Function (Q)* that represents how good the business goals and the adaption goals of the managed element is achieved in a given time frame. The Q-score of the system can be seen as a numerical value between 0 and 1 that represents how adaptive the system is in the given time frame (Equation 2.2). A Q-score of 0 would mean the worst possible fulfilment of the goals, while a score of 1 indicates the best possible outcome.

$$Q : (I \times \Sigma \times \mathbb{C} \rightarrow O \times \Sigma) \rightarrow [0,1] \tag{2.2}$$

Q-score should change for each different use-case as the goals of each system is different. The goal of the adaptation logic is to improve this Q-score. Petrovska [37] defines a system as adaptive if the Quality Function Q converges towards a limit ℓ. This is defined formally in Equation 2.3.

$$\begin{gathered} \exists \ell, \epsilon, i \forall t \geq i : |Q(sf_t) - \ell| < \epsilon, \\ \text{with } \ell, \epsilon \in [0,1] \text{ and } 0 \leq \ell - \epsilon < \ell < \ell + \epsilon \leq 1 \end{gathered} \tag{2.3}$$

Figure 2.3 shows the evolution of the system functions of an adaptive system. The system is considered adaptive as the Q-score converges towards a limit ℓ with a threshold ϵ.

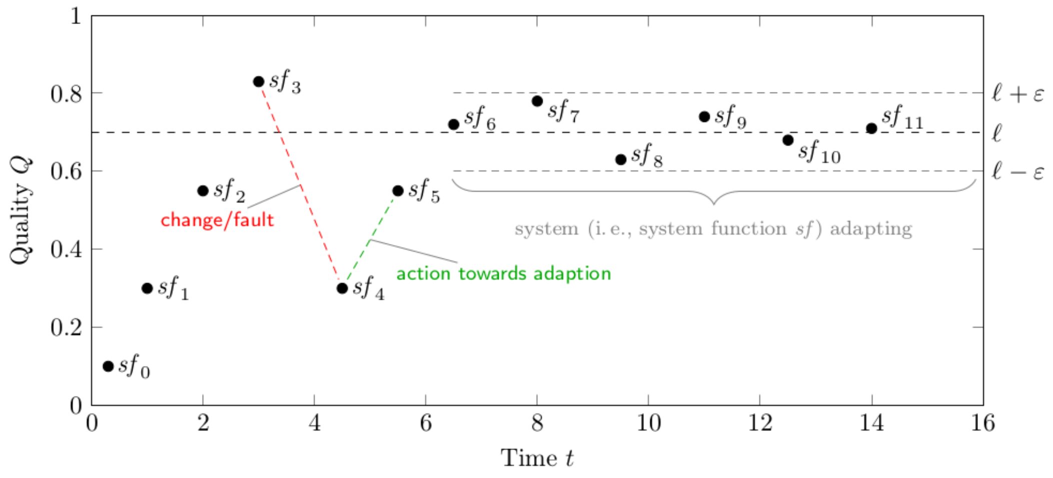


Figure 2.3: *The figure shows the evolution of the system functions sf_t over time. The system functions converge toward a limit ℓ and with the threshold ϵ. Because the system functions converges towards a limit, this system is deemed adaptive.*

As explained and defined in Equation 3.3, if a system is adaptive, the Q-score converges towards a limit ℓ. The Q-score does not necessarily increase monotonically. A change in the context or an internal change in the managed element can lead to sudden drops or increases in the value of the Q-score. Because of this, the value of the Q-score at a specific time frame does not give any information about the adaptivity of the system. Adaptivity of the system can only be determined by looking at the patterns the Q-score evolves towards. Petrovska [36] also proposes a framework called **SAST framing** to determine whether a system can be considered as adaptive. SAST stands for specifying the **s**ystem functions, **a**daptivity goals, the **s**ituation and the **t**ime period the system is considered adaptive for. The author argues that all four of these components need to be specified in order to determine a system as self-adaptive or not.

2.2 Subjective Logic

Cyber-physical systems operate in an environment that is filled with uncertainties. Different agents can perceive the same event in different ways. Binary logic and probabilistic logic are two of the most commonly used frameworks to understand the universe. Binary logic consists of propositions that are either true or false and probabilistic logic maps the likelihood of an event to a value between 0 and 1. Neither of these frameworks are suitable to assess subjective observations coming from the measurements of cyber-physical systems. Subjective logic (SL) is a framework that aims to bridge this gap.

Subjective logic allows capturing the fuzziness and the uncertainties of the real world. Subjective logic deals with subjective opinions instead of propositions. These subjective opinions allow probability values with varying degrees of uncertainties. This property comes from the Dempster-Shafer theory. Subjective logic also allows mapping a subjective opinion to its classical probability distribution [43] [24].

A subjection opinion encompasses a "belief" about a state variable $X \in \mathbb{X}$. This thesis uses a binary domain $\mathbb{X} = \{x, \bar{x}\}$ for the model problem described in 3.1, For our use-case x represents a cell is occupied by a dirt, while its complementary counterpart $\bar{x}$ represents the cell is unoccupied. These types of subjective opinions are called *binomial subjective opinion* as the domain is binary. This thesis and its predecessors are focused solely on binomial subjective opinions. Therefore theoretical discussions in this section will be limited to this subdomain as well.

Binomial subjective opinions are defined formally as the following [25]. Let X be a binomial random variable in the binary domain $\mathbb{X} = \{x, \bar{x}\}$. Opinion about the truth of value x is the ordered quadruplet $O_x = (b_x, d_x, u_x, a_x)$ where the parameters are defined as the following. b_x is the belief mass in support of $X = x$. d_x is the disbelief mass in support of $X = \bar{x}$. u_x is the uncertainty mass that represents the vacuity of evidence. a_x is the base rate, the prior probability of x without evidence. These parameters satisfy the additivity requirement in Equation 2.4.

$$b_x + d_x + u_x = 1 \tag{2.4}$$

A *vacuous opinion* is defined as a subjective opinion that has $u_x = 1$. As explained before a binomial subjective opinion can be projected into a probability distribution. This projected probability of a subjective opinion about the value x is defined in Equation 3.5.

$$P(x) = b_x + u_x a_x \tag{2.5}$$

Barycentric coordinate system on an equilateral triangle can be used to visualize binomial subjective opinions. An example of this can be seen in Figure 2.4.

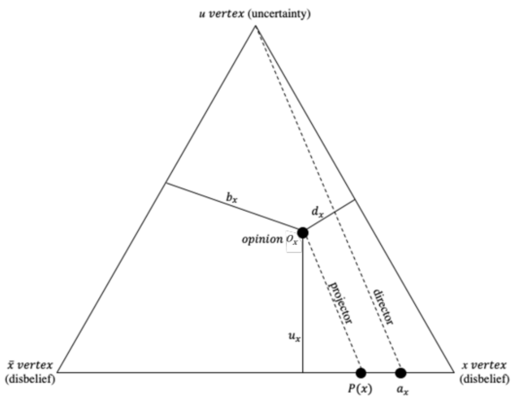


Figure 2.4: *Visualization of a binomial subjective opinion using a barycentric coordinate system on an equilateral triangle. Adopted from [25]*

A binomial subjective opinion such as $O_x = (0.4, 0.2, 0.4, 0.9)$ can be plotted using the triangle in Figure 2.4. The lines connected to the point represents belief mass(b_x), disbelief mass(d_x) and uncertainty(u_x) of the subjective opinion. The base rate (a_x) resides in a point on the base line and is connected to the uncertainty vertex by a line called base rate director. The projected probability ($P(x)$) can be found by drawing a parallel line to the base director and calculating the point the line intersects with the base line. The parallel line is called the projector.

2.2.1 Knowledge Aggregation with Subjective Logic

There are often many different opinions about the same proposition. This could be several cyber-physical systems publishing their observations about a physical object in the environment. This observations can support the same belief or conflict with each other due to some measurement error. Whether they support the same belief or not, these observations have to be merged into a single, collective opinion. This process is called belief fusion. We use belief fusion in this thesis for aggregating knowledge between different dirt cleaning robots.

Belief fusion can be achieved via various different operators. These operators can be extensions of binary logic operators such as the AND operator or operators that are unique to the subjective logic framework. Several commonly used operators in subjective logic are as follows [19, 26]:

- **Belief Constraint Fusion** (BCF) combines the opinion of the agents into a single collective opinion. The opinions must support the same belief otherwise the belief fusion is not possible with BCF. An example of this operator can be a group of friends trying to decide which movie to watch. If all of the group agree on a specific movie, a single collective opinion can be formed. Otherwise, the group can not agree.
- **Cumulative Belief Fusion** (CBF) treats the opinion of the agents as evidence for a proposition. The opinions cumulatively increase the belief or the disbelief of the final collective opinion. The uncertainty of the collective opinion decreases with each coming evidence. An example of this operator can be aggregating measurements for a specific experiment into a single collective opinion that has an uncertainty smaller than each individual measurement. This operator is best used for non-conflicting measurements.
- **Averaging Belief Fusion** (ABF) averages all of the incoming observations into a single collective opinion. Each opinion affects the final result in the same way. Unlike other operators, vacuous opinions are aggregated into the final collective opinion with the same weight as all of the other observations. An example of this operator can be a board of professors grading the examination of a student.
- **Weighted Belief Fusion** (WBF) averages all of the incoming observations into a single collective opinion like the ABF operator. However in WBF, each opinion is assigned a weight depending on its uncertainty value. Opinions with high uncertainty are assigned lower weights so they affect the final collective opinion in a smaller way. As vacuous opinions have an uncertainty $u_x = 1$, they get assigned a weight of 0 and they do not affect the aggregated opinion.
- **Consensus & Compromise Fusion** (CCF) preserves the belief masses of the individual opinions while doing the aggregation. If the opinions are conflicting, a

compromise opinion with an increased uncertainty is formed. An example of this operator can be a board of doctors diagnosing a patient. Each doctor has a different area of expertise and looks at the problem from a different perspective. CCF operator would identify the shared set of beliefs, all doctors in the group agrees upon.

The thesis of Neuss [34] showed that using any of these operator by themselves is not enough to achieve long term knowledge aggregation. The author states that a combination of the CCF and the CBF operators resulted in the best outcome for our use-case [34]. This can be seen from Figure 2.5, where the performance of the combination operator does not degrade over time unlike the CCF and CBF operators. The combination aggregation scheme outperform both of the other operators.

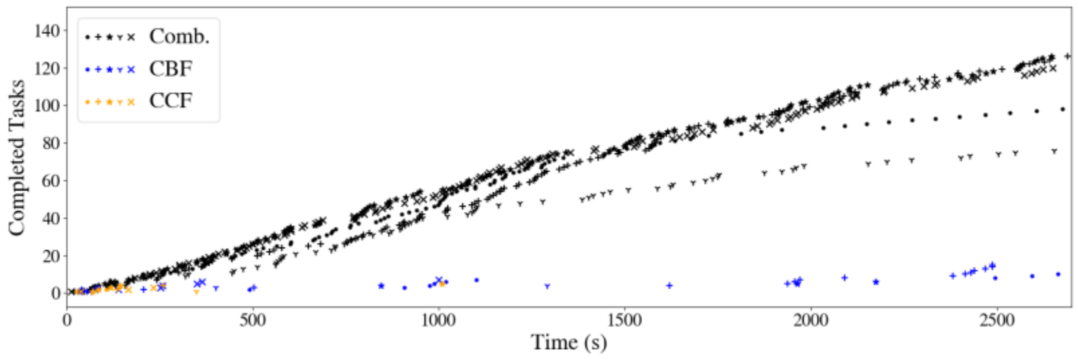


Figure 2.5: *The figure shows the combination aggregation scheme outperforms CCF and CBF operators for our use-case described in Section 3.1. Number of completed task slowdown for the CCF and CBF operators while the combination aggregation scheme's performance does not degrade over time. Adopted from [34]*

In this thesis, we continue to use this combination of CCF and CBF in order to do knowledge aggregation. The experiments in Chapter 6 uses this combination operator for the settings in which knowledge aggregation with subjective logic is enabled.

3 Previous Work

The software system that we use in this thesis is a product of many previous projects and theses. This chapter will explain the previous work this thesis is built upon. This thesis uses two main subsystems, namely the ROS-based implementation of the multi-agent self-adaptive cyber-physical system (MA-SACPS) and the wrapper that extends its functionality called the evaluation framework. Section 3.1 will explain the model problem that is being examined. Section 3.2 will explain the evolution of the simulator and its state we inherited for this thesis. Section 3.3 will mention the motivation for the evaluation framework and its implementation details at the start of this thesis.

3.1 The Model Problem

The model problem that we are investigating in the scope of this thesis was first proposed by Yousfi [52]. The same use-case was used throughout the evolution of the system. The problem was adapted by Quijano [40] and took its final form during the theses of Neuss [34] and Bergemann [5].

The use-case is defined as the following. There are a predetermined number of robots traversing a fixed map with different rooms and obstacles. Robots have the functionality to clean dirts by moving on top of them. The dirts spawn in random location of the map depending on the chosen dirt distribution pattern. They have fixed spawn intervals which is chosen at the beginning of the simulation. Figure 3.1 shows the model problem seen from the graphical interface of the Gazebo simulator.

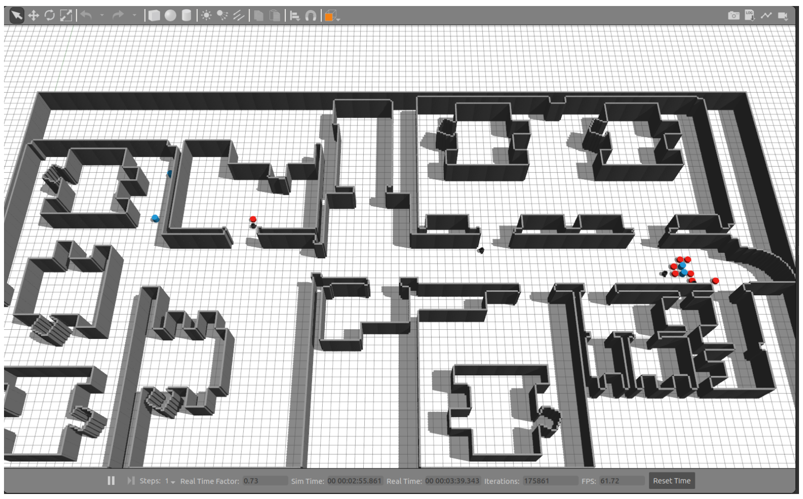


Figure 3.1: *The figure shows the model problem as seen from the Gazebo simulator view. The map is separated into rooms with the use of walls. Red cylinders represents the detected dirts and the blue cylinders are dirts that are not yet detected. Black TurtleBot robots traverse the map in order to detect and clean the dirts. Taken from Bergemann [5].*

The robots are equipped with LiDAR sensors to observe their surroundings. The sensor have a maximum range of 3.5m. The robots do not have a priori knowledge about the locations of the dirts or the room. They also can not observe the room fully as the room used in this thesis has the dimensions 10 meters x 10 meters and the LiDAR sensor only has a maximum range of 3.5 meters and they are subjected to Gaussian error ϵ_d. For every LiDAR measurement the error ϵ_d is sampled using Equation 3.1 where the units for the standard deviation and the mean is in meters.

$$\epsilon_d \sim \mathcal{N}(\mu = 0.022, \sigma = 0.02) \tag{3.1}$$

If the measurement after the error is added exceeds the maximum range of the sensor, it is clipped to 3.5 meters. A graphic demonstration of the difference between the pure measurement of the LiDAR sensor and the measurement with the Gaussian error added can be seen in Figure 3.2.

After the thesis of Quijano [40] it was realized that the uncertainties that come from the noisy sensor only affects the dirts detected by the very end of the sensor. This behaviour occured very rarely to have an impact on the system. This led to the creation of false positive and false negative tasks in the thesis of Neuss [34].

The tasks can be ground truths or false positives. Ground truths are dirts that actually exist and the robots should clean them. False positive dirts are assigned to a specific robot and can only be detected by that robot. The system also supports having false negatives. In the case of false negatives the robot is unable to detect an existing dirt. The robots aggregate their knowledge about the environment to mitigate all these uncertainties. In the ideal case the robots should clean the ground truths, avoid the false positives and counteract the false negatives through knowledge aggregation.

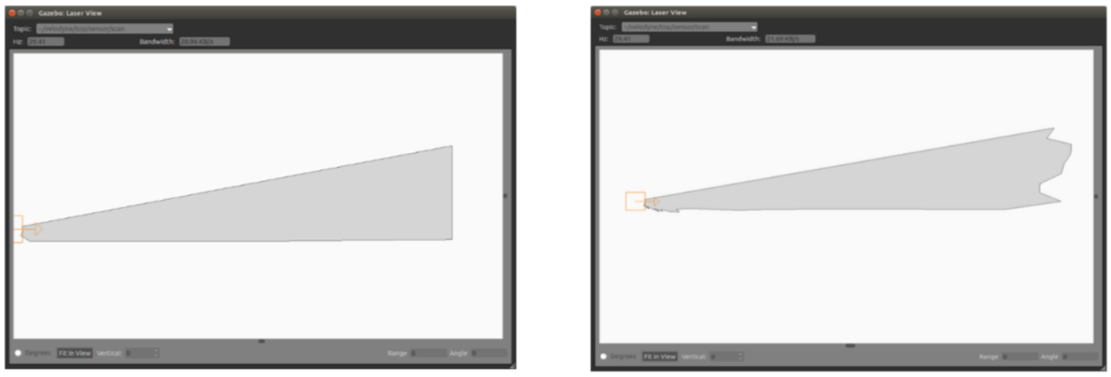


Figure 3.2: *The figure shows the difference between the unaltered measurement of the LiDAR sensor and the measurement after the Gaussian error is added. The figure on the left is the unaltered measurement and the figure on the right is the measurement with the Gaussian noise $\mathcal{N}(\mu = 0.022, \sigma = 0.02)$. Adopted from Neuss [34]*

3.2 Implementation of MA-SACPS

Implementation of the ROS-based multi-agent self adaptive cyber physical system (MA-SACPS) evolved as a result of several theses and projects.

The software that aims to implement the model problem described in section 3.1 was first developed in the scope of a practical course in Technical University of Munich. The practical course was titled “Praktikum: Smart Self-Adaptive Cyber-Physical Systems - A road towards Autonomous Systems”. There were two teams Team 1 [4] and Team 2 [33] that proposed their solutions for a ROS and Gazebo-based implementation of multi-agent self-adaptive cyber-physical system. Both of the teams proposed their own approach for the same problem and best parts of both of the approaches were merged into a single system during the master’s thesis of Quijano [40]. The merged system had adaptive scheduling, costmaps that contain knowledge-aggregation, auction-based allocation, realistic detection and goal validation. The new system could detect dirts with LIDAR sensors that had Gaussian noise, thus followed a realistic approach. Quijano was also investigating the usage of Subjective Logic at the time and used the theory for representing the uncertainty for the knowledge aggregation [40]. After Quijano’s thesis Neuss started his own master’s thesis on the evaluation of the knowledge aggregation. Neuss discovered several errors and limitations with the system. This shifted the focus of his thesis to fix the discovered issues. Most of the bugs and limitations were resolved, resulting in the version of the simulator that we inherited for this thesis [34]. A high-level overview of the final version of the simulator can be seen from the image below. The orange boxes in the image represent the centralized ROS-nodes that exist once for the whole system and the blue nodes are the distributed ROS-nodes which exist for each separate robot. The nodes communicate using publisher-subscriber pattern. Individual ROS-nodes that form the system will be explained in their own subsections.

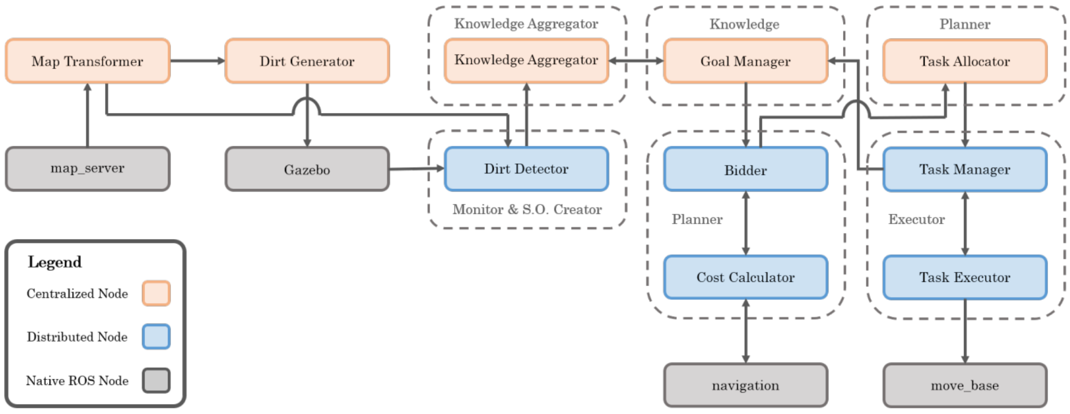


Figure 3.3: *ROS nodes of the simulator. Adopted from Neuss [34]*

Map Transformer

Map transformer is a centralized node that transforms the high resolution map used by Gazebo into a 20x20 low-resolution grid to be used internally by the other nodes. The grid cells can either be occupied with an obstacle like a wall or be empty. The generated grid map is used for the context model which hold the information of where the dirts on the map is located. The grid map is created only once and at the beginning of the simulation. This node is overridden by the evaluation framework's `Map Provider` node.

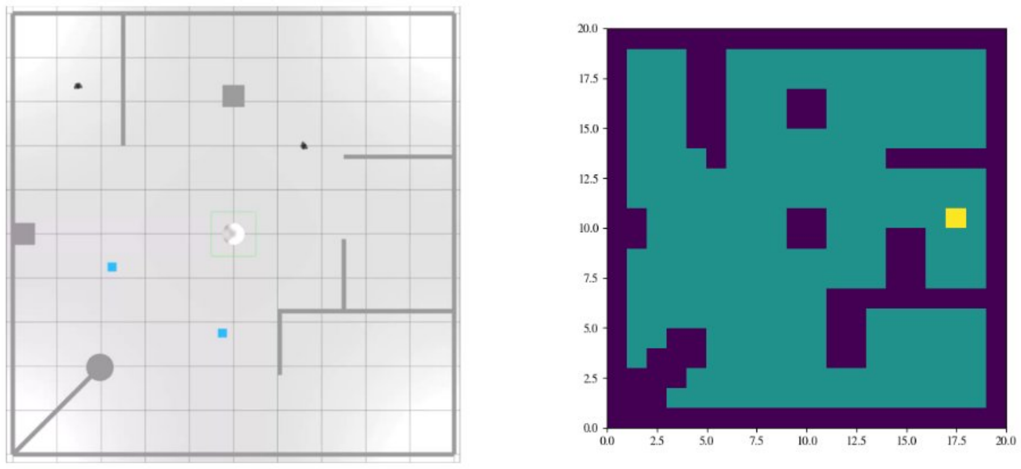


Figure 3.4: *The right-side shows the high-resolution map as seen in Gazebo. Left-side shows the low-resolution context model that holds the locations of the obstacles and the dirts. Adopted from Neuss [34]*

Dirt Generator

Dirt generator is a centralized ROS node that is responsible for spawning the ground truth and false positive dirts on the map. The dirts have the physical form of a cylinder with 0.25 meters in Gazebo. The dirts have a physical boundary so the robots can detect them using the sensors.

The `Dirt Generator` node samples a Beta distribution $B(\alpha = 2, \beta = 2)$ and scales it to the width (W) and the height (H) of the map to find the spot where the new dirt should be spawned. The calculated values corresponds to the x and y coordinates on the map (Equation 3.2 , 3.3).

$$x_{dirt} \sim \frac{W}{B(\alpha = 2, \beta = 2)} x^{\alpha-1}(1-x)^{\beta-1} \tag{3.2}$$

$$y_{dirt} \sim \frac{H}{B(\alpha = 2, \beta = 2)} x^{\alpha-1}(1-x)^{\beta-1} \tag{3.3}$$

The cell of the grid map that contains the position (x_{dirt}, y_{dirt}) is determined. If the cell is unoccupied by a wall or by another task, the new dirt is spawned in the middle of that cell. The false positive dirts can spawn in cells that are occupied by other tasks. If the dirt can not be placed in the proposed cell, another sample for (x_{dirt}, y_{dirt}) using Equation 3.1 and Equation 3.2. This process repeats until a suitable cell is found for the new dirt.

The time intervals in which a new dirt should be spawned is fixed at the beginning of the simulation. Ground truths (true positives) always spawn with the given spawn interval. This interval is 60 seconds for all of the experiments done in this thesis. The time interval in which a false positive should be spawned, is calculated from the false positive probability of the robots and the spawn intervals for the ground truths(T_{TP}). The false positive probability for each robot ($p_{FP,i}$) is taken as a simulation parameter at the start of the simulation. The calculation for the spawn interval of false positives for the robot i ($T_{FP,i}$) can be found in Equation 3.3.

$$T_{FP,i} = \frac{1 - p_{FP,i}}{p_{FP,i}} T_{TP} \tag{3.4}$$

The dirt distribution logic described here is overriden by the `Global Dirt Generator` node after the merger with the evaluation framework.

Dirt Detector

`Dirt Detector` is a distributed node that exists for each robot. It is responsible for analysing the output of the LiDAR sensor for the robot it belongs to. `Dirt Detector` node determines whether the laser sensor detects a valid task or not. Valid tasks are distinguished from other objects such as walls, obstacles or other robots. After detecting a dirt, it cross references the list from the `Goal Manager` node to determine whether the detected dirt is already assigned to the robot.

`Dirt Detector` node also handles the false negative measurements. As explained previously in section 3.1, false negatives are dirts that actually exist but the robot is unable to measure.

Knowledge Aggregator

`Knowledge Aggregator` is a centralized ROS node that handles the knowledge aggregation using the Subjective Logic library. It is responsible for creating a map called the opinion grid that contains the aggregated observations. The created grid contains the opinions aggregated according to the chosen Subjective Logic operator. Subjective Logic and the available operators are explained in detail in Section 2.2. The opinion grid is updated whenever there is a new opinion comes from the `Dirt Detector` node.

Goal Manager

Goal Manager is a centralized node that keeps a list of all the available tasks that exist in the map. There are two types of tasks: 'undetected tasks' and 'goals'. An undetected dirt is classified as a goal when the robots detect it and the opinion grid for that dirt exceeds a certain threshold. When a dirt is classified as a goal, its physical boundary in Gazebo is removed. This allows the robots to move towards the center of the goal to clean it.

The goals can be ground truths or false positives. False positives are programmed to dissappear from the map if they are not cleaned in 2 minutes. This is done to avoid overcrowding the map with false positive dirts.

Goal Manager is triggered whenever the `Dirt Detector` node detects a new dirt. The cleaned dirts are removed from the list and their physical models are removed from Gazebo by this node as well.

Bidder

`Bidder` is a distributed node that exists for each robot. This node gets triggered whenever the Goal Manager node classifies a new goal. The schedule of the robot which is an ordered list of tasks the robot should complete is stored inside the `Bidder` node. A bid about the new goal is formed by looking at the closest goal of the robot. The cost of inserting the new goal before and after the closest goal is calculated by the `Cost Calculator` node. The smaller of those two costs is determined as the ’bid’ and sent to the `Task Allocator` node.

Cost Calculator

The Cost Calculator node calculates the possible paths to add a new goal to the robot’s existing schedule. As seen in Figure 3.5, when a new goal is detected, the possible path of adding the new goal before and after the closest goal is calculated.

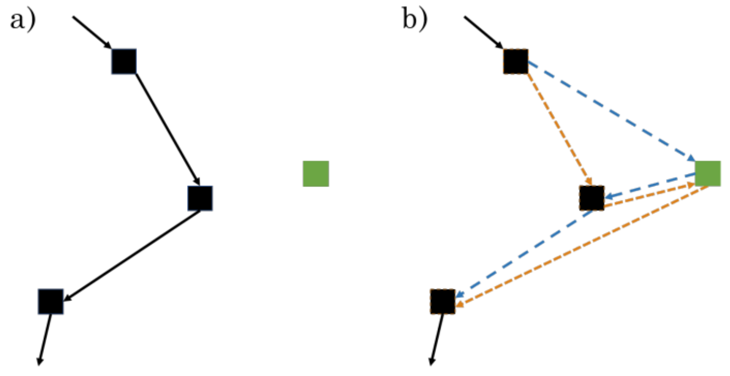


Figure 3.5: *The figure (a) shows the original schedule of the robot. In figure (b) a new goal (green square) is detected and the possible paths are calculated (blue path and the orange path). The path that has the lower cost will be determined as the ’bid’ and sent to the Bidder node. Adopted from Neuss [34]*

The shorter path of those two is determined as the bid and gets sent to the Bidder node. The paths are usually not straight lines as the robot needs to avoid the obstacles and the walls in the map. The paths are calculated using the native ROS node `move_base`.

Task Allocator

`Task Allocator` is a centralized ROS node. Its purpose is to collect all of the bids from the available robots for a specific dirt. `Task Allocator` finds the smallest among them and assigns the dirt to the robot that owns that bid. The bid collection ends if all of the robots submit their bids for a specific dirt or a predetermined timer runs out. The time-out scenario happens when a robot is unable to submit their bid. Once a dirt is assigned to a robot, the `Task Manager` node is notified.

Task Manager

`Task Manager` is a distributed node that exists for each robot. It holds a queue of tasks the robot needs to complete. The task that is on top of the queue is sent to the `Task Executor` node. Task Executor node explained in the subsection below sends a message back to the Task Manager when that task is completed. The completed tasks are removed from the queue. When a new task from the `Task Allocator` node arrives, the queue is adjusted accordingly.

Task Executor

Task executor is a distributed node that exists for each robot. It sends messages to the native ROS node `move_base` to make the robot move from its current position to position of the next goal the robot has to accomplish. The next goal is usually a dirt that needs to be cleaned. If no dirt is detected, the goal can be an exploration goal that points to a random point in the map as well. The node is also responsible for notifying the `Task Manager` node when a goal is reached.

Output Timer

The `Output Timer` node is a different node in the sense that it is a part of the MA-SACPS subsystem but it is used by the evaluation framework. It was added during the thesis of Bergemann to allow collecting the data evaluation framework needs [5]. It subscribes to the desired ROS topics, format the messages coming from them and publishes the formatted information to the ROS topics evaluation framework listens to.

3.3 Evaluation Framework

The evaluation framework is an abstract wrapper around the simulator that extends its functionality. Figure 3.6 shows how it intersects with the MA-SACPS subsystem. It was proposed and implemented during the master's thesis of Bergemann [5]. The main motivation of the evaluation framework is to evaluate the adaptivity of the system with the introduction of the Q-score. The framework also offers many improved functionalities in terms of configuration and evaluation. It allows changing the map and the dirt distribution which was not possible previously. Comparisons between the configuration and the evaluation made by the MA-SACPS subsystem and the evaluation framework can be found in Section 5.2. Functionalities of the evaluation framework are used extensively in this thesis as well.

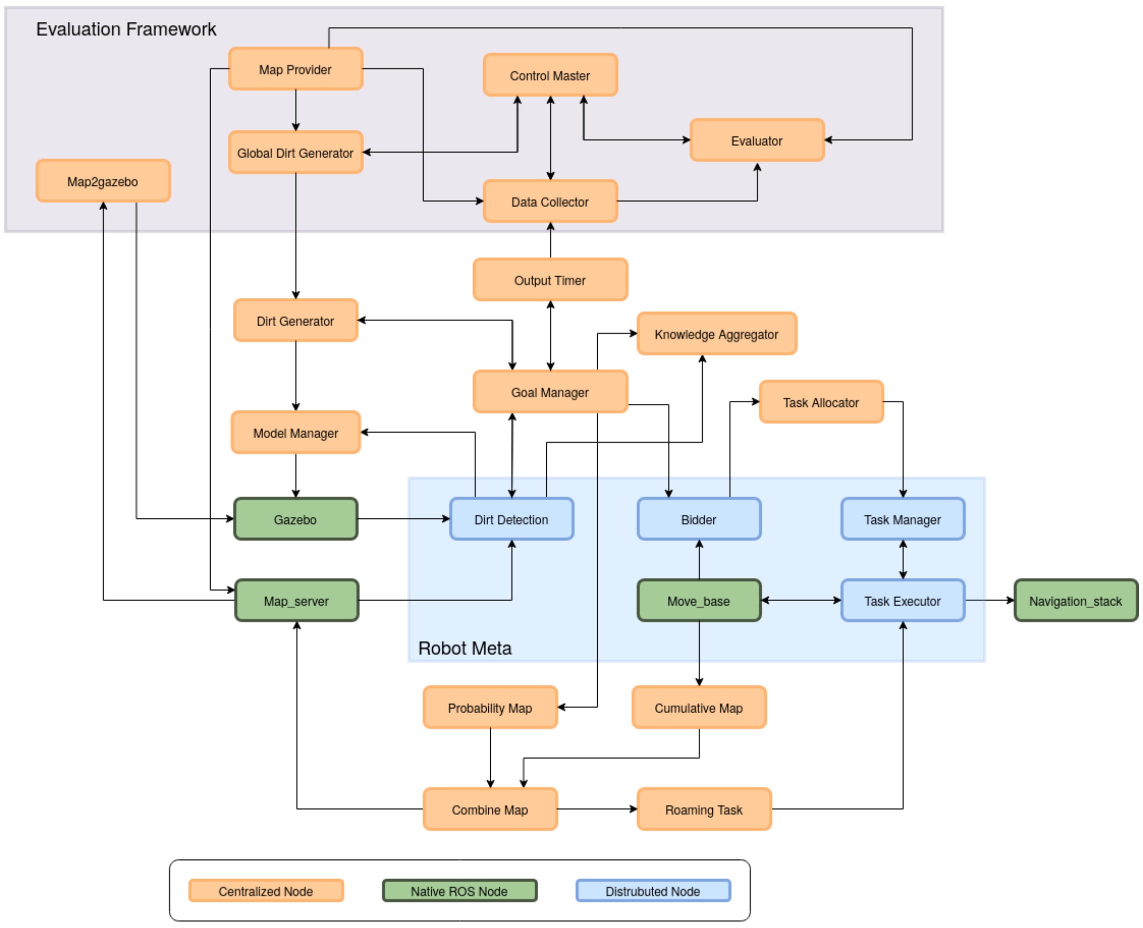


Figure 3.6: *High level view of the previous state of the combined system. The image shows how the ROS-nodes of the MA-SACPS and the evaluation framework intersect. Modified the image taken by Petrovska et al. [38]*

Control Master

`Control Master` is a centralized ROS node that orchestrates launching the other nodes of the evaluation framework. The node waits until all of the nodes of the evaluation framework boots up properly and launch the actual simulation. `Control Master` node ensures the MA-SACPS subsystem receives the map and the input data coming from the evaluation framework. At the end of the simulation, this node will shut down the simulator and start the evaluation process. After the evaluation is completed, the node will shut down all of the ROS nodes and the `rosmaster` to conclude the simulation.

Map Provider

`Map Provider` is a centralized node that overrides the `Map Transformer` of the MA-SACPS subsystem. This node extends the `Map Transformer` node by adding two additional functionalities. First, it removes the unreachable grid cells from the grid map. Also, it allows changing the resolution of the map by scaling the map according to the resolution. The resolution can be controlled with the parameters.yaml configuration file (Listing 3.1).

```yaml
# parameters.yaml

...
# Define how big each cell should be in the provided map:
# (a cell is a pixel of the inital map image)
map_resolution: 0.5 # m/cell (width and height of a cell)
...
```

Listing 3.1: *The configuration line in parameters.yaml that allows changing the resolution of the map*

Global Dirt Generator

`Global Dirt Generator` is a centralized node that overrides the `Dirt Generator` node of the MA-SACPS subsystem. It extends the dirt generation process in several ways. As explained in the related subsection, `Dirt Generator` node only allows having uniform dirt distribution using a Beta distribution. `Global Dirt Generator` node allows non-uniform dirt distribution, where the dirt is distributed around hotspots. The number and the location of the hotspots can be configured via the parameters.yaml file. Figure 3.6 showcases this functionality. The left-most image on Figure 3.6 represents the uniform distribution where each cell has the same probability of being chosen for spawning the new dirt. The middle and the right-most images shows non-uniform dirt distribution. The right-most map has one hotspot located in the bottom left corner of the map, while the middle map has several hotspots.

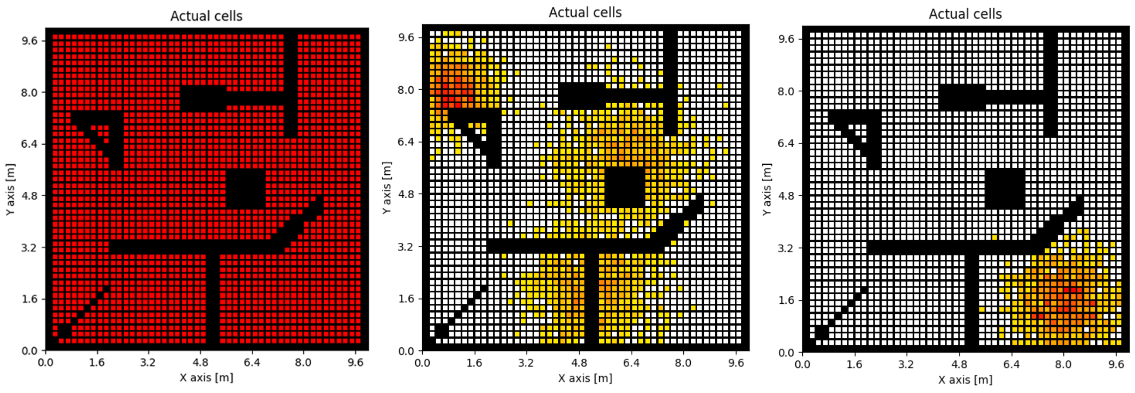


Figure 3.7: *The left-most map has the uniform dirt distribution enabled. Each cell has the same probability of being chosen for spawning the new dirt. The middle and the right-most images has non-uniform dirt distribution. The right-most map has one hotspot located in the bottom left corner of the map, while the middle map has several hotspots. Figure taken from Bergemann [5].*

Furthermore, the Beta distribution is replaced with normal distribution for the dirt generation process. The user configures the position and the spread of each hotspot at the beginning of the simulation via the parameters.yaml file (Listing 3.2).

```yaml
# parameters.yaml

...
dirt_hotspots:
- {
    x: -4.0,
    y: 3.0,
    spread: 0.6
  }
- {
    x: 2.5,
    y: 1.0,
    spread: 0.8
  }
...
```

Listing 3.2: *The parameter in parameters.yaml that is used to configure the positions and the spreads of the hotspots used for non-uniform dirt distribution. The x and y fields are used as the mean of the normal distribution while the spread field is used as the standard deviation.*

The dirt generator samples two normal distributions for each hotspot corresponding to the coordinates in x and y axes. The mean of the normal distribution is the hotspot center given in the parameters.yaml file and the standard deviation is the corresponding spread variable (Listing 3.2). These two samples are combined to create a bi-variate normal distribution that is used to determine the location of the new dirt.

Map2gazebo

Map2gazebo is a centralized node that transforms the given low-resolution 2d grid map into a high-resolution 3D Gazebo map.

Data Collector

Data collector is a centralized node that collects the necessary data to be used by the `evaluator` node. It subscribes to the topics published by the `Output Timer` node of the simulator.

Evaluator

Evaluator is a centralized node that runs at the end of the simulator. It takes the data from the `Data Collector` node and calculates the Q-score with that data. This node has a helper script called `score_functions.py` that holds the score functions used to calculate the Q-score. The parameters.yaml configuration file determines which weights should be used for which score functions, and the Evaluator node calculates the Q-score for each timeframe. It also calculates the average and maximum Q-scores of that specific simulation run and writes the results into csv files to be examined later.

4 Related Work

This chapter summarizes the related work in the literature. Even though the field of SAS is constantly growing and expanding, we kept the scope of this literature review focused on two topics. Section 4.1 explains the related work on evaluating self-adaptive systems. Section 4.2 summarizes the literature on mitigating uncertainties in self-adaptive systems.

4.1 Evaluating Self-Adaptive Systems

Evaluation of self-adaptive systems are an important but often overlooked research area. There are several important pieces of literature exploring this field. Zhu et al. [53] describes the main metric to evaluate the learning behaviour of a robotic system is to take the difference between different runs of the same system and see if the system's behaviour is improving or not according to the defined goals. This approach requires a way of calculating the difference between metrics. This is often done by having a unified metric [53]. A similar notion is also stated in [42] to evaluate self-adaptation in large software systems. The authors state that different metrics can be unified into a average metric and this average score can be used for comparison. The comparison is usually done between the current run and an established base case [42].

A recent study [17] showed that evaluation of self-adaptive systems mostly use similar metrics as other, non-adaptive, software systems. The evaluation metrics usually very use-case specific and mirror the business costs. The metrics can be loosely grouped to several categories. Time based metrics include measuring the time it takes for completion [31], decision [32], response [10], discovery, execution or lifetime [34, 40]. Cost based metrics try to evaluate the for maintenance costs[16], engineering costs [10], resource costs [39], service costs [49] or energy consumption [22, 23, 39]. Other metrics include evaluating the performance [21, 31], accuracy [23, 44, 51], stability [1, 44, 51]. Even though there are several metrics to evaluate self-adaptive systems, no metric exists to evaluate the adaptivity of a SAS due to lack of definitions on system adaptivity [36].

4.2 Mitigating Uncertainties in SAS

Cyber-physical systems may encounter uncertainties in three phases. These phases are the requirement phase, design phase and run-time phase [41]. A recent survey on the research community perspective [20] showed that the majority of researchers believe that the mitigation of uncertainties have to happen in all of these phases. The survey results on the opinion of the research community can be seen in Figure 4.1.

The motivation behind handling uncertanties in design phase and in run-time phase is the following. If an uncertainty can be identified during design phase, it should be mitigated as early as possible in the design phase. Mitigating uncertainties in design phase lowers the cost and improves efficiency. However in most cases the uncertainties are unknown in design phase and can only be resolved in the run-time phase [20].

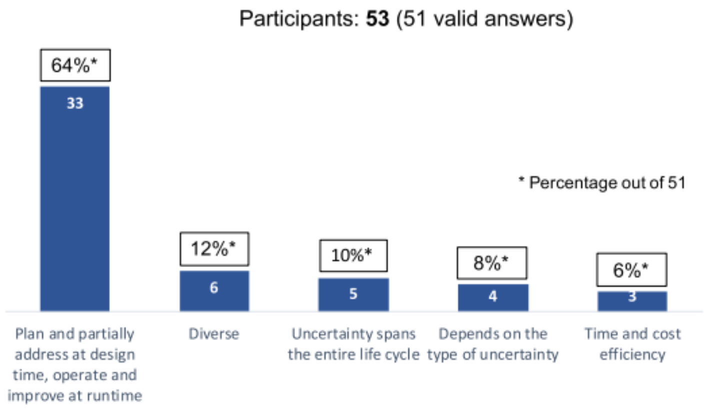


Figure 4.1: *The figure shows the result of a survey done to researchers in adaptivity research. The majority of the researchers believe uncertainties have to be handled in both design phase and the run-time phase. Taken from [20]*

For this work we mostly deal with uncertainties that occur in the run-time phase. Uncertainties that occur in run-time can be classified into internal and external uncertainties [37]. Internal uncertainties are unexpected malfunctions or imperfections that comes from within the agent. Examples of internal uncertainties can be sensor malfunctions or sensor noise. The other type of uncertainties are external uncertainties. These are uncertainties that comes from unpredictable changes in the environment.

There are couple of proposed techniques to mitigate run-time uncertainties. Internal uncertainties like sensor failure can be compensated in various ways [7, 15]. External uncertainties like unpredictable environments can be mitigated as well [3, 7, 8, 15, 35, 41].

Most mitigation techniques are based on one or more feedback loops [7]. The mitigation methods are often use-case specific as well [35]. Rainbow [15], one of the best frameworks to mitigate uncertainties, focuses on architectural reusability. In Rainbow, the feedback loop and the managed element are isolated, resulting in a more knowledge-specific but system-independent infrastructure. Rainbow does not allow altering the adaptation logic in run-time [15].

5 Experimental Infrastructure and Implementation

The goal of this thesis is to evaluate the impact of Knowledge Aggregation (KA) with Subjective Logic (SL) on the overall system's adaptivity in multi-agent self-adaptive cyber-physical systems (MA-SACPS). The simulation system used in this thesis is built upon the ROS-based implementation of MA-SACPS built by Neuss [34] and the evaluation framework implemented by Bergemann [5]. The MA-SACPS subsystem acts as a base to test the effect of knowledge aggregation with Subjective logic in a multi-agent robotic system. The evaluation framework implemented by Bergemann extends this MA-SACPS subsystem [5]. It introduces the adaptivity metric and allows configuring the parameters of a simulation run easily. Section 5.1 will introduce the vocabulary used throughout the chapter. Section 5.2. will explain the previous state of the system in detail.

These two subsystem were merged in the scope of this thesis in order to calculate the adaptivity metric (Q-score) for the multi-agent robotics system with knowledge aggregation. Section 5.3 will describe the steps taken to merge the subsystems and the encountered challenges.

Last but not least, merging of these two subsystems allowed a completely new and improved experimental tooling system. This new system automates many of the steps to design, configure, run, store and analyse the experiments. It allows running the simulations in parallel by containerizing them and reduces the previously needed manual effort greatly. Section 5.4 will explain this new experimental tooling system in detail.

5.1 Implementation Related Vocabulary

Our definition of the MA-SACPS subsystem

As explained in Chapter 3, the simulation system used in this thesis consists of two subsystems. One of them is the evaluation framework which is explained in detail in Section 3.3. The other subsystem consists of the software system used in the thesis of Neuss [34]. The software system used in the thesis of Neuss [34] is referred to as the ROS-based implementation of the multi-agent self adaptive cyber-physical systems, or the MA-SACPS subsystem for short. We use the terms ROS-based implementation of MA-SACPS, MA-SACPS subsystem or simply MA-SACPS interchangeably. All of these terms refer to this software system. This subsystem, its architecture and implementation details are explained in detail in Section 3.2.

Difference between simulations and experiments

Throughout the thesis we use the terms simulations, simulation runs, experiments extensively. Simulation, simulation run or individual runs are used interchangeably. These words refer to a single run of the ROS-based system. With the configuration used in this thesis an individual run takes 45 minutes.

Experiments are made up of several simulations with slightly different configurations. They are designed to prove a hypothesis. For example, Experiment 1.1, which is explained in detail in Section 6.2, we change the false positive probability of the robots from 0.1 to 0.9 to test how it affects the performance of the knowledge aggregation. That experiment has 9 different configuration for the false positive probabilities and it is run with subjective logic on and off. Therefore it is made of 18 simulations. The experiment is repeated 5 times for statistical robustness. Thus 90 simulations are run in total for that experiment.

5.2 Previous State of the System

In this section, the previous state of the ROS-based implementation of the MA-SACPS and the evaluation framework will be discussed. Both subsystems will be explained in terms of configuration, data generation, and evaluation of the collected data.

Configuration

The MA-SACPS subsystem has 11 ROS nodes. Each of these nodes have their own configurable parameters. During the evolution of the system these parameters were collected into a single file called master.launch. All configurable parameters can be found there and be changed. These parameters will then be passed into their own respective ROS nodes. For the experiments done in thesis of Neuss [34], this master.launch file was modified via shell scripts. A single shell script would modify the necessary parameters run the experiment, modify the parameters again, run the simulation again and do this for how many times necessary. As a result of this choice, this shell script would run for hours at a time. If it got interrupted for some reason the entire experiment would need to be restarted.

The evaluation framework improved the configuration step by introducing a single point of entry to the system. It collected all the configurable parameters for the MA-SACPS subsystem and the evaluation framework in a single human-readable file called parameters.yaml. These parameters are passed down to the correct nodes by a ROS node called `control_master`. The control_master node allows the parameters that belongs to the MA-SACPS subsystem to be accessed by the master.launch file.

Data Generation

The MA-SACPS subsystem and the evaluation framework has different data generation methods due to the different perspectives they aim to evaluate.

The MA-SACPS subsystem originally aimed to evaluate the performance of the knowledge aggregation. To evaluate the knowledge aggregation, it collects the robot locations, belief maps, subjective logic observations and achieved goals for every time point. These are collected via a ROS functionality called rosbags. Rosbags are created by dumping the output of one or more ROS topics into a file. This file can then be analyzed or played back to replicate the exact simulation run.

Evaluation framework aims to evaluate the adaptivity of the system by calculating the quality score (Q-score). Contrary to the MA-SACPS subsystem, the evaluation framework collects all of the necessary data in runtime instead of using rosbags. The evaluation framework calculates the Q-score at the end of each run with a ROS node called `evaluator` using this collected data. The Q-score for each timepoint is calculated and written into a csv file called results.csv. The summary of the result that has the average Q-score, max Q-score and the configuration parameters for that specific run is written to a file called result_summary.csv. The evalation framework also generates information about the configuration, a plot about how the Q-score evolves, a video about the simulation history, transformed maps, an image showing how the dirt was distributed and creates separate files for each of these. All of these files are put it into a folder called `evaluation_output`. Detailed explanations on what is collected by the evaluation framework and the folder structure can be found in the thesis of Bergemann [5]. The counter generated in the thesis of Bouzid [6] is put into this folder as well.

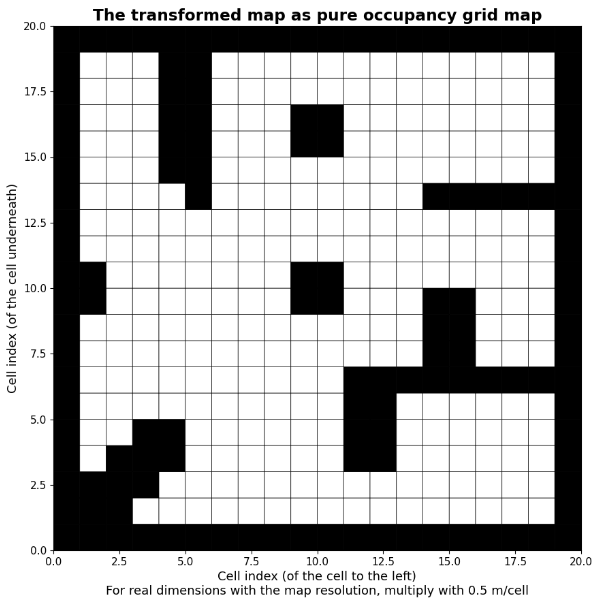

The transformed map as pure occupancy grid map
Cell index (of the cell underneath)
Cell index (of the cell to the left)
For real dimensions with the map resolution, multiply with 0.5 m/cell
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

Figure 5.1: *A sample transformed map created by the evaluation framework and stored in the evaluation_output folder. Implemented by Bergemann [5]*

Evaluation

Because of the different data collection methods used, the MA-SACPS subsystem and the evaluation framework evaluates the collected results differently.

The MA-SACPS subsystem collects all of the necessary information into the rosbag dump file to be evaluated after the run. Several rosbags from different runs after put into a folder. When all of the rosbags for an experiment are collected, the python script called `data_analysis.py` is run. This script loads all of the necessary rosbag files according to the selected experiment, reads them and creates the desired plots.

The evaluation framework follows a different approach. The results are collected while the simulation is running and they are evaluated at the end of the run via a ROS node called `evaluator`. This node calculates the Q-score and generates the files explained in the subsection above. A sample plot that is created by the `evaluator` node can be seen in Figure 5.2. The figure shows the evolution of the Q-score for each time frame. The Q-score is calculated from two business goals (no crashes and minimum 10% cleaned) and four adaptation goals.

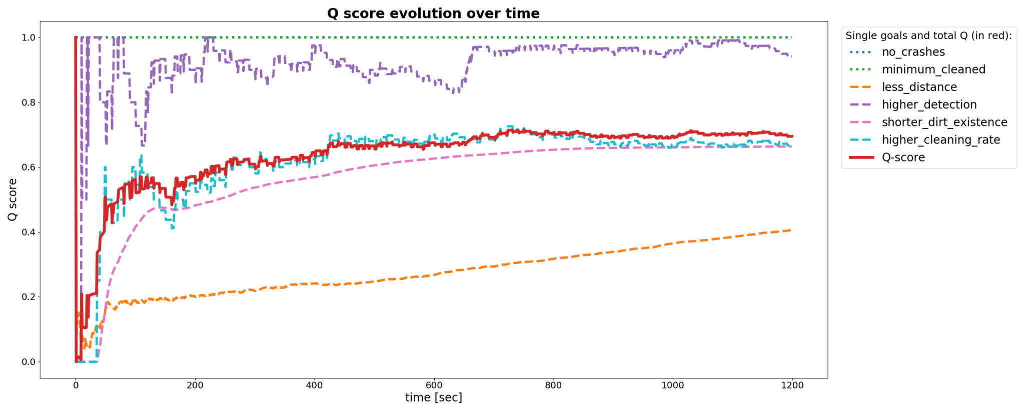


Figure 5.2: *A sample plot created by the evaluation framework that shows the evolution of the Q-score. Implemented by Bergemann [5]*

5.3 Merging the MA-SACPS Subsystem and the Evaluation Framework

In order to use the improved functionalities of the evaluation framework with the MA-SACPS subsystem, these subsystems had to be merged. These improvements include calculation of the Q-score, easy configuration, extended visualizations and more evaluation graphs. Even though the evaluation framework was designed as a wrapper for the MA-SACPS subsystem, the scope of Bergemann's implementation at that time did not include some of the MA-SACPS subsystem's functionalities.

In the scope of this thesis, we merged the evaluation framework with the MA-SACPS by implementing the missing functionalities. These additions can be summarized in two major areas. The first improvement was allowing the framework to collect the data generated in Neuss's thesis. This was done by creating a new ROS node called `Rosbag Recorder`. Details of this ROS node can be found in Section 5.3.1. The second improvement introduced in the scope of this thesis was adding the generation of false

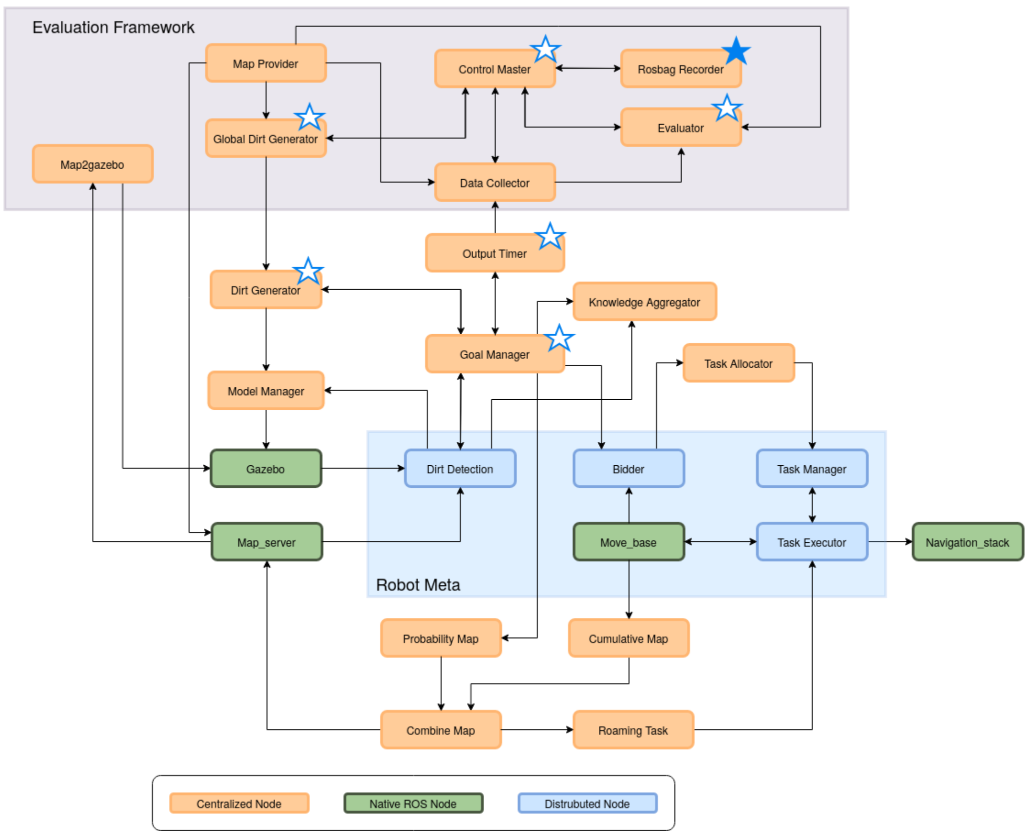


Figure 5.3: *ROS nodes of the merger between evaluation framework and the MA-SACPS. The filled blue star implies the node was added during this thesis. The outlined blue stars implies a previously created node was modified. Modified the original image taken from Petrovska et al. [38]*

positive dirts to the evaluation framework. False positive dirt generation is a key aspect when evaluating the knowledge aggregation. Adding this functionality to the wrapper proved to be more challenging than expected. Section 5.3.2 explains the changes made to allow false positive dirt generation and the encountered challenges. Finally, Section 5.3.3 introduces the false positive penalty and how it was implemented. The architecture graph of the newly added `Rosbag Recorder` node and the modified ROS-nodes can be seen in Figure 5.3.

5.3.1 Data Collection

One of the challenges encountered while merging the MA-SACPS subsystem and the evaluation framework was about data collection. The MA-SACPS subsystem dumps the data from the relevant topics into a Rosbag file. Rosbag is a file format used to store messages coming from specific topics for the entirety of the simulation. This storage file is later analyzed via a separate python script to generate the plots. However the evaluation framework collects the data on run-time and generates its results using a ROS node called evaluator. Because of the discrepancies between these subsystems, a new ROS node that collects these rosbags had to be added to the evaluation framework during the merge.

The newly added node is called `Rosbag Recorder`. It is added to the evaluation framework code and it allows recording the same rosbag files the MA-SACPS subsystem records.

It is configurable via the `parameters.yaml` file evaluation framework offers. Currently it has 4 parameters that can be changed to customize the operations of the node.

- `rosbag_data_record`: Boolean flag that determines whether this node should work. If false, rest of the arguments are ignored and the node does not record any data.
- `rosbag_data_record_topics`: Specifies the topics that are wanted to be stored as rosbags. Only relevant if `rosbag_data_record` is true.
- `rosbag_data_record_output_folder`: Specifies the absolute path to the folder rosbags should be stored. Only relevant if `rosbag_data_record` is true.
- `rosbag_data_record_output_name`: Specifies the prefix of the output file name of recorded rosbags. Only relevant if `rosbag_data_record` is true.

5.3.2 False Positive Dirt Generation

The dirt generation was moved to the evaluation framework from the MA-SACPS subsystem in the scope of Bergemann's thesis[5]. While this allowed having a much more extended dirt generation process with different distributions and hotspots, it didn't support the generation of false positive dirts. Generation of false positive dirts was crucial to evaluate the knowledge aggregation and had to be implemented. This functionality was added to the dirt generator of the evaluation framework in the scope of this thesis. There were several challenges encountered while implementing this feature.

The false positive dirts are handled differently in several ways than ground truth dirts. Several changes had to be made to the `Global Dirt Generator`, `Dirt Generator` and the `Goal Manager` ROS nodes. The interactions of all of the changed nodes can be seen from Figure 5.1. The changes will be explained in the following subsections.

Spawn Intervals

The calculation of the spawn intervals for the ground truths and false positives are explained in the `Dirt Generator` subsection of Chapter 3. The spawn interval for the ground truths (T_{TP}) is taken as a simulation parameter in the beginning of the run. The spawn intervals for false positives of each individual robot are calculated from T_{TP} and the false positive probabilities for each robot ($p_{FP,i}$) as shown in Equation 3.4. This logic was implemented in the `Dirt Generator` node in the scope of the thesis of Neuss [34]. However, implementing false positives was not in the scope of the thesis of Bergemann [5]. Thus the spawn intervals which are necessary to generate false positive dirts were not calculated in the `Global Dirt Generator` node of the evaluation framework. As mentioned before, `Global Dirt Generator` overrides the dirt distribution functionality for our thesis, so the logic to calculate false positive spawn intervals had to be added to the `Global Dirt Generator` node.

The coding pattern used in `Dirt Generator` node was reused and added to the `Global Dirt Generator`. As seen in Listing 5.1, the spawn intervals are stored as an array (`self.spawn_intervals`). The array's first element (`self.spawn_intervals[0]`) is the spawn interval for the ground truths, stored as seconds. The false positive spawn interval for the robot i is stored in the element $i+1$ (`self.spawn_intervals[i+1]`). The values in the array correspond to the seconds the timer in the node has to wait in order to spawn the corresponding task.

```python
# global_dirt_generator.py

...
# convert probabilities into timers
fp_spawn_intervals = (1.0 - self.fpp_props) / \
    self.fpp_props * spawn_interval

# Store all spawn intervals in this array.
# The order is ground truth, robot 0, robot 1, ...
self.spawn_intervals[0] = spawn_interval
self.spawn_intervals[1:] = fp_spawn_intervals
...
```

Listing 5.1: *Spawn intervals are added to the* `Global Dirt Generator` *ROS node to allow false positive dirt generation*

GoalManager crash bug

A bug that causes the `Goal Manager` node to crash was discovered during the thesis of Bouzid [6]. This crash would occur sporadically in the middle of a simulation. After debugging the logs, this behaviour was pinpointed to the `Goal Manager` node. In the `Goal Manager` node there is a logic that loops over a dictionary of existing dirts. When a dirt is cleaned while this loop is running, the entire node crashes and shutdowns. This is because a dictionary should not change while it is being traversed. To fix this issue the dictionary of existing dirts is cloned before the loop begins. Only this clone is traversed. Even if a dirt is cleaned in the middle of the loop, the clone is not affected. The code snippet in Listing 5.2 shows how this logic was implemented. After this change, the node allows the dictionary to change during the traversal.

```python
# GoalManager.py

# Reason for the line below is to prevent changes to
#  fp_spawn_times during iteration. If a false positive
#  dirt is cleaned during this iteration, GoalManager
#  will crash. To prevent that we iterate over a snapshot
#  instead of the actual dictionary.

tmp_fp_spawn_times = self.fp_spawn_times.copy()
for task_id in tmp_fp_spawn_times.keys():
  spawn_time = tmp_fp_spawn_times[task_id]
  ...
```

Listing 5.2: *A snapshot of list of the existing dirts is taken to avoid the crash*

GoalObject and DirtObject

There are two different Python objects to represent a goal in the MA-SACPS subsystem. This is a result of the evolution of the system. These objects are called `DirtObject` and `GoalObject`. DirtObject is always a ground truth while the GoalObject has a field called fp that holds whether the dirt is a ground truth or a false positive. During the implementation of the evaluation framework false positives were not prioritized. The implementation goal of the framework was to be an abstract wrapper that can support different simulators. Because of this dirts were represented with DirtObjects. However, for this thesis we require spawning false positives from the evaluation framework. Therefore, the usages of DirtObjects were replaced with GoalObjects that can handle false positives (Listing 5.3). The implication of this was breaking the support for the simulators that does not handle false positives. This limitation is explained further in Section 7.2.

```
# global_dirt_generator.py

...
pose = Pose(position=r_position,
 orientation=Quaternion(x=0.0, y=0.0, z=0.0, w=1.0))

# This object and its publisher is changed
#  to use GoalObjects instead of DirtObjects
#  to support false positives.

# Please note that this probably breaks
#  the support for the custom simulator.

dirt = GoalObject(index, pose, trust_value, [])
...
```

Listing 5.3: *DirtObject used in global_dirt_generator.py is changed to GoalObject*

Modifications to the Dirt Generator node

As explained before, the `Global Dirt Generator` node overrides the calculation of where the dirts should be spawned. It allows having more complicated dirt distribution patterns like hotspots. However the `Dirt Generator` is still responsible for adding the dirt to Gazebo. As explained in the subsection above, all usages of DirtObject was changed to GoalObjects in the `Global Dirt Generator` node. As the actual generation happens in the `Dirt Generator` node, the usages of DirtObjects had to be changed to GoalObjects in this node as well.

In addition to this change, the logic to add the new dirt to Gazebo was simplified as well. Adding a new dirt to Gazebo, or the dirt generation process as called in the code base, is handled by two different functions. One of the functions handles the dirts that are created by the `Global Dirt Generator` node of the evaluation framework, while the other function handles the dirts created by the legacy `Dirt Generator` node. These two functions mostly had the same logic with the difference that one of them supported false positive dirts as well. As this difference was removed in the scope of this thesis, the function that handles the tasks coming from the evaluation framework was reworked to reuse the other function. This simplified the code and allowed a more maintainable and easy to extend system. The change can be seen from the code snippet in Listing 5.6.

```python
# DirtGenerator.py

def __spawn_dirt_from_wrapper(self, dirt: GoalObject):
  # Spawns dirt object in the map at the position
  #  which was generated for the dirt delivered via the
  #  input parameter of this function. The dirt object
  #  comes from the topic published by
  #  the evaluation framework

  duplicate, fp_duplicate = self.__check_for_duplicates(
    dirt.pose.position,
    dirt.fp)

  self.__spawn_dirt(dirt, fp_duplicate)
```

Listing 5.4: *__spawn_dirt function of the Dirt Generator node is reused to handle adding the dirts coming from the evaluation framework into Gazebo*

5.3.3 False Positive Completion Penalty

Another idea tested with this thesis was how cost of the operation, dirt cleaning in our case, affects the performance of the knowledge aggregation. In the previous state of the system dirt cleaning was a 0-cost operation. Robots instantly cleaned any dirt they reached. However, this is not the case for most of the real-world operations. Cyber-physical systems spend considerable amount of resources and time while completing their tasks. This was represented in our extended system by introducing a penalty on the amount of false positive dirts cleaned.

We introduced a new score function for penalizing the completion of false positives. As seen in Equation 5.2, the score function takes the ratio of the completed false positive dirts ($n_{FP,completed}$) over the number of spawned false positives ($n_{FP,spawned}$). The calculated ratio is subtracted from 1 to make it a penalty. This new score function was added to the helper script(score_functions.py) of the `evaluator` node. The implementation of this new score function can be seen in Listing 5.5. This new score function was used in addition to the old `higher_cleaning_rate`(q_1(t)) score function of the thesis of Bergemann [5]. The `higher_cleaning_rate` score function used in the thesis of Bergemann [5] used the ratio of completed dirts over the total number of spawned dirts. This was not suitable for us, as the total number of spawned dirts includes false positives as well. This was changed to use only the ratio of number of completed ground truths ($n_{TP,spawned}$) over the number of spawned ground truths ($n_{TP,spawned}$). Equation 5.1 shows the calculation of this score function(q_1(t)). The implementation of the `higher_cleaning_rate` score function can be seen from Listing 5.4. These two score functions were summed with their corresponding weight functions to calculate the final Q-score (Equation 5.3).

$$q_1(t) = \frac{n_{TP,completed}}{n_{TP,spawned}} \tag{5.1}$$

$$q_2(t) = 1 - \frac{n_{FP,completed}}{n_{FP,spawned}} \tag{5.2}$$

$$Q(t) = w_1 * q_1(t) + (1 - w_1) * q_2(t) \tag{5.3}$$

```python
# score_functions.py

def function_0(portion_number, total_amount):
  if portion_number is None or total_amount is None:
  # if input is missing (can be the case in the beginning)
    return 1.0
  elif total_amount == 0:
    # it is not possible to have a portion of nothing
    return 1.0
  elif remaining_dirt_number < 0 \
      or total_spawned_dirt_number < 0 \
      or remaining_dirt_number > total_spawned_dirt_number:
    return 0.0  # this should not be possible
  else:
    return portion_number / total_amount
```

Listing 5.5: *function_0 from [5] is used to calculate the ratio of completed ground truths*

```python
# score_functions.py

def function_7(gt_finished,
               total_finished,
               gt_spawned,
               total_spawned):
  # Penalize completion of False positives

  total_amount = total_spawned - gt_spawned
  portion_number = total_finished - gt_finished
  if portion_number is None or total_amount is None:
    # if input is missing
    return 1.0
  elif total_amount == 0:
    # it is not possible to have a portion of nothing
    return 1.0
  elif portion_number < 0 \
      or total_amount < 0 \
      or portion_number > total_amount:
    return 0.0  # this should not be possible
  else:
    return (total_amount - portion_number) / total_amount
```

Listing 5.6: *A new score function, function_7, is created to calculate the false positive completion penalty*

Previously the system did not differentiate when calculating the numbers of completed or spawned ground truths and false positive dirts. In order to calculate the score functions above, the differentiation had to be made. There were couple of changed made to the `Goal Manager` and `Output Timer` nodes in order to achieve this.

A new ROS publisher called `goal_obj_attained` was added to the `Goal Manager` node. This new publisher published a GoalObject whenever a goal was completed (Listing 5.7).

```python
# GoalManager.py

for _goal in self.all_tasks:
  if self.__compare_poses(_goal.pose, goal_pose):
    self.all_tasks.remove(_goal)
    self.goal_obj_attained_pub.publish(_goal)
    # if task is FP also remove timer from dictionary
    if len(_goal.fp) > 0:
      self.__delete_spawn_times_entry(_goal.id)
```

Listing 5.7: *New ROS publisher in the Goal Manager publishes a GoalObject whenever a goal is attained.*

`Output Timer` is the node that is used to pass information from the MA-SACPS subsystem to the evaluation framework as explained in section 3.3. A new subscriber was added to this node that listens to the newly added `goal_obj_attained` publisher. The callback function that is shown in Listing 5.8, subscribes to the publisher and determines whether the goal is a false positive or not. If the goal is a ground truth, a counter is incremented.

```python
# output_timer.py

def spawned_goal_gt_callback(message):
  global spawned_goal_gt_counter,
  current_undetected_dirt_locations,
  all_dirt_ever

  # Increase the counter if goal is ground truth
  if len(message.fp) == 0:
    spawned_goal_gt_counter += 1
    spawned_goal_gt_pub.publish(
      Int32(data=spawned_goal_gt_counter)
    )
```

Listing 5.8: *The subscriber callback function that increments a counter whenever the received goal is a ground truth. The counter is published for the use of the evaluation framework score functions*

This counter is published to the evaluation framework and used in the calculation of the score functions explained above.

5.4 New Experimental Tooling

One of the biggest contributions of this thesis is the introduced new experimental tooling system. Section 5.3.1 will explain why a new tooling system was needed. Section 5.3.2 will describe the design requirements of the system. Section 5.3.3 will introduce the new experimental tooling system and show what architectural decisions was made to fulfill the requirements. Rest of the subsections will do a deep-dive in individual components of the new tooling system and explain them in detail.

5.4.1 Motivation

As explained in the Chapter 3, the system grew into a complex and integrated system as the result of several projects and theses. The current version of the system consists of dozens of ROS nodes that works in harmony. Each ROS node requires several parameters. Configuring the whole system requires a lot of manual effort. Evaluation framework developed by Bergemann [5] simplifies this process greatly by introducing a master configuration file. However this configuration file exists in the simulation level. There was no tool for configuration in the experiment level. As explained further in section 5.1, the simulations refer to the individual runs of the system while experiments refer to a collection of simulations.

Previously, running the experiments required a lot of manual effort. Main motivation of this new tooling is to automate the steps as much as possible and reduce the time it takes to run the experiments. Each simulation runs for 45 minutes. Considering each experiment contains many simulations and that there is a huge parameter space that is being explored, it was quickly realized that running the system locally in one computer was not feasible. Running all of the experiments take thousands of hours. If we decided to run all of the experiments locally it would have greatly reduced the scope and the success of this thesis. That's why this new tooling was designed from scratch to allow running the experiments in parallel. It later evolved to a complete end-to-end system that supports configuration, storage of results, debugging and analysis of the collected data.

5.4.2 Design Goals

There were several design goals when designing the new experimental tooling system. The decided requirements can be seen in list 5.3.1.

R.1 The tooling should allow running simulations in parallel.
R.2 The simulation should be self-contained and should not need a complicated installation process.
R.3 The simulation should run on any suitable computer with minimal effort.
R.4 The tooling should handle scaling up or down the number of parallel simulations seamlessly.
R.5 The tooling should allow easily adding new experiments.
R.6 The tooling should allow running evaluations for the collected data with minimal manual effort.

R.7 The system should allow easily sharing the generated vast amount of data.
R.8 The entire system should be version controlled.

5.4.3 System Architecture

In order to satisfy the design requirements in subsection 5.4.2, a novel system architecture was designed from scratch. The proposed system automates all of the steps of the experimentation process and allows running the simulations in parallel.

The proposed system consists of five subsystems. These subsystems are named `Experiment Designer`, `Cloud Storage`, `Task Queue`, `Worker Containers` and the `Analysis Subsystem`. They will all be explained in detail in their respective subsections.

These subsystem are decoupled from each other and they all each handle a different function of the overall system. The configuration functionality is handled by the `Experiment Designer` script, which will be explained in detail in subsection 5.4.7. The results are stored using `Cloud Storage` that is explained in subsection 5.4.6. The execution happens in `Worker Containers`, which is the merged system explained in section 5.3 converted into a Docker container with all of its dependencies. The containerization process is explained in section 5.4.5. The orchestration needed by the parallelization is handled by the `Task Queue` tool, which is explained in subsection 5.4.8. Lastly, evaluation of the generated results are done by a collection of scripts that are named the `Analysis Subsystem`. The `Analysis Subsystem` ties the whole system together by downloading the output of the simulations, evaluating the data and generating the plots which are the final output of the system.

The new experimental tooling system has two main sequences. These are the configuration sequence and the execution sequence.

The first sequence is the configuration sequence which is triggered whenever the user runs the `Experiment Designer` tool. As seen in Figure 5.4, the user first configures the experiment with the config.json file and runs the experiment_designer.py Python script. This script reads through the configuration and generates the parameters.yaml files necessary to configure the individual runs. These parameter.yaml files are uploaded into a `Cloud Storage` bucket with a unique folder name. This folder name is generated by concatenating the experiment name with the run id. The details of this process is explained in subsection 5.4.7. Lastly, this unique folder name is added to the task queue by calling the /add endpoint of the `Task Queue`.

The second sequence is the execution sequence which is triggered whenever a `Worker Container` is restarted. The sequence is visualized in Figure 5.5. On startup, the worker container queries for a task label name from the `Task Queue` using the /take endpoint. `Task Queue` responds with the label name of an uncompleted simulation. This label name corresponds to a unique folder in the `Cloud Storage`. When the container receives this task label name, it downloads the parameters.yaml file from the correct `Cloud Storage` folder. The container then copies this parameters.yaml file to the correct location and executes the simulation with the given parameters. At the end of the simulation, the results are put into a folder called `evaluation_output`. This folder is compressed and uploaded to the same `Cloud Storage` folder. The container restarts after this file is

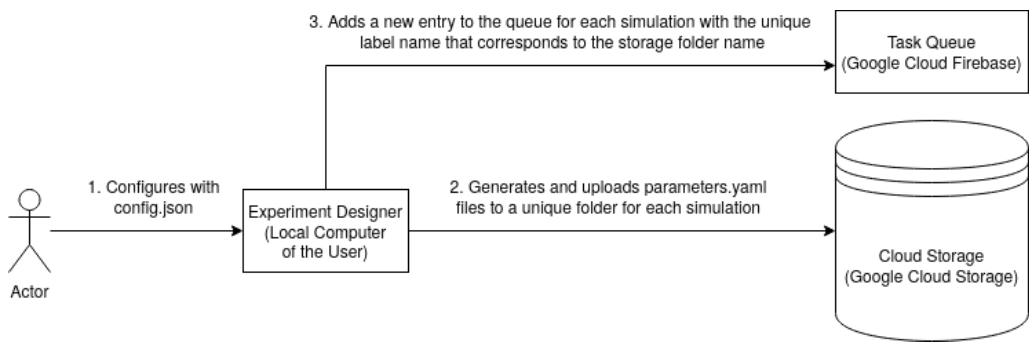

3. Adds a new entry to the queue for each simulation with the unique label name that corresponds to the storage folder name
Task Queue (Google Cloud Firebase)
1. Configures with config.json
Actor
Experiment Designer (Local Computer of the User)
2. Generates and uploads parameters.yaml files to a unique folder for each simulation
Cloud Storage (Google Cloud Storage)

Figure 5.4: *The diagram shows the sequence of the actions that runs during the configuration of the experiments. This sequence runs whenever the user executes the experiment designer script*

uploaded successfully, and starts this sequence again. The details of the containerized system can be found in section 5.4.5.

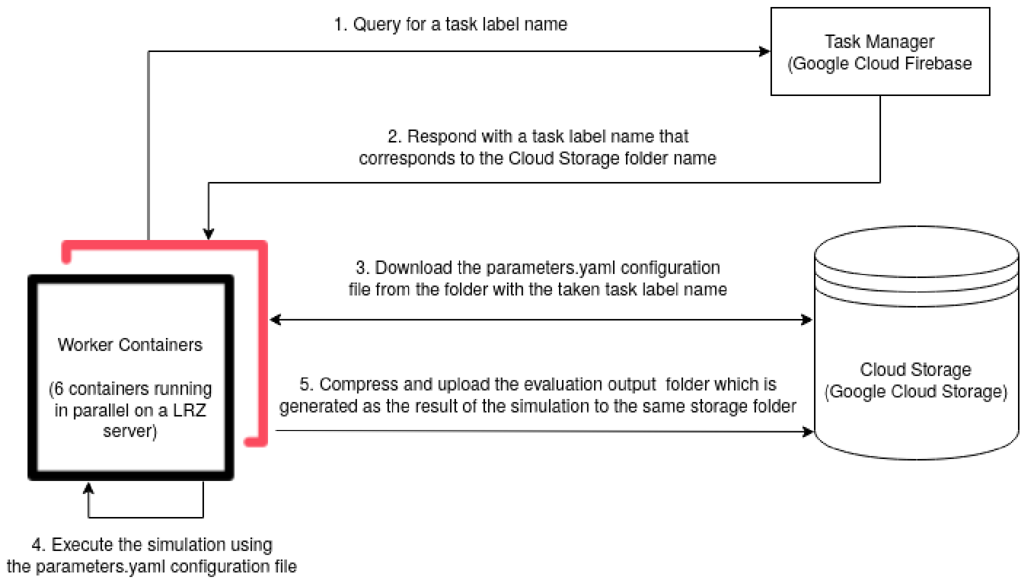

1. Query for a task label name
Task Manager (Google Cloud Firebase
2. Respond with a task label name that corresponds to the Cloud Storage folder name
3. Download the parameters.yaml configuration file from the folder with the taken task label name
Worker Containers
(6 containers running in parallel on a LRZ server)
5. Compress and upload the evaluation output folder which is generated as the result of the simulation to the same storage folder
Cloud Storage (Google Cloud Storage)
4. Execute the simulation using the parameters.yaml configuration file

Figure 5.5: *The diagram shows the sequence of the actions that runs during the execution of the experiments. This sequence is triggered whenever a worker container restarts. The worker containers are configured to restart whenever a simulation is finished, so this sequence runs continuously.*

5.4.4 Worker Containers

As explained in detail in section 5.3, the MA-SACPS subsystem and the evaluation framework was merged in the scope of this thesis. The merged system was turned into a Docker container with all of its dependencies in order to satisfy requirement R.2 and R.3. This containerization step makes the system self-contained as the whole system can be installed just by loading a Docker image (R.2). The containerization also allows running the system in any computer that runs Ubuntu 18.04 and has Docker installed (R.3). This simplification of the installation process allows easily running the simulation in many different computers. Docker mostly isolates the container from the rest of the computer, thus we are also able to run multiple simulations on the same computer.

The containerization of the simulation system was made possible with the parameters.yaml file introduced in the thesis of Bergemann [34]. This file provides a single point of entry for the configuration of the system which was not possible before.

There were couple of issues encountered while creating the first Docker container that contains the merged system. The first challenge was resolving the absolute path to the simulator launch file. This path had to be given inside the parameters.yaml file as seen in Listing 5.9.

```
# parameters.yaml

absolute_path_to_simulator_launch:
  "/thesis_ws/src/original/ros/test/master.launch"
```

Listing 5.9: *The parameters.yaml configuration line that is used to configure the absolute path to the simulator launch file*

This was a challenge as the ROS workspace usually resides inside the Linux home folder and the home folder path depends on the container ID. As the container ID is unique for each worker container, this was very challenging to resolve. The solution to this problem was found by putting the ROS workspace in the Linux root folder. The root folder path does not depend on the container ID. Using this trick, the absolute path to the simulator could be written as `"/thesis_ws/src/original/ros/test/master.launch"` for each container.

The second challenge was due to the limitations of the Docker. Normally, the system uses the graphical user interface of Gazebo to visualize the simulation. However, this was very challenging to achieve and unnecessary while running the system inside a Docker container. This challenge was avoided by running the Gazebo simulator headless. By running Gazebo headless, the simulation system could be run in the background without launching a graphical user interface.

The third challenge of containerizing the system was giving it read and write access to the Google Cloud Platform (GCP). This access was needed as the container had to send a query to get the label name from the `Task Queue` as explained in section 5.4.3. Furthermore, the container had to download the parameters.yaml configuration file from the `Cloud Storage` and upload the results after the simulation was done. All of these actions require permission to access the private GCP project. This challenge was resolved by creating a service account from GCP. The service account was used by all of the containers and had read/write access to the necessary GCP services. Access token of the service account was added to the container image. The containers uses the access token to authenticate on startup.

After these challenges were handled, the container could be run successfully. As we wanted to minimize the manual effort, it was decided that the worker containers should run continuously. In order to achieve robustness and scalability, the containers were designed as stateless. The containers run a script called `run_sim.sh` every time they restart. This script handles the authentication, querying the `Task Queue` and receiving the label name, downloading the parameters.yaml file from the `Cloud Storage`, copying it to the correct location, setting up the ROS environment, executing the simulation, compressing the results and uploading them back to the `Cloud Storage` (Listing 5.10).

```
# run_sim.sh

#!/bin/bash
gcloud auth activate-service-account \
  container-user@i4-tamer-thesis.iam.gserviceaccount.com \
  --key-file=/i4-tamer-thesis-18e6e929bad2.json

RUN_ID=$(curl https://us-central1- \
  i4-tamer-thesis.cloudfunctions.net/take)

gsutil cp gs://i4-thesis-multi-agent-data/ \
  $RUN_ID/parameters.yaml parameters.yaml

mv parameters.yaml /thesis_ws/src/original/ \
  ros/src/evaluation-framework-code/launchers/config/

source /thesis_ws/devel/setup.bash
export TURTLEBOT3_MODEL=burger

roslaunch launchers main_launcher.launch

mkdir /thesis_ws/src/original/ros/src/ \
  evaluation-framework-code/evaluation_output/logs

cp -r ~/.ros/log/latest /thesis_ws/src/ \
  original/ros/src/evaluation-framework-code/ \
  evaluation_output/logs/

tar -czvf results.tar.gz \
  /thesis_ws/src/original/ros/ \
  src/evaluation-framework-code/evaluation_output

gsutil cp results.tar.gz \
  gs://i4-thesis-multi-agent-data/$RUN_ID/results.tar.gz
```

Listing 5.10: *The bash script that orchestrates the operations that run inside the worker container. The script runs when the Docker container is booted up. The script handles authentication, querying the Task Queue, downloading the parameters.yaml file, executing the simulation, compressing and uploading the generated results*

The `run_sim.sh` bash script is made to run every time the Docker container is loaded, by defining it as such in the Dockerfile. The base Docker image that contains the merged system, all of its dependency libraries, the GCP authentication access token and the run_sim.sh bash script. The base image is used to create another Docker image with the Dockerfile in Listing 5.11.

```
# Dockerfile

FROM full-deal-self-adaptive-robotics-system:v3.0
CMD /run_sim.sh
```

Listing 5.11: *The Dockerfile that creates the automatic Docker image that runs run_sim.sh whenever it is loaded. As seen in the snippet different versions of the base image can be used to create the final image. For this sample, v3.0 of the base image was used. The versioning is explained in detail in subsection 5.4.9*

The new image is created and pushed to DockerHub by running the following command in the folder that contains the Dockerfile.

```
sudo docker build -t tamertemizer/ \
  full-deal-self-adaptive-robotics-system:v3.0 .

sudo docker push tamertemizer/ \
  full-deal-self-adaptive-robotics-system:v3.0
```

Listing 5.12: *The first command is used to build the Docker image that runs the run_sim.sh automatically. That command should be called from the directory of the Dockerfile. The second command pushes the created Docker image to DockerHub to be versioned. The versioned image can be pulled from any computer connected to the Internet.*

The new image always runs the run_sim.sh bash script whenever it is loaded. As explained above run_sim.sh script handles all of the necessary operations within the worker container. The container is run using the following command.

```
sudo docker run -d --restart always tamertemizer/ \
  full-deal-self-adaptive-robotics-system:v3.0
```

Listing 5.13: *The command that is used to start and run the worker containers continuously.*

5.4.5 Choosing the Cloud Provider

Several cloud providers were investigated to find the best infrastructure to run the simulations. The considered options were running the experiments Google Cloud Platform (GCP), Microsoft Azure and LRZ Computation Cloud. These providers were benchmarked for cost, available computation resources, ease of integration with rest of the system, availability and scalability.

Minimizing the cost was a necessary requirement. Because of this only providers with free trials were benchmarked. GCP offers 300 dollars worth of free computation resources for 3 months [18] while Azure offers 200 dollars of free computation resources for 2 months. The trial accounts can be upgraded to paid ones during or after the trial. On LRZ Computation Cloud, our group had a quote of virtual CPUs that can be assigned to virtual machines.

As explained on system architecture and worker containers subsections, simulations run in separate Docker containers on Ubuntu 18.04. The provider had to support this functionality. Google Cloud Platform has the Google Kubernetes Engine (GKE) which allows running individual containers. Azure also provides this functionality via Azure containers. Both providers allow changing the CPU architecture, number of assigned CPU cores and adding the desired GPU. Increasing the allocated resources increase the cost. Free trials on GCP and Azure introduces quotas on available resources. Both of them allow having only 1 GPU core for the free trial version. Both of them have a quota that allows running 8 virtual CPU cores at most at the same time.

In the end, Google Cloud Platform was chosen for the `Task Queue` (Section 5.4.8) and `Cloud Storage` (Section 5.4.6) subsystems of the new experimental tooling. The decision to put the `Cloud Storage` in GCP was made after realizing storing vast amounts of data is inexpensive and the free trial of 300 dollars was more than enough for the duration of this thesis. `Task Queue` tool was also implemented using Google Cloud's Firebase platform. This decision was made because of the Firebase Cloud Functions functionality. Using Cloud Functions a serverless backend can be implemented very easily and with minimal cost [13]. The details of the `Task Queue` and the design choices made while implementing the tool can be found in Section 5.4.8.

For the worker containers, using the resources of GCP was deemed infeasible. Even though storage is cheap in GCP, computation with the resources that we need was expensive. Furhermore, the quotas imposed by the free trial of GCP, severely limited the number of parallel simulations. As mentioned before, our group had access to some suitable servers in LRZ Computation Cloud. One of the servers had `Intel(R) Xeon(R) Gold 6148 CPU` with 2.40GHz clock speed. After running the simulation system in that server with varying number of parallel containers, the best configuration was found. At the end, 6 worker containers was run in parallel in that server in the LRZ Computational Cloud.

5.4.6 Cloud Storage

One of the drawbacks of the old system was that all of the data was dumped into the local storage of the computer that is running the simulations. This is changed with the new tooling because of several reasons.

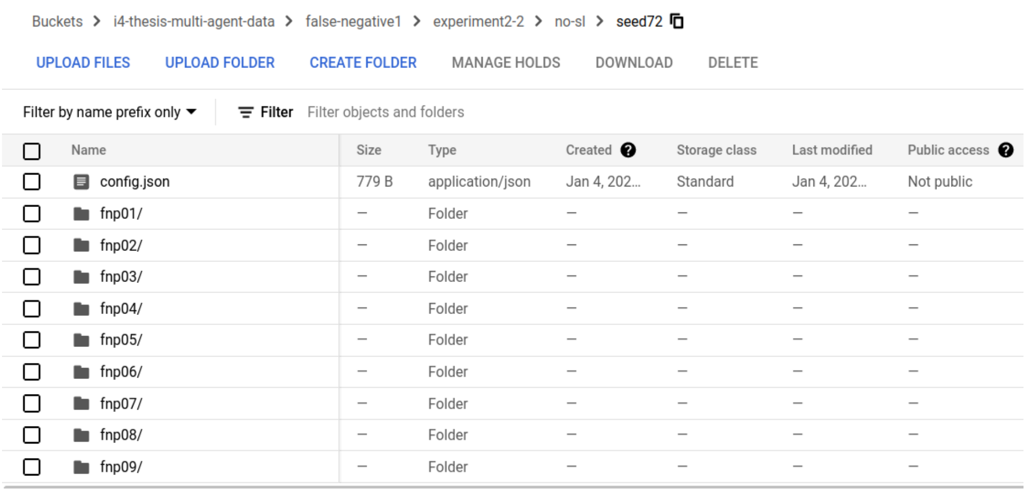
The data generated from the simulations take up a considerable amount of storage space. The data can easily take hundreds of gigabytes with the test runs. It is undesirable to store that data in the computer of the user.

Secondly, with the new tooling, experiments are being run in parallel. According to R.6, the tooling should allow running evaluations for the collected data with minimal effort. If the collected data for a specific experiment exists in separate computers, it is not possible to analyse it without manual effort.

Lastly, according to R.7, the generated results should be shareable between the members of the group. This is hard to achieve when the data is located on the local computer of the user. Given the size of the data, it was hard to share between different users.

Because of these reasons, storing the data in a cloud storage unit was decided. All of the data collected during a simulation run is uploaded automatically to the cloud storage. Google Cloud Storage was chosen to keep the synergy between the other components of the tooling that uses the Google Cloud Platform. Data can be accessed through the web portal of GCP or the `gsutil` CLI tool.

The graphical user interface of the web portal of GCP Cloud Storage can be seen in Figure 5.6. The folder structure is determined as the following. The top level folders are named after the experiment name. In the figure below, the experiment name is false-negative1/experiment2-2/no-sl. This experiment name consists of 3 parts which is configured by the user in the config.json file of the `Experiment Designer` tool. If the experiment name contains forward slash characters (/), `Cloud Storage` separates the subsections into their own folders. In Figure 5.6, the experiment name contains two slashes, which results in three nested folders. This functionality is used to organize the experiments better. After the experiment name, different folders for each seed is created with the format *seed* + *seed_no*. In the figure, inside of the folder seed72 can be seen. This folder contains a different folder for each of the individual simulation runs. These runs are named after the run ids. This folder also contains the config.json file used to create this experiment. The folder that have different run ids (e.g. fnp05) contains the parameters.yaml configuration file for that specific run and the *evaluation_output* compressed into a tarball named results.tar.gz. This folder structure and the configuration files are created by the `Experiment Designer` tool explained in detail in Section 5.4.7. Creation of the tarball is done by the `run_sim.sh` script of the worker containers explained in detail in Section 5.4.4.


Buckets > i4-thesis-multi-agent-data > false-negative1 > experiment2-2 > no-sl > seed72
UPLOAD FILES
UPLOAD FOLDER
CREATE FOLDER
MANAGE HOLDS
DOWNLOAD
DELETE
Filter by name prefix only
Filter
Filter objects and folders

Name	Size	Type	Created	Storage class	Last modified	Public access
config.json	779 B	application/json	Jan 4, 202...	Standard	Jan 4, 202...	Not public
fnp01/	–	Folder	–	–	–	–
fnp02/	–	Folder	–	–	–	–
fnp03/	–	Folder	–	–	–	–
fnp04/	–	Folder	–	–	–	–
fnp05/	–	Folder	–	–	–	–
fnp06/	–	Folder	–	–	–	–
fnp07/	–	Folder	–	–	–	–
fnp08/	–	Folder	–	–	–	–
fnp09/	–	Folder	–	–	–	–

Figure 5.6: *A Google Cloud Storage bucket showing different simulation results being stored. Each folder contains the evaluation_output folder compressed with the name results.tar.gz and the respective parameters.yaml configuration file for that run.*

5.4.7 Experiment Designer

Experiment designer tool is the entry point of the user. It is located in the user's local computer and it is used to configure the experiments. The need for this component comes from the high number of individual simulations that was needed for this thesis. This tool builds upon the easy configuration scheme introduced with the parameters.yaml file of the evaluation framework [3]. The parameters.yaml allows configuring all of the relevant parameters from a single file. A separate parameters.yaml had to generated for every single simulation run. As the parameter space is very large, this required many hours of manual labor. The earlier method used to automate this process was having shell scripts that ran several experiments back-to-back. The script had to be located on the same computer that is running the experiments. The same shell script had to run for hours at a time. If this shell script was interrupted for some reason, the entire experiment had to be run again. Adding new experiments required editing the script which is inconvenient especially when running these experiments in parallel on many machines. Because of these reasons a new tool for configuration called experiment designer is introduced in the scope of this thesis.

Experiment designer tool allows configuration in the experiment level (many runs) instead of the simulation level(1 run). This means a single config.json file is used to generate all the necessary parameters.yaml files for an experiment. These generated parameters.yaml files are uploaded to the cloud storage with the unique label name of experiment-id/run-id. Also the new label name is broadcasted to be added to the task queue.

The available configuration fields can be seen below.

- `experiment_name`: Unique experiment name that will be used for the cloud storage bucket name
- `run_prefix`: Prefix that will be appended after the experiment name and before the value of the independent variable
- `simulation_time`: Duration that the simulation will be run (in seconds)
- `seed`: Simulation seed for dirt generation, can be an array to create tasks for multiple seeds at the same time
- `use_sl`: Boolean flag that determines whether knowledge aggregation with subjective logic should be on.
- `sl_operator`: Subjective Logic operator. Can be CCF, CBF or Comb.
- `adaptive_scheduling`: Boolean flag that determines whether adaptive scheduling should be on.
- `dirt_use_uniform_distr`: Boolean flag that determines whether dirt generation should follow uniform distribution or hotspots.
- `spawn_interval`: The time interval in which a new dirt will be spawned (in seconds)
- `variables`: JSON object that holds all of the changed variables. Explained in detail in the following list.

The variables field is an array of object that has the following fields.

- `name`: Name of the parameter that will be changed. This field should match the name of the parameter in the parameters.yaml config file of the evalation framework.
- `values`: Different values the variable should have. A separate simulation task will be created for each value in this array.

- sim: Boolean flag that determines whether the parameter exist in the parameters.yaml config file or should be passed to the simulator as an external argument. If the parameter can be seen in the paramters.yaml file this field should be set to false.

```
// config.json

{
  "experiment_name": "false-negative-experiment/sl",
  "run_prefix": "fnp",
  "simulation_time": 2700,
  "seed": [71, 72, 73, 74, 75],
  "use_sl": true,
  "sl_operator": "Comb.",
  "adaptive_scheduling": true,
  "dirt_use_uniform_distr": true,
  "spawn_interval": 60,
  "variables": [
    {
      "name": "robot_0_fnp",
      "values": [0.1, 0.2, 0.3, 0.4, 0.5],
      "sim": false
    },
    {
      "name": "robot_1_fnp",
      "values": [0.1, 0.2, 0.3, 0.4, 0.5],
      "sim": false
    }
  ]
}
```

Listing 5.14: *A sample experiment_designer config file that modifies the false negative probabilities of the two robots at the same time*

5.4.8 Task Queue

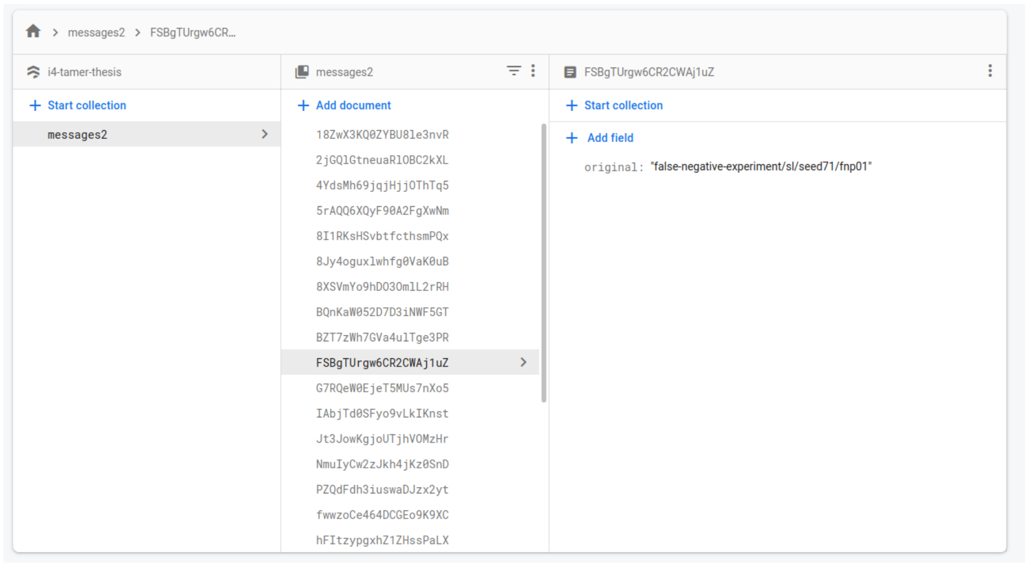
With the simulation containerized and ready to be deployed in parallel, the need for configuring each container with its own parameters.yaml file arised. Experiment designer tool created the parameters.yaml files for each simulation but there were some decisions that were made on how to send the parameters to the containers itself. There were three main approaches explored in this project.

First one was creating the parameters.yaml file via the experiment designer tool and creating each docker container by giving the parameters.yaml file as a docker input. Even though this method was the most straightforward one, it had its limitations. The biggest limitation came while trying to satisfy R.3 and R.4 at the same time. R.3 states that the experiment should run on any suitable computer with minimal effort. This would imply that the system should support all major cloud providers and local computers. However all of these systems have different command line tools to create docker containers. Creating a container on Google Cloud and a local computer needs different steps. This would mean the experiment designer had to know which kind of system will run the simulations. This made R.4. very challenging. Because according to the requirement, the tooling should allow scaling up or down the number of parallel simulations seamlessly. This is very easy to do if the exact same container is replicated. However if each container requires a unique step of setting up the parameters when creating, this is much more challenging.

Trying out the first approach led to the realization that copying the parameters.yaml file to each container is not a very suitable way for our requirements. For the second approach the parameters.yaml file was put into the cloud storage with a unique directory name of experiment-id/run-id. Only this unique id was passed into the docker containers on start-up as an environment variable. When the container received this unique id, it downloaded the parameters.yaml file from the correct directory of the cloud storage. Although it was clear that this approach was better than the first one, still there was a unique step for each container which made it impossible to replicate the same container.

At the end, the most suitable approach was using a small web server called the `Task Queue`. The parameters.yaml files were put into the cloud storage with a unique id like the second approach. At this third approach, the docker containers send a request to the `Task Queue` backend and ask for an available task. If there are any tasks in the queue, the server responds with the unique id. The container downloads the correct parameters.yaml file using this id. Implementation of the worker containers is explained in detail in Section 5.4.4. With this approach, the need to configure each container individually was gone. The same container can be replicated many times which makes the whole system very scaleable. The state of remaining tasks only needs to exist in the task queue.

The task queue is a serverless backend implemented with Google Firebase. It uses Firestore as its database and Cloud Functions for the functionality.



```
messages2

18ZwX3KQ0ZYBU8le3nvR
2jGQlGtneuaR1OBC2kXL
4YdsMh69jqjHjjOThTq5
5rAQQ6XQyF90A2FgXwNm
8I1RKsHSvbtfcthsmPQx
8Jy4oguxlwhfg0VaK0uB
8XSVmYo9hD030mlL2rRH
BQnKaW052D7D3iNWF5GT
BZT7zWh7GVa4ulTge3PR
FSBgTUrgw6CR2CWAj1uZ
G7RQeW0EjeT5MUs7nXo5
IAbjTd0SFyo9vLkIKnst
Jt3JowKgjoUTjhVOMzHr
NmuIyCw2zJkh4jKz0SnD
PZQdFdh3iuswaDJzx2yt
fwwzoCe464DCGEo9K9XC
hFItzypgxhZ1ZHssPaLX
```

```
original: "false-negative-experiment/sl/seed71/fnp01"
```

Figure 5.7: *View of the task queue in the Google Firestore web dashboard showing different simulation labels that will be executed by the worker containers*

The backend has two endpoints exposed. These endpoints are implemented using Cloud Functions. The first endpoint /add inserts a new task to the queue. It is called by the experiment_designer tool.

```
// Take the text parameter passed to this HTTP endpoint
//  and insert it into Firestore under the path
//  /messages/:documentId/original

exports.add = functions.https.onRequest(
  async (req, res) => {
    // Grab the text parameter.
    const original = req.query.text;
    if(original && original.length > 0 )
    {
      // Push the new message into Firestore
      //  using the Firebase Admin SDK.
      const writeResult = await firestore.
        collection('messages').add({original: original});
    }
    else
    {
      res.json({result: ‘Fill text parameter.‘});
    }
});
```

Listing 5.15: */add endpoint of the Firebase task queue. Used for adding a new simulation task to the task queue.*

The second endpoint /take pops a task from the queue. When this endpoint is called, it returns a task label and removes that label from the queue. It is called by the worker containers.

```
exports.take = functions.https.onRequest(
  async (req, res) => {
    const messagesRef = firestore.collection('messages');
    const query = await messagesRef.limit(1).get();
    query.forEach(doc => {
      const data = doc.data();
      res.send(data.original);
      doc.ref.delete();
    });
});
```

Listing 5.16: */take endpoint of the Firebase task queue. Used for taking a task label from the queue. The returned label is removed from the task queue.*

5.4.9 Version Control

One of the requirements was allowing version control for the entire system. This was needed to allow rapid changes for the experiments. All of the code base resides in a Git repository. Also the different versions of the containers are stored using DockerHub. Using DockerHub for the containers allow versioning of the Docker images. As the Docker images are layered, small changes to the system can easily be integrated to all of the worker computers without the need to download the entire image every time [11].

5.4.10 Debugging

The new tooling also stores the logs generated by the ROS nodes and console logs. All generated logs are dumped into their separate files and saved with the rest of the results in the evaluation_output folder. The logs can be examined manually in the case of a failure. Debugging is easier because of this functionality, as the logs indicate where an error might have happened. As is the case with most software systems, many runs have to be made to understand the issue. Considering each run of the simulation takes 45 minutes, this is a very time consuming task. With this added logging function, many simulations can be run in parallel to speed up the debugging process.

5.4.11 Analysis

The new tooling automates data analysis process as well. The analysis step extracts the data from the cloud storage, extracts it, formats it and generates the related plots. The analysis component of the new tooling builds upon the data_analysis python script used in the thesis of Neuss [34].

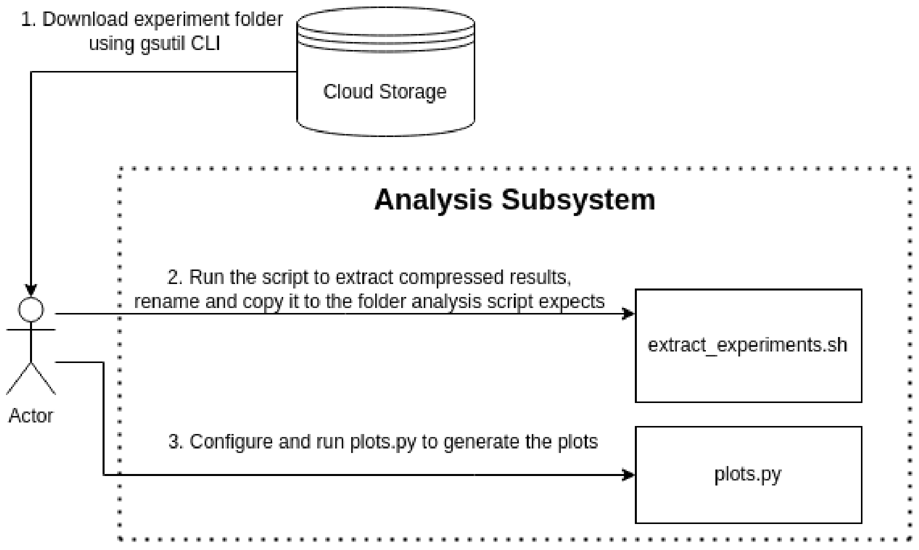

1. Download experiment folder using gsutil CLI
Cloud Storage
Analysis Subsystem
2. Run the script to extract compressed results, rename and copy it to the folder analysis script expects
extract_experiments.sh
Actor
3. Configure and run plots.py to generate the plots
plots.py

Figure 5.8: *The diagram shows the sequence of running the experiments using the analysis subsystem. First the user downlaods the folder containing all of the simulation results from the Cloud Storage. Then they run the extract_simulations.sh shell script shown in Listing 5.17. Finally the user runs the plots.py to evaluate the results and generate the plots.*

The analysis sequence can be seen in Figure 5.8. As seen in the figure ,first the data is downloaded from the correct cloud storage directory to the computer of the user using the `gsutil` CLI tool. The data in the cloud storage resides in tar.gz compressed form. This tarball needs to be extracted to generate the plots. The script shown in Listing 5.17

extracts the compressed results and renames the necessary files according to the naming format required by the data_analysis python script.

```
# extract_experiments.sh

#!/bin/bash

cd false-positive-experiment

ROSBAG_PATH=thesis_ws/src/original/ros/src/ \
 evaluation-framework-code/evaluation_output/ \
 recorded_data/rosbags

RES_SUM_PATH=thesis_ws/src/original/ros/src/ \
 evaluation-framework-code/evaluation_output/results

for SEED in 71 72 73 74 75
do
 for FPP in 1 2 3 4 5 6 7 8 9
 do
  cd sl/seed$SEED/fpp0$FPP/
  echo $PWD
  tar -xvf results.tar.gz
  cp $ROSBAG_PATH/*.bag ../../../../../../ \
   data/r2_sltrue_fptrue_fntrue_spi60_seed${SEED} \
   _r0fpp0.${FPP}_r0fnp0.2_r1fpp0.${FPP}_r1fnp0.2.bag

  cp $RES_SUM_PATH/*.csv ../../../../../../ \
   results/r2_sltrue_fptrue_fntrue_spi60_seed${SEED} \
   _r0fpp0.${FPP}_r0fnp0.2_r1fpp0.${FPP}_r1fnp0.2full.csv

  cd ../../..
 done
done
```

Listing 5.17: *The script that extracts the downloaded results tarball. The script renames the necessary files to the naming format used by the analysis script and copies them to the required folders*

Currently the analysis component allows analyzing the rosbag files explained in section 5.2.1 and the result_summaries generated by the evaluation framework. These files are copied to their respective directories to prepare for the evaluation by the script in Listing 5.17. Finally, data_analysis.py script, implemented in the scope of the thesis of Neuss [34], is run to generate the plots. Also a new script called plots.py is added to allow plotting the change of the adaptivity metric (Q-score) during the experiments. The generated plots are put into a separate folder called `plots` for the user. The generated plots and their commentary is shown in detail in Chapter 6.

6 Evaluation

The goal of this thesis is to evaluate the impact of Knowledge Aggregation (KA) with Subjective Logic (SL) on the overall system's adaptivity. This chapter explains our evaluation approach and results. Section 6.1 introduces the experiments and how they tie to the explored hypotheses. Section 6.2 explains the existing metrics to calculate the Q-score and how they were reworked for this thesis. Finally Section 6.3 shows the results of the experiments and explains how knowledge aggregation impacts the adaptivity of multi-agent cyber-physical systems from various aspects. Section 6.4 summarizes the evaluation results and comments on the benefits and drawbacks of the knowledge aggregation with Subjective Logic.

6.1 Experimental Setup

As explained before the primary goal of this thesis is to evaluate the impact of KA with SL on the overall system's adaptivity. We aim to achieve this goal by varying several key parameters of the system in order to see how they affect the Q-score introduced in Section 2.1.2. The proposed experiments are run using KA with SL and without the KA. The difference in Q-scores of these two sub-experiments (Equation 6.1) give the impact of KA with SL on the overall system's adaptivity. The calculation of the Q-score is described in Section 6.2.

$$Q_{gain} = Q_{SL} - Q_{Base} \tag{6.1}$$

Five research questions were proposed in order to identify and evaluate the key parameters for the impact of KA with SL. These research questions can be seen below.

- **RQ.1** How does changing the false positive probability of the robots affect KA with SL?
- **RQ.2** How does penalizing the erroneous actions impact the performance of KA with SL?
- **RQ.3** How does changing the false negative probability of each robot affect KA with SL?
- **RQ.4** How does the value of threshold affect KA with SL?
- **RQ.5** Is there an optimal threshold value that can be set to maximize the performance of KA with SL?
- **RQ.6** Does changing the dirt distribution patterns affect the performance of the system?

By looking at the research questions defined above, several hypotheses were defined. These hypotheses are matched with experiments in Table 6.1.

Research Question	Hypothesis	Experiment
RQ.1	**H1.1** KA introduces a performance overhead. However, the benefits of KA increase as the false positive probability increases. **H1.2** The average time to complete a TP task fluctuates more as the FP probabilities increase.	False Positive Probability
RQ.2	**H2.1** Benefits of KA become more prominent as the cost of making errors increase.	False Positive Penalty
RQ.3	**H3.1** KA with SL is not the optimal plan of action for cases with high false negative probability and low false positive probability.	False Negative Probability
RQ.4 RQ.5	**H4.1** Higher threshold values increase accuracy. **H4.2** Lower threshold values increase the total number of cleaned tasks.	Threshold
RQ.6	**H5.1** KA benefits from higher overlap between agents.	Dirt Distribution

Table 6.1: *The table shows the summary of hypotheses derived from the research questions presented in List 6.1. The first column indicates the research question, the second column shows the explored hypotheses derived from that research question. The third column indicates the name of the performed to explore the hypotheses. The details of the experiments can be found in the subsections in Section 6.3.*

6.2 Quality Function

Quality function (Q-score) defined in Section 2.1 is used to evaluate the adaptivity of our simulation system. The Q-score and its score functions were implemented in the scope of the thesis of Bergemann [5]. The score functions are evaluated for each time frame by the `evaluator` node of the evaluation framework. The details of this node can be found in Section 3.3. As explained in Section 2.1, all of the score functions are implemented following Petrovska's [37] definition of adaptivity and they give a result between 0 and 1. The score functions are defined for two different categories. These are the business goals and the adaptation goals. The business goals determines whether an agent is satisfying the minimum requirements needed in order to consider that agent as successful. The adaptation goals determine how successfully the agent is adapting in order to maximize its performance. These output values of these adaptation score functions (q_i) are multiplied with a weight (w_i) and summed up in order to calculate the final Q-score (Q). The calculation can be seen from Equation 6.2. N stands for the number of score functions defined for the system. As seen in Equation 6.3 weights (w_i) always sum up to 1. The implementation of the score functions can be seen from Appendix B.

$$Q(t) = \sum_{i=1}^{N} q_i(t) w_i \tag{6.2}$$

$$\sum_{i=1}^{N} w_i = 1 \tag{6.3}$$

6.2.1 Business Goals

The business goals are score functions that determine whether the agent is satisfying the minimum required goals in order to be counted as successful. If one of the business goals are not satisfied, the Q-score automatically drops to 0. There were two business goals defined for our use-case implemented by Bergemann [5]. These were *minimum_cleaned* and *no_crashes*.

Minimum Cleaned

This business goal is used to ensure the agent cleans at least 10% of the spawned dirts. This business goal becomes active only after 5 dirts are spawned in the map. This is done to prevent getting Q-scores of 0 at the beginning of the simulation. The desired ratio of the minimum cleaned dirts can be configured via the parameters.yaml file as seen in Listing 6.1.

```yaml
# parameters.yaml

minimum_cleaned: {
    type: "mission_goal",
    function_index: 5,
    variables: {
      input_to_check: "finished_goal_gt_number",
      offset: 0,
      aim: "spawned_goal_gt_number",
      factor: 0.1,
      start_requirement: 5
    }
  }
```

Listing 6.1: *The configuration field in the paramters.yaml file that allows changing the desired minimum cleaned ratio*

This score function was changed in the scope of this thesis to ignore the false positive dirts. Previously this function would look at the number spawned dirt which included the false positives as well. As false positive dirts should not be cleaned in the ideal scenario, this behaviour was not desired. To fix this, the score function was changed to only use the ratio of the number of completed ground truths ($n_{TP,completed}$) over the number of spawned ground truths ($n_{TP,spawned}$).

No Crashes

This business goal is used to ensure the robots traversing do not crash with each other. If the robots crash with each other for some reason, the Q score drops to 0. This was deemed as a necessary business goal as in the real world the robots should prevent crashes at all costs. Crashes can damage an expensive robot and cause unpredicted consequences in the environment.

6.2.2 Adaptation Goals

The adaptation goals are the score functions that evaluate how well the overall system is adapting for the given problem. The adaptation goals are implemented according to the framework Petrovska proposes in [37]. They map the behaviour of the system to a numerical value between 0 and 1. A value of 0 means the system is performing the worst possible way for the given score function while a value of 1 means the system is behaving perfectly. All of the scores from the adaptation goals are multiplied with their respective weights (w_i) to get the final Q-score as seen in Equation 6.2.

Higher Cleaning Rate

The *higher_cleaning_rate* adaptation goal calculates the ratio of the completed tasks over the total number of spawned tasks. This function is designed to evaluate how well the agents clean the dirts in the map. A value of 1 indicates all of the spawned dirts are cleaned, while a value of 0 would mean no dirt is cleaned at all. This function was implemented during the scope of the thesis of Bergemann [5]. As mentioned previously Bergemann's thesis [5] did not consider the false positive dirts during its implementation. This score function was calculating the ratio using the total number of dirts because of that reason. However higher cleaning rate should only depend on the ground truth dirts, not the false positives. This change was introduced in the scope of this thesis. The final version of this score function calculates the ratio of the number of completed ground truth dirt ($n_{TP,completed}$) over the number of spawned ground truth dirts ($n_{TP,spawned}$). The calculation can be seen from Equation 6.4.

$$q(t) = \frac{n_{TP,completed}}{n_{TP,spawned}} \tag{6.4}$$

Shorter Dirt Existence

The *shorter_dirt_existence* score function uses the summed up actual existence time of all of the dirts and the summed up theoretical dirt existence times. The actual existence time of a dirt ($t_{i,actual}$) is the number of seconds from its spawning to its cleaning. The theoretical dirt existence time ($t_{i,theoretical}$) is the number of seconds between the spawn time frame of that specific dirt to the current time frame in the simulation. The theoretical dirt existence time would be equal to the actual dirt existence time if no dirts were cleaned during the simulation. Both actual dirt existence time and the theoretical existence time is summed for each dirt. This can be seen from the Equation 6.5. In Equation 6.5, N_{dirt} represents the number of dirts spawned.

$$\begin{aligned} T_{actual} &= \sum_{i=1}^{N_{dirt}} t_{i,actual} \\ T_{theoretical} &= \sum_{i=1}^{N_{dirt}} t_{i,theoretical} \end{aligned} \tag{6.5}$$

This score function returns a value of 1 if every dirt is cleaned as soon as they are spawned and a score of 0 if no dirt is cleaned at all. This is achieved by taking a ratio of the summed up actual dirt existence time (T_{actual}) over the summed up theoretical dirt existence time ($T_{theoretical}$) and subtracting the result from 1. The exact calculation can be seen from Equation 6.6.

$$q(t) = 1 - \frac{T_{actual}}{T_{theoretical}} \tag{6.6}$$

Higher Detection

The *higher_detection* score function calculates the ratio of the detected dirts ($n_{detected}$) over the total number of dirts (N_{dirt}). It aims to reward the agents for detecting the maximum number of dirts. The exact calculation can be seen from Equation 6.7.

$$q(t) = \frac{n_{detected}}{N_{dirt}} \tag{6.7}$$

Less Distance

The *less_distance* score function aims to reward the agents to move as less as possible. As moving consumes energy thus introduces costs to the user, moving less is considered as an adaptation goal. If no robots move at all the output of this score function will be 1. For the opposite case of robots moving as much as possible, the output of this function will be 0. The score is calculated from the traveled distance of the robots ($x_{traveled}$), elapsed time ($t_{elapsed}$), maximum possible velocity (V_{max}) and the number of robots (n_{robot}). The calculation can be seen from Equation 6.8.

$$\begin{aligned} x_{max} &= V_{max} \cdot t_{elapsed} \cdot n_{robot} \\ q(t) &= min(1.0, max(0.0, \frac{x_{max} - x_{traveled}}{x_{max}})) \end{aligned} \tag{6.8}$$

Lower Error Rate

The *lower_error_rate* is a new score function introduced in the scope of this thesis. It was added to the existing set of adaptation goals to be able to penalize the completion of false positives. In the current implementation of the system, cleaning the dirts are a 0-cost operation. This means that whenever a robot moves to the center of the dirt, the dirt is cleaned instantly. It was realized that this behaviour does not exist in the real world. In the real world, cleaning is an expensive operation that might take couple of minutes or introduce costs to the system. Also completing false positive tasks might be dangerous, expensive or confusing from the user perspective. Overall, the robot should aim to complete ground truths and avoid false positives. There are couple of ways to enforce this behaviour such as introducing a waiting time after cleaning each dirt or penalizing the completion of false positives in the score functions. Both methods have their advantages and disadvantages. Introducing a new adaptation score function to penalize the completion of false positives were chosen for this thesis. However the other method of introducing a waiting time after completing each task is worth exploring as well. This is discussed in detail in the future works in Section 7.3.

The implementation of this new score function can be found in Section 5.3.3. The *lower_error_rate* score function calculates the ratio of the number of completed false positive dirts over the number of spawned false positive dirts and substracts the result from 1. If all of the false positive dirts are cleaned this score function return 0. In the opposite case of no false positive dirts being cleaned, this function returns a score of 1. The exact calculation can be found in Equation 6.9.

$$q(t) = 1 - \frac{n_{FP,completed}}{n_{FP,spawned}} \tag{6.9}$$

6.2.3 Metric Selection

The score functions used previously was deemed contradictory for this thesis. It was realized that the adaptation goals such as *less_distance*, *higher_detection*, *shorter_dirt_existence* was trying to optimize for contradictory goals or they were unnecessary to evaluate the adaptivity of the system. The business goals were left as is but for the adaptation goals, a simpler set was defined. The new set of score functions consists of two business goals and two adaptation goals.

The two business goals, namely *minimum_cleaned* and *no_crashes* was deemed necessary for our use-case as well and they are reused in this thesis. The only change to the business goals was made to the *minimum_cleaned* score function so that it ignores the false positive dirts. The details can be found above in the respective subsection of the *minimum_cleaned* score function.

The adaptation goal set was reduced to have only two goals. It was realized that the *higher_cleaning_rate* score function is the most important goal for the adaptation. It is believed that the adaptation of the system can be measured to an acceptable degree only by looking at the ratio of the completed ground truths. However this function does not include any information about the false positives. Completion of the false positives is an undesirable behaviour for cyber-physical systems. It might be dangerous, expensive or confusing if the robot chose completing false positive dirts over the completion of ground truths. This realization led to the development of a new score function *lower_error_rate* as explained in Section 5.3.3 and its respective subsection in Section 6.2.2. The final Q-score for our thesis is defined in Equation 6.10. Similarly as before, if one of the business goals are not satisfied, this Q-score drops to 0 automatically.

$$\begin{aligned} q_1(t) &= \frac{n_{TP,completed}}{n_{TP,spawned}} \\ q_2(t) &= 1 - \frac{n_{FP,completed}}{n_{FP,spawned}} \\ Q(t) &= w_1 \cdot q_1(t) + (1 - w_1) \cdot q_2(t) \end{aligned} \tag{6.10}$$

After evaluating the results of Experiment 2, the impact of false positive penalty described in detail in Section 6.3.2, w_1 was chosen as 0.7 for the experiments. The exceptions of this are mentioned in the respective subsections of the experiments and in the table in Section 6.3.

6.3 Experimental Results

As explained in Section 6.1, five experiment were designed. These five experiments map to the research questions and hypotheses as seen in Table 6.1. There were several key parameters that were kept constant throughout the experiments. All of the experiments are done with 2 robots. However this is a design choice, not a limitation of the system. An experiment that changes the number of robots can be found in Neuss [34].

Another parameter that were kept constant was the seed of the simulation. This was changed after the thesis of Neuss [34] to reduce the undesired uncertainties in the system. Instead, the same simulation seed (Seed 72) is run 5 times and the results are averaged out for statistical robustness.

The spawn intervals are kept as a constant of 60 seconds per 1 ground truth and the simulation time is kept as 45 minutes. These values are kept at those values to maintain the comparability with the experiments of Neuss [34] and Bergemann [5]. One important difference that is made to the configuration of the evaluation framework of Bergemann [5] was to change the used map and the map resolution (0.5 meters) to the ones used in Neuss [34].

The first experiment varies the false positive probabilities for all robots at the same time. This means that each robot will always have the same false positive probability as each other. The same pattern is used for Experiment 2 and Experiment 5 as well.

For some of the experiments, the experiment parameter space has more than one dimension. For example Experiment 4 varies the threshold value between 0.2 and 0.8. The same experiment is repeated for false positive probability 0.2 and 0.8 which adds another dimension to the parameters. All of the experiments are run for the KA with SL case and the base case with no knowledge aggregation.

The detailed parameters for all of the experiments can be seen in the table below and the respective subsection of the experiment.

	FP Probability	**FP Penalty**	**FN Probability**	**Threshold**	**Dirt Distribution**
Experiment	1	2	3	4	5
Research Question	RQ.1	RQ.2	RQ.3	RQ.4, RQ.5	RQ.6
Knowledge Aggregation	[Comb , Off]	[Comb , Off]	[Comb , Off]	[Comb , Off]	[Comb , Off]
FP Probability	[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]	[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]	0.2	[0.2, 0.8]	[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
FN Probability	0.2	0.2	[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]	0.2	0.2
FP Penalty Weight	0	[0 , 0.3, 0.5, 0.8]	0	[0 , 0.3, 0.5, 0.8]	[0 , 0.3]
Threshold	0.8	0.8	0.8	[0.2, 0.4 ,0.6, 0.8]	0.8
Dirt Distribution	Uniform	Uniform	Uniform	Uniform	[Uniform, 1 Hotspot, 3 Hotspots]
Spawn Interval (s)	60	60	60	60	60
Simulation Time (s)	2700	2700	2700	2700	2700
Seed	72	72	72	72	72
No of Reruns	5	5	5	5	5

6.3.1 Experiment 1: Impact of False Positive Probabilities

In Experiment 1, the impact of varying false positive probabilities of each robot is explored. The false positive rates for all robots were varied from 0.1 to 0.9. The false positive probabilities of the robots determine the ratio of false positive tasks to total tasks spawned for each robot. As mentioned in Section 3.2, the spawn interval for ground truths are kept as a constant for all of the experiments. The consequence of this design choice is that the increase of the false positive probability translates into the increase of total number of tasks in the map. The spawn interval timer of the false positive tasks can be seen from Equation 6.11. The details of this process can be seen in the Dirt Generator subsection of Section 3.2.

$$T_{FP,i} = \frac{1 - p_{FP,i}}{p_{FP,i}} T_{TP} \tag{6.11}$$

This experiment is run with 2 robots and the robots have the same false positive probabilities. This experiment was designed to test H1.1 and H1.2 shown in Table 6.1. The experiment also aims to answer the RQ.1, “How does changing the false positive probability of the robots affect KA with SL?”.

In the first part of H1.1, we hypothesized that KA with SL would introduce a performance overhead initially. This was hypothesized because KA propagates a dirt as a goal only after a certain number of observations. This slows down the propagation of the tasks and therefore should result in a loss in performance. However KA with SL should also increase accuracy. Thus the second part of H1.1 states that the benefits of KA with SL should increase as the false positive probability of the robots increase. This hypothesis is proved in our experimental results as well. Figure 6.1 shows the gain of KA with SL according to Equation 6.2. Points above the 0-line would indicate benefits of KA with SL compensates and overcomes the performance overhead.

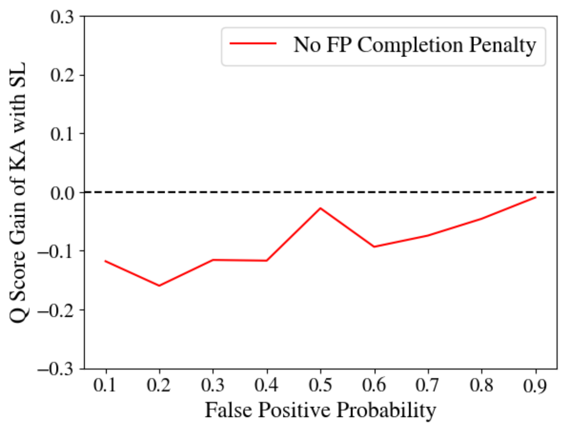


Figure 6.1: *The figure shows gain of KA with SL against the base case of disabled knowledge aggregation. The gain is calculated using Equation 6.2. If the gain is above the 0-line, that indicates that KA with SL is more beneficial to the system than the base case. The results show KA with SL introduces a overhead but this overhead is compensated when the false positive probability increases. This is caused by the increased accuracy gained by KA with SL.*

The figure shows that the base case is notably better in lower false positive probabilities. This is caused by the aforementioned performance overhead caused by the KA with SL. As the false positive probability increases, benefits of KA with SL increases and the line gets closer to the 0-line. However even in very high false positive probabilities the benefits of KA with SL due to increased accuracy is not enough to compensate the initial performance overhead. This result is expected as the false positive penalty or accuracy as an adaptation goal is not considered for this experiment. The cost of cleaning tasks are zero. This was changed in Experiment 2.

We also noted a significant spike in false positive probability of 0.5. This spike should be investigated to eliminate the risks of implementation errors. This behaviour is noted in the threats to validity part of Section 7.2.

The second hypothesis tested in this experiment states that the average time to complete a ground truth task fluctuates more as the false positive probabilities of the robots increase. This was hypothesized for several reasons. The first reason being a higher probability of false positive dirt generation would indicate a higher level of external uncertainty in the system. This would result in more unpredictable results, thus a higher spread in the average time to complete true positives. This is reflected in our experimental results as well. In Figure 6.2, it is seen that the spread of average time to complete a ground truth task tend to increase as the false positive probability increases.

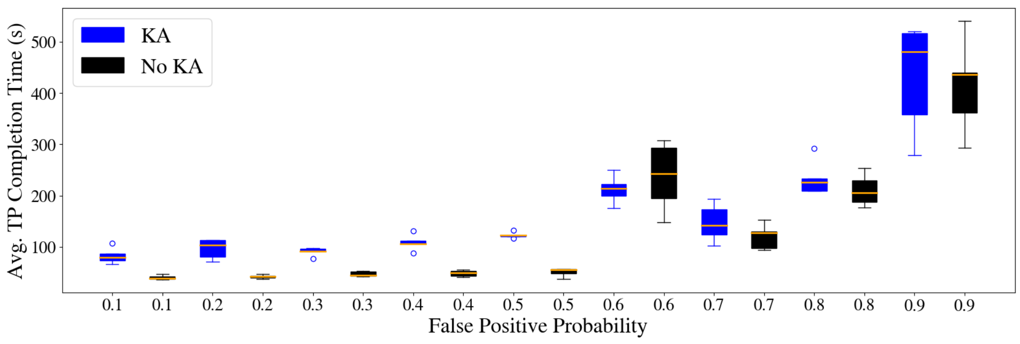


Figure 6.2: *The figure shows the average time to complete TP tasks tend to increase as the false positive probability increases. The first case (blue) represents the scenario where KA with SL is used and the black bars represents the base case. The orange line indicates the median. The results show the spread tend to increase as the false positive probability increases.*

6.3.2 Experiment 2: Impact of False Positive Penalty

For this experiment we reused Experiment 1 with a changed Q-score. As explained in Section 6.2.3, the Q-score calculation was reworked for this thesis. The new score functions and the equation to calculate the Q-score can be seen from Equation 6.12. For Experiment 1, w_1 is taken as 1. This results in only the *higher_cleaning_rate* score function being used. As explained in Section 6.2.3 and 5.3.3, this is not a realistic adaptation goal for our model problem. The cleaning is a 0-cost operation in our model problem, meaning that the robots instantly clean the tasks they touch. However, cleaning or any other operation cyber-physical systems perform in the real world usually have high costs. The operations of a CPS might take a long time to complete or might be expensive. If a CPS chooses a false positive task over a ground truth, it either wastes a lot of time or resources to complete an unnecessary or sometimes dangerous operation. Because of this, accuracy is an important adaptation goal in most real world cyber-physical systems. As explained in previous subsection, this was represented in our system in a new score function called *lower_error_rate*. In this experiment, we change the weights of that score function to explore how it affects the performance of KA with SL.

The equation to calculate the Q-score is repeated on Equation 6.12 for better readability.

$$\begin{aligned} q_1(t) &= \frac{n_{TP,completed}}{n_{TP,spawned}} \\ q_2(t) &= 1 - \frac{n_{FP,completed}}{n_{FP,spawned}} \\ Q(t) &= w_1 \cdot q_1(t) + (1 - w_1) \cdot q_2(t) \end{aligned} \tag{6.12}$$

In order to evaluate the impact of the false positive penalty, w_1 is changed from 1.0 to 0.7, 0.5 and 0.2. This results in the false positive penalty weight $(1 - w_1)$ of 0, 0.3, 0.5, and 0.8. In H2.1, we hypothesized that the benefits of KA would become more prominent as the false positive penalty weights increase. This was indeed represented very clearly in our experimental results. As seen in Figure 6.3, when the false positive penalty weight increases, the performance of KA with SL increases significantly. This is caused by the increased accuracy KA with SL brings.

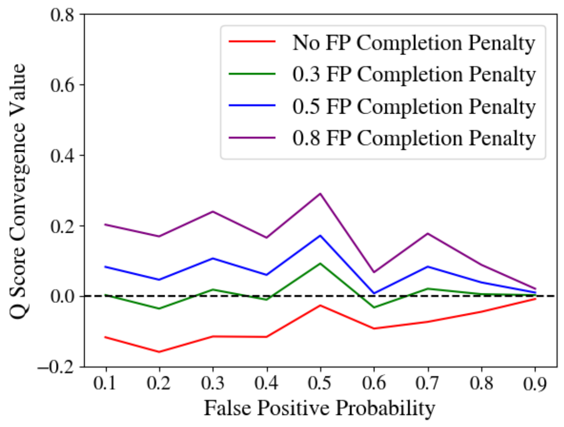


Figure 6.3: *The figure shows the Q-score gain of using knowledge aggregation for varying false positive probabilities with different penalty weights introduced. The gain is calculated by $Q_{SL} - Q_{NO-SL}$. The false positive completion is penalized according to the Q-score calculation in Equation 6.12. The figure shows the benefit of the knowledge aggregation increases steeply if making errors are penalized.*

Another interesting finding is that the false positive penalty weight of 0.3 is the value that compensates for the performance overhead for our model problem. Any FP penalty weight over 0.3 results in a scenario where using KA with SL is beneficial for the system.

This value is used in some of the other experiments to normalize for the performance overhead.

6.3.3 Experiment 3: Impact of False Negative Probabilities

In Experiment 3, false negative probabilities for all robots were varied in order to investigate the parameter's impact on the performance of KA with SL (RQ.3). The false negative probability for the two robots were varied from 0.1 to 0.9. False positive probability was kept at 0.2, a very low value. False negative tasks are dirts that exist in the world and should be cleaned but can not be seen by the agent.

The first expectation for this experiment was the fact that the average time to complete true positive tasks would increase as the false negative probabilities increase with base case notably faster than the scenario with KA with SL. This is caused by the performance overhead of the KA with SL. In Figure 6.4, it is seen that the base case performs better than the case with KA with SL throughout the experiment.

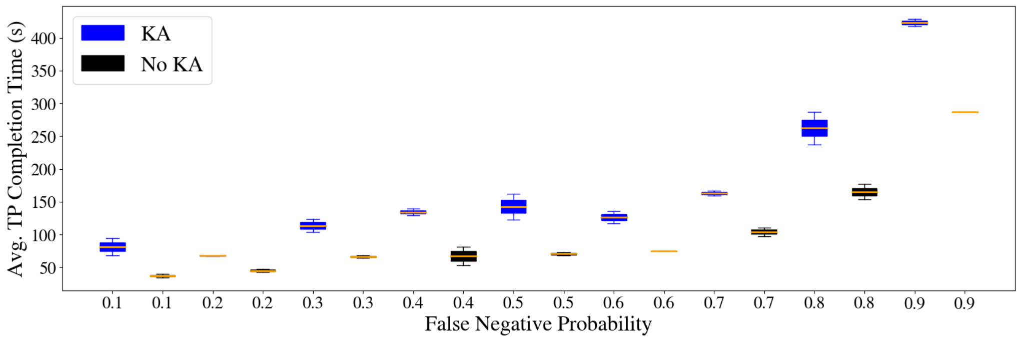


Figure 6.4: *The figure shows the average time to complete TP tasks tend to increase as the false negative probability increases. The first case (blue) represents the scenario where KA with SL is used and the black bars represents the base case. The orange line indicates the median. The results show that the base case is better throughout the experiment. This would indicate base case is more suitable for scenarios where false negative probability is high and false positive probability is low.*

Furthermore, the optimal plan of action for cases with high false negative probability and low false positive probability is to propagate the detected dirts as goals immediately after the first observation [34]. Because of this reason in H3.1, we hypothesized that KA with SL is not the optimal plan of action in cases with high false negative probability. In Figure 6.5, Q-Score at the end of the simulations are shown for the base case and the case with KA with SL.

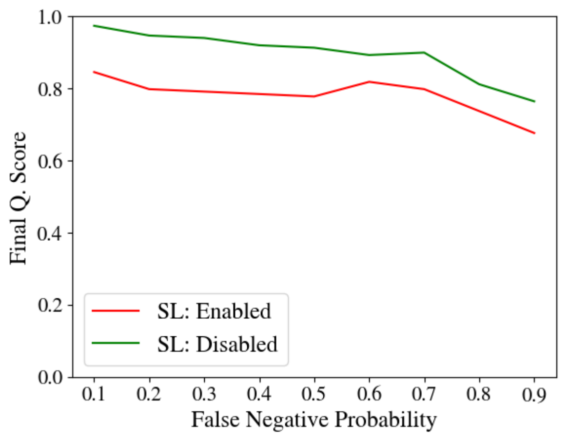


Figure 6.5: *The figure shows the final Q-score using KA with SL and the base case. FN probability is varied for this experiment and the results show a nearly parallel line between the Q_{SL} and Q_{Base}. This means that KA with SL is not affected significantly from increasing the FN probability.*

The lines are mostly parallel which indicates that KA with SL introduces a performance overhead but is not affected by the FN probability significantly. This is a surprising result for us as we expected to see a degration of performance by the KA with SL when the false negative probabilities increase. This expectation was based on our intuition explained above for H3.1. This was not reflected in our experimental results, in fact the opposite was observed. The KA with SL case had a minor increase in performance after 0.6 FN probability. This might be an artifact of "lucky runs" however the fact that there is an increase not a decrease in the Q-score should be investigated further.

6.3.4 Experiment 4: Impact of Threshold

For the fourth experiment, the impact of threshold on the performance of KA with SL was examined (RQ.4). The threshold value is varied between 0.2, 0.4, 0.6, and 0.8. The experiment is run for false positive probability of 0.2 and 0.8. This experiment is similar to the one run in Neuss [34] but the 0.5 false positive probability is removed in our case. This is done to avoid the unexpected spike that happens around 0.5 FP probability that can be seen in Experiment 1 and Experiment 2.

The threshold is a property of Subjective Logic. As explained in Section 2.2, Subjective Logic opinions are ordered quadruplets with the notation $O_x = (b_x, d_x, u_x, a_x)$. In this notation b_x is the belief mass, d_x is the disbelief mass, u_x is the uncertainty mass that represents the vacuity of evidence and a_x is the base rate. The details of Subjective Logic can be found Section 2.2. The threshold value is a threshold of the probability calculated from the SL opinion. As mentioned in Section 2.2, a SL opinion can be projected into a probability distribution using Equation 6.13.

$$P(x) = b_x + u_x a_x \tag{6.13}$$

A threshold of 0.8 would mean only the opinion with $P(x) > 0.8$ will be propagated as a goal. Thus, the higher the threshold value goes, the stronger KA with SL is enforced.

Intuitively, the threshold value can also be interpreted as the number of observation a dirt needs to get in order to be propagated as a goal. More observations are needed to propagate a task as a goal when the threshold value is higher. Using this knowledge two hypotheses were created. H4.1 hypothesizes that the higher threshold values increase accuracy. This is expected as one of the primary benefits of KA with SL is the increased accuracy. H4.2 hypothesizes that lower threshold values increases the total number of cleaned tasks. This is also expected as lower threshold values would decrease the performance overhead caused by KA with SL. Both of these hypotheses are shown to be true in our experimental results in Table 6.2.

FP Probability	Threshold	TP Completion Ratio	FP Completion Ratio	Total Cleaned
0.2	0.2	0.79	0.21	52
	0.4	0.79	0.21	53
	0.6	0.72	0.28	50
	0.8	0.80	0.19	47
0.8	0.2	0.43	0.57	109
	0.4	0.46	0.53	125
	0.6	0.42	0.57	102
	0.8	0.47	0.53	98

Table 6.2: *The table shows the false positive probability, the threshold value, number of total cleaned tasks (TP + FP) and the error ratio of the cleaned dirts.*

As seen in Table 6.3, higher threshold values tend to increase the TP completion ratio while the lower threshold values tend to increase the total number of cleaned dirts. The next research question (RQ.5) that we had was investigating whether there is an optimal threshold value that maximizes the number of cleaned TPs. The same experiment was run with different false positive penalty weights to change the scenarios and see if one threshold value performs the best for all scenarios.

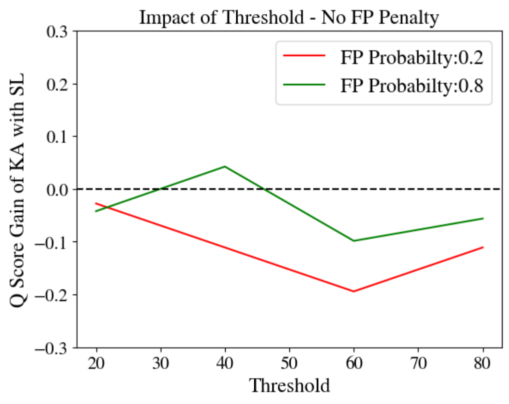


Figure 6.6: *The figure shows the impact of threshold values for FP probability 0.2 and 0.8. The gain is calculated using Equation 6.1. The image is generated with 0 FP completion penalty.*

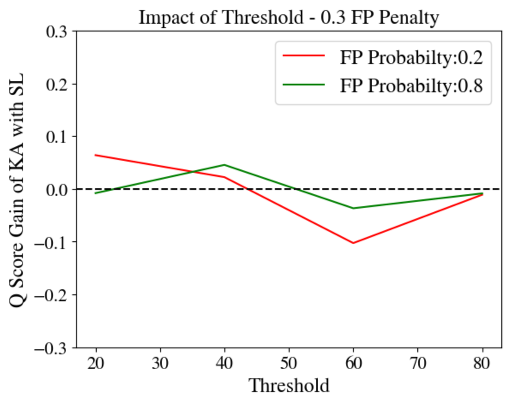


Figure 6.7: *The figure shows the impact of threshold values for FP probability 0.2 and 0.8. The gain is calculated using Equation 6.1. The image is generated with 0.3 FP completion penalty.*

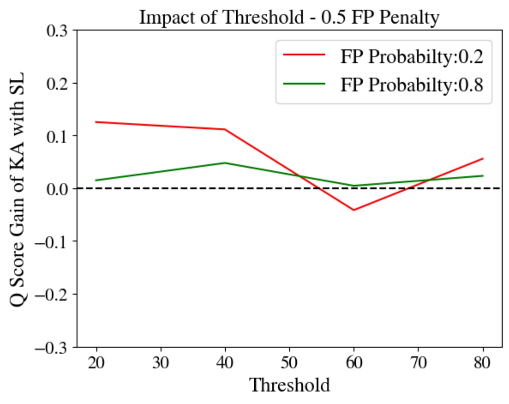


Figure 6.8: *The figure shows the impact of threshold values for FP probability 0.2 and 0.8. The gain is calculated using Equation 6.1. The image is generated with 0.5 FP completion penalty.*

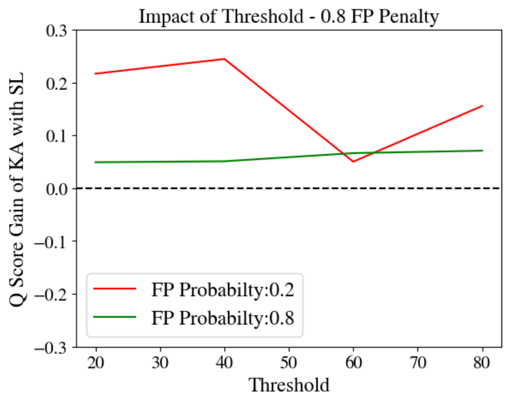


Figure 6.9: *The figure shows the impact of threshold values for FP probability 0.2 and 0.8. The gain is calculated using Equation 6.1. The image is generated with 0.8 FP completion penalty.*

As seen in plots in Figure 6.6 - 6.9, the optimal threshold value depends on the scenario and the adaptation goals. Our experimental results show that the optimal threshold value depends on the use-case, the uncertainties in the system and the adaptation goals. For most of the scenarios we tried, medium-to-low threshold values like 0.4 performed slightly better than the alternatives. Note that all of the experiments are run with a very high threshold value of 0.8 which is not optimal.

6.3.5 Experiment 5: Impact of Dirt Distribution

For this last experiment, the dirt distribution pattern was modified to see its impact on KA with SL. Experiment 1 was reused for this experiment as well. The false positive

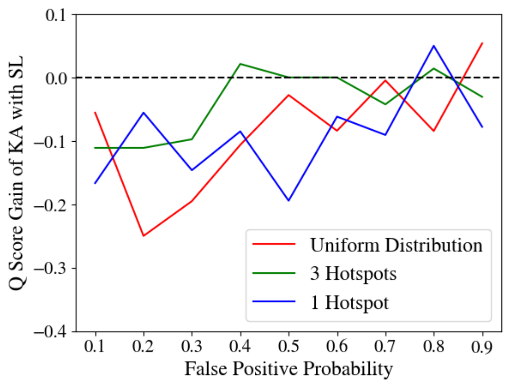


Figure 6.10: *The figure shows the impact of different dirt distribution patterns for various FP probabilities. The gain is calculated using Equation 6.1. The image is generated with 0 FP completion penalty.*

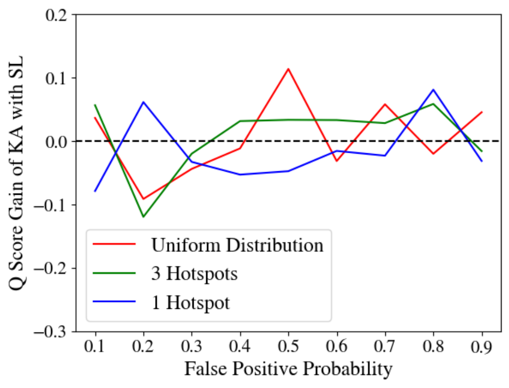


Figure 6.11: *The figure shows the impact of different dirt distribution patterns for various FP probabilities. The gain is calculated using Equation 6.1. The image is generated with 0.3 FP completion penalty.*

probability was changed between 0.1 and 0.9 and the experiment was repeated for uniform dirt distribution, all dirts concentrated on 1 hotspot and 3 hotspots. The main hypothesis tested in this experiment is checking whether more overlap between agents results in a better knowledge aggregation (H5.1). It was hypothesized that 1 hotspot scenario would have more overlap, thus result in a better performance for the KA with SL case. However the experimental results were inconclusive. The Figure 6.10 and 6.11 shows the gain of KA with SL for different dirt distribution patterns. Values over the dotted 0 line means KA with SL is performing better than the base case. Figure 6.10 has 0 FP completion penalty while Figure 6.11 has 0.3 FP completion penalty to compensate for the initial performance overhead.

As seen in the figures above, the experimental results are inconclusive and can not be used to prove that better overlap results in more benefits for the KA with SL case. This hypothesis should be investigated further to make a conclusive statement. The overlap metric can also be explored via changing the map size. Smaller maps would result in more overlap between the agents while larger maps would result in less overlap.

6.4 Discussion

Our experimental results mostly validated our hypotheses based on the results of the previous works [5, 34] and intuition about the potential benefits of KA with SL. As mentioned in H1.1, we hypothesized that KA with SL introduces a performance overhead initially. This was shown in the work of Neuss [34] by looking at the average time to complete TPs. This hypothesis was further proved in our thesis by comparing the results from an adaptivity perspective by looking at the Q-Scores. We believe this overhead is mainly caused by the fact that KA with SL uses several observations to propagate a task as a goal while the base case immediately propagates the task as a goal. This causes the base case to clear more TPs but more FPs as well. This behaviour is seen in Experiment 4 as well. Higher threshold values intuitively means the KA is stronger. The experimental results show that higher threshold values increase accuracy while decreasing the total number of cleaned tasks.

Another thing that was tested was how KA with SL performed in scenarios where the accuracy is a main business concern. In the real world these scenarios can be use-cases where the operation of the CPS is expensive or dangerous. In Experiment 2, we showed that when accuracy is a main concern and the completion of FPs are penalized, KA with SL performs significantly better than the base case. This result is an especially important finding as it clearly presents a scenario where KA with SL performs significantly better than the base case.

Our experimental results had surprising findings as well. One of our preliminary hypothesis was that more overlap between agents should results in better performance of KA with SL. As more overlap should result in more observation for the same tasks, we believed it would increase the accuracy and the overall system performance. Experiment 5 was designed to test this hypothesis by changing the dirt distribution to concentrate all tasks in one or more hotspots. We believed this would increase the overlap between the agents and 1 hotspot case should perform better than the other cases. However this was not reflected in the experimental results. In fact the 1 hotspot case on overall resulted in a worse performance by the KA with SL. This finding should be investigated further. The overlap hypothesis can be tested further by changing the map size and comparing the performance of KA with SL on larger maps and smaller maps. According to the overlap hypothesis the smaller maps should result in a better performance for the KA with SL.

To conclude, our experimental results showed that KA with SL introduces a performance overhead but compensates and overcomes the overhead if the uncertainty in the system is high and accuracy is a primary concern.

7 Conclusion

To conclude we want to summarize our findings in section 7.1, mention the remaining issues and threats to validity in section 7.2. Lastly, possible future improvements will be explained in section 7.3.

7.1 Summary

In this thesis, we explore the benefits and drawbacks of Knowledge Aggregation (KA) based on Subjective Logic (SL) for a Multi-Agent Self-Adaptive Cyber-Physical System (MA-SACPS) from a lens of adaptivity. We use the simulated MA-SACPS subsystem previously implemented by Neuss [34] and several other previous projects. We merged this MA-SACPS subsystem with the evaluation framework implemented by Bergemann [5] in order to be able to calculate the adaptivity metric (Q-Score). There were several challenges encountered during this merging. The details are explained in detail in Section 5.3.

Merging the two subsystems allowed configuring the entire system from a single configuration file. Leveraging this, the merged system and all of its dependencies were turned into a stand-alone Docker container. A new experimental tooling system was developed around this container to automate the configuration, execution, storage and analysis steps of the experimentation process. The new tooling system also allows running the simulations in parallel. In the end, we managed to reduce the required computational time to run all of the experiments 6 times. This was a result of resource constraints, the tooling would allow reducing the computational time even further. The new experimental tooling system is explained in detail in Section 5.4.

After the implementation was done, the evaluation of the impact of KA with SL on the overall system's adaptivity started. Five experiments were designed in total. These experiments are the impact of false positive probability, false positive penalty, false negative probability, threshold and dirt distribution. The results show that KA with SL introduces a performance overhead due to the fact that it requires several observations before a detected dirt can be propagates as a goal. However, the increased accuracy KA with SL brings can compensate and overcome the overhead in certain scenarios. Our experimental results show that benefits of KA with SL increase as the false positive probability (internal uncertainty) increases. Furthermore, KA with SL starts to benefit the adaptivity and the performance of the system when the completion of false positives are penalized. By looking at the results, we would state that KA with SL is suitable to be used in scenarios where high accuracy by the agents are desired. These could be use-cases where the operation of the CPS is expensive, dangerous or the completion of false positives are highly undesired due to some reason.

7.2 Threats to Validity

We have identified several issues that might require attention. They will be addressed in this section.

- Q-score depends on the computational resources of the computer that is running the simulation. So the experiments must be run on computers with similar resources in order to be comparable.
- Changes made during this thesis might break the decoupled structure between the evaluation framework and the simulators. We only used the realistic simulator developed by [34], thus the changes we introduced are only tested for that subsystem. We have reason to believe the support for the custom simulator developed by [45] might be broken due to using GoalObjects instead of DirtObjects in the `global_dirt_generator` node of the evaluation framework. The changes are documented in the code, the support for the custom simulator can be restored with minimal effort by looking at the documentation in the code.
- We encountered several unexpected bumps and drops in the Q-score graph, the most significant one being the rise at false positive probability 0.5 and the drop at false positive probability 0.6 in Experiment 1. This pattern exists with different dirt seeds and when repeating the same seed multiple times. This behaviour should be examined.
- Currently completion of false positive dirts are penalized in the Q-score by adding a new score function that calculates the ratio of completed false positive dirts over the number of spawned false positive dirts. Even though this can be seen as an acceptable way of penalizing false positives, a better way exists. Adding a waiting period after the completion of each dirt achieves the same result and represents the cost of the cleaning operation better. This was not done in the scope of this thesis due to technical limitations.

7.3 Future Work

There are several aspects this work can be extended. The envisioned improvements can be summarized in three categories, improvements on the ROS-based implementation, improvements on the new tooling and extending the experimentations.

The MA-SACPS subsystem we used in this thesis is open to many improvements. It would be very interesting to add a modifiable waiting time to the dirt cleaning operation. This would open the possibility of evaluation the self-adaptive behaviour of the system in relation to the cost of the operation. This was done via the modified Q-score (section 5.2.3) in this thesis but the waiting time would be a better alternative. Another interesting improvement would be adding the so-called overlap metric. This would be a numerical value that represents how much the robots' sensors are overlapping with each other. This would be a very interesting metric to have as it was hypothesized that the amount of overlap the robots have should impact the performance of the knowledge aggregation greatly. Another beneficial addition to this research would be adopting the newest formalism of adaptivity based on the series expansion made by Petrovska [37].

The newly implemented tooling can be improved greatly as well. The tooling system currently does not have a very user friendly configuration interface. Everything is done via config files and shell scripts. However it would be very useful to have a web interface to configure and monitor the experiments. The current implementation allows for this via the endpoints it exposes however the implementation of this web interface was not in the scope of this thesis. Individual components of the tooling can be improved as well. For example the task queue can be converted into a priority queue to allow prioritizing which experiments are run first. The docker containers sometimes get stuck during running the ROS simulation. Currently these containers have to be restarted manually. A heartbeat like signal can be published throughout the experiment and stuck containers can be restarted automatically by looking at this signal.

Lastly, the experiments can be extended to investigate a wider parameter space. Changing the map and evaluate how it affects the performance of the knowledge aggregation from an adaptivity perspective would be especially interesting. Different map sizes and map layouts with more open spaces or more corridors can be tested. Changing the robot number would open up noteworthy research questions as well.

List of Figures

List of Listings

List of Tables

Bibliography

[1] F. Amar, C. Grand, and G. Besseron. “Performance evaluation of locomotion modes of an hybrid wheel-legged robot for self-adaptation to ground conditions”. In: *8th ESA Workshop on Advanced Space Technologies for Robotics and Automation, ASTRA 2004* (2004).

[2] K. Angelopoulos, A. Papadopoulos, and V. Souza. “Model predictive control for software systems with CobRA”. In: *Proceedings - 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2016* (2016).

[3] L. Baresi, L. Pasquale, and P. Spoletini. “Fuzzy goals for requirements-driven adaptation”. In: *2010 18th IEEE International Requirements Engineering Conference* (2010).

[4] T. Beffart, K. Hawryluk, and R. C. Medina. *SSACPS Praktikum Team 1: Approach Description*. Chair of Software and System Engineering, 2019.

[5] S. Bergemann. “Evaluation Framework for a Self-Adative Cyber-Physical System on an Example of a Multi-Robot System”. MA thesis. Technische Universität München, Mar. 2021.

[6] N. Bouzid. “Evaluating the Impact of the Base Rate in Knowledge Aggregation with Subjective Logic in Multi-Agent Self-Adaptive Cyber-Physical Systems”. In: (Nov. 2021).

[7] Y. Brun, G. Serugendo, C. Gacek, H. Giese, H. Kienle, et al. “Engineering self-adaptive systems through feedback loops”. In: *Software engineering for self-adaptive systems* (2009).

[8] B. Cheng, P. Sawyer, N. Bencomo, and J. Whittle. “A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty”. In: *International Conference on Model Driven Engineering Languages and Systems* (2009).

[9] S. Cheng, D. Garlan, and B. Schmerl. “Architecture-based self-adaptation in the presence of multiple objectives”. In: (2006).

[10] S. Cheng, D. Garlan, and B. Schmerl. “Evaluating the effectiveness of the rainbow self-adaptive system”. In: *Proceedings of the 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2009* (2009).

[11] *DockerHub*. URL: `https://www.docker.com/products/docker-hub` (visited on 01/15/2022).

[12] A. Filieri, C. Ghezzi, and A. Leva. “Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements”. In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, Proceedings* (2011).

[13] *Firebase*. URL: `https://firebase.google.com/docs` (visited on 01/28/2022).

[14] D. Garlan and S. C. B. Schmerl. “Software architecture-based self-adaptation”. In: *Autonomic Computing and Networking* (2009).

[15] D. Garlan, S. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. "Rainbow: Architecture-based self-adaptation with reusable infrastructure". In: *Computer* (2004).

[16] S. Gerasimou, R. Calinescu, and S. Shevtsov. "UNDERSEA: An Exemplar for Engineering Self-Adaptive Unmanned Underwater Vehicles". In: *Proceedings - 2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2017* (2017).

[17] I. Gerostathopoulos, T. Vogel, D. Weyns, and P. Lago. "How do we Evaluate Self-adaptive Software Systems?" In: (2021).

[18] *Google Cloud Platform*. URL: `https://cloud.google.com/docs` (visited on 01/28/2022).

[19] R. Heijden, H. Kopp, and F. Kargl. "Multi-Source Fusion Operations in Subjective Logic". In: *2018 21st International Conference on Information Fusion, FUSION 2018* (2018). DOI: `10.23919/ICIF.2018.8455615`.

[20] S. Hezavehi, D. Weyns, P. Avgeriou, R. Calinescu, R. Mirandola, et al. "Uncertainty in Self-adaptive Systems: A Research Community Perspective". In: *ACM Transactions on Autonomous and Adaptive Systems* (Dec. 2021).

[21] G. Hoffman and C. Breazeal. "Effects of anticipatory perceptual simulation on practiced human-robot tasks". In: *Autonomous Robots* (2010).

[22] M. Iftikhar, G. Ramachandran, and P. Bollansée. "DeltaIoT: A Self-Adaptive Internet of Things Exemplar". In: *Proceedings - 2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2017* (2017).

[23] P. Jamshidi, J. Camara, and B. Schmerl. "Machine learning meets quantitative planning: Enabling self-adaptation in autonomous robots". In: *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems* (2019).

[24] A. Josang. "A Logic for Uncertain Probabilities". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 3 (2001).

[25] A. Josang. *Subjective Logic*. Springer International Publishing Switzerland, 2016. ISBN: 978-3-319-42335-7. DOI: `10.1007/978-3-319-42337-1`.

[26] A. Josang, D. Wang, and J. Zhang. "Multi-source fusion in subjective logic". In: *20th International Conference on Information Fusion, Fusion 2017 - Proceedings)* (2017). DOI: `10.23919/ICIF.2017.8009820`.

[27] J. Kephart and D. Chess. "The vision of autonomic computing". In: *Computer* (2003).

[28] S. Kugele, A. Petrovska, and I. Gerostathopoulos. "Towards a taxonomy of autonomous systems". In: *European Conference on Software Architecture*. Springer, 2021.

[29] R. Lemos, D. Garlan, and C. Ghezzi. "Software engineering for self-adaptive systems: research challenges in the provision of assurances". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2017).

[30] R. Lemos, H. Giese, and H. Müller. "Software engineering for self-adaptive systems: A second research roadmap". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2013).

[31] V. Matena, T. Bures, and I. Gerostathopoulos. "Model problem and testbed for experiments with adaptation in smart cyber-physical systems". In: *Proceedings - 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2016* (2016).

[32] G. Moreno, C. Kinneer, and A. Pandey. "DARTSim: An exemplar for evaluation and comparison of self-adaptation approaches for smart cyber-physical systems". In: *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems* (2019).

[33] M. Neuss, S. Bergemann, and M. Büchner. *SSACPS Praktikum Team 2: Approach Description*. Chair of Software and System Engineering, 2019.

[34] M. Neuss. "Run-Time Reasoning and Solving Conflicting Observations with Subjective Logic in Multi-Agent Self-Adaptive Cyber-Physical Systems". MA thesis. Technische Universität München, Dec. 2020.

[35] P. Oreizy, M. Gorlick, R. Taylor, D. Heimhigner, G. Johnson, et al. "An architecture-based approach to self-adaptive software". In: *IEEE Intelligent Systems and Their Applications* (1999).

[36] A. Petrovska. "Is every system a self-adaptive system? A critical reflection. Under review". In: (2022).

[37] A. Petrovska, S. Kugele, T. Hutzelmann, T. Beffart, S. Bergemann, et al. "Defining Adaptivity and Logical Architecture for Engineering (Smart) Self-Adaptive Cyber-Physical Systems". 2022.

[38] A. Petrovska, S. Quijano, I. Gerostathopoulos, and A. Pretschner. "Knowledge Aggregation with Subjective Logic in Multi-Agent Self-Adaptive Cyber-Physical Systems". In: (Oct. 2020). DOI: 10.1145/3387939.3391600.

[39] M. Provoost and D. Weyns. "DingNet: A self-adaptive internet-of-things exemplar". In: *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems* (2019).

[40] S. Quijano. "Runtime Multi-Agent Knowledge Aggregation with Subjective Logic in Self-Adaptive Cyber-Physical Systems". MA thesis. Technische Universität München, Nov. 2019.

[41] A. Ramirez, A. Jensen, and B. Cheng. "A taxonomy of uncertainty for dynamically adaptive systems". In: *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS* (2012).

[42] S. Schmid, I. Gerostathopoulos, and C. Prehofer. "Self-Adaptation Based on Big Data Analytics: A Model Problem and Tool". In: *Proceedings - 2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2017* (2017).

[43] G. Shafer. “A mathematical theory of evidence”. In: *Princeton University Press* (1976).

[44] N. Villegas, H. Müller, and G. Tamura. “A framework for evaluating quality-driven self-adaptive software systems”. In: *Proceedings - International Conference on Software Engineering* (2011).

[45] J. Weick. “Realization of Adaptive System Transitions for Smart Self-Adaptive Cyber-Physical Systems using a Modular Approach with Learned Parameters on an Example of Multi-Robot Collaboration”. MA thesis. Technische Universität München, July 2020.

[46] D. Weyns. “Software Engineering of Self-Adaptive Systems: An Organised Tour and Future Challenges”. In: *Handbook of Software Engineering* (2017).

[47] D. Weyns and T. Ahmad. “Claims and evidence for architecture-based self-adaptation: A systematic literature review”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2013).

[48] D. Weyns and V. G. B. Schmerl. “On patterns for decentralized control in self-adaptive systems”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2013).

[49] D. Weyns and R. Calinescu. “Tele Assistance: A Self-Adaptive Service-Based System Exemplar”. In: *Proceedings - 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015* (2015).

[50] D. Weyns, B. Schmerl, and V. Grassi. “On patterns for decentralized control in self-adaptive systems”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2013).

[51] J. Yoon, T. Shiratori, and S. Yu. “Self-supervised adaptation of high-fidelity face models for monocular performance tracking”. In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (2019).

[52] S. Yousfi. “Design and Implementation of ROS-Based Simulated Multi-Robot Systems for Data Collection”. MA thesis. Technische Universität München, 2018.

[53] Z. Zhu and H. Hu. “Robot learning from demonstration in robotic assembly: A survey”. In: (2018).

A Additional Code Snippets

This chapter will present some of the code snippets used throughout the thesis.

A.1 Score Functions

A.1.1 Configuration

```yaml
# parameters.yaml

no_crashes: {
    type: "mission_goal",
    function_index: 4,
    variables: {
      input_to_check: "robot_crashes",
      threshold: 0
    },
    description: "No robot should crash at all."
  }
```

Listing A.1: *The configuration field in the parameters.yaml file for the no_crashes business goal.*

```yaml
# parameters.yaml

minimum_cleaned: {
    type: "mission_goal",
    function_index: 5,
    variables: {
      input_to_check: "finished_dirt_number",
      offset: 0,
      aim: "spawned_dirt_number",
      factor: 0.1,
      start_requirement: 5
    }
  }
```

Listing A.2: *The configuration field in the parameters.yaml file for the minimum_cleaned business goal.*

```yaml
# parameters.yaml

higher_cleaning_rate: {
    type: "adaptation_goal",
    weight: 0.9,
    function_index: 0,
    variables: {
      portion_number: "finished_goal_gt_number",
      total_amount: "spawned_goal_gt_number"
    }
  }
```

Listing A.3: *The configuration field in the parameters.yaml file for the higher_cleaning_rate adaptation goal.*

```
# parameters.yaml

shorter_dirt_existence: {
    type: "adaptation_goal",
    weight: 0.2,
    function_index: 6,
    variables: {
      portion_number: "actual_dirt_existence_time",
      total_amount: "theoretical_dirt_existence_time"
    }
  }
```

Listing A.4: *The configuration field in the parameters.yaml file for the shorter_dirt_existence adaptation goal.*

```
# parameters.yaml

higher_detection: {
    type: "adaptation_goal",
    weight: 0.2,
    function_index: 0,
    variables: {
      portion_number: "detected_dirt_number",
      total_amount: "spawned_dirt_number"
    },
    description: "The more dirt is detected, the better it is."
  }
```

Listing A.5: *The configuration field in the parameters.yaml file for the higher_detection adaptation goal.*

```

# parameters.yaml

less_distance: {
    type: "adaptation_goal",
    weight: 0.1,
    function_index: 3,
    variables: {
      traveled_distance: "traveled_distance_total",
      elapsed_time: "theoretical_elapsed_time",
      max_possible_velocity: "parameter:robot_max_velocity",
      robot_count: "parameter:sim_no_of_robots"
    },
  }
```

Listing A.6: *The configuration field in the parameters.yaml file for the less_distance adaptation goal.*

```
# parameters.yaml

lower_error_rate: {
    type: "adaptation_goal",
    weight: 0.1,
    function_index: 7,
    variables: {
      gt_finished: "finished_goal_gt_number",
      total_finished: "finished_dirt_number",
      total_spawned: "spawned_dirt_number",
      gt_spawned: "spawned_goal_gt_number"
    },
  }
```

Listing A.7: *The configuration field in the parameters.yaml file for the lower_error_rate adaptation goal.*

A.1.2 Implementation

```
# config.json

{
  "experiment_name": "experiment1/sl",
  "run_prefix": "fpp",
  "simulation_time": 2700,
  "seed": [72],
  "use_sl": true,
  "sl_operator": "Comb.",
  "adaptive_scheduling": true,
  "dirt_use_uniform_distr": true,
  "spawn_interval": 60,
  "variables": [
    {
      "name": "robot_0_fpp",
      "values": [0.1, 0.2, 0.3, 0.4,
        0.5, 0.6, 0.7, 0.8, 0.9],
      "sim": false
    },
    {
      "name": "robot_1_fpp",
      "values": [0.1, 0.2, 0.3, 0.4,
        0.5, 0.6, 0.7, 0.8, 0.9],
      "sim": false
    }
  ]
}
```

Listing A.8: *The config.json file used to configure the Subjective Logic enabled sub-experiment of Experiment 1*

A.2 Experiment Configuration Files

This section contains the config.json configuration used for the experiments in Section 6. These config.json files can be reused to duplicate the experiments.

A.2.1 Experiment 1 - Impact of False Positive Probability

This experiment was divided into two subexperiments. Both sub-experiments have the same configuration except one of them has SL enabled and the other has SL disabled.

```
# config.json

{
  "experiment_name": "experiment1/sl",
  "run_prefix": "fpp",
  "simulation_time": 2700,
  "seed": [72],
  "use_sl": true,
  "sl_operator": "Comb.",
  "adaptive_scheduling": true,
  "dirt_use_uniform_distr": true,
  "spawn_interval": 60,
  "variables": [
    {
      "name": "robot_0_fpp",
      "values": [0.1, 0.2, 0.3, 0.4,
        0.5, 0.6, 0.7, 0.8, 0.9],
      "sim": false
    },
    {
      "name": "robot_1_fpp",
      "values": [0.1, 0.2, 0.3, 0.4,
        0.5, 0.6, 0.7, 0.8, 0.9],
      "sim": false
    }
  ]
}
```

Listing A.9: *The config.json file used to configure the Subjective Logic enabled sub-experiment of Experiment 1*

```
# config.json

{
  "experiment_name": "experiment1/no-sl",
  "run_prefix": "fpp",
  "simulation_time": 2700,
  "seed": [72],
  "use_sl": false,
  "sl_operator": "Comb.",
  "adaptive_scheduling": true,
  "dirt_use_uniform_distr": true,
  "spawn_interval": 60,
  "variables": [
    {
      "name": "robot_0_fpp",
      "values": [0.1, 0.2, 0.3, 0.4,
        0.5, 0.6, 0.7, 0.8, 0.9],
      "sim": false
    },
    {
      "name": "robot_1_fpp",
      "values": [0.1, 0.2, 0.3, 0.4,
        0.5, 0.6, 0.7, 0.8, 0.9],
      "sim": false
    }
  ]
}
```

Listing A.10: *The config.json file used to configure the Subjective Logic disabled sub-experiment of Experiment 1*

A.2.2 Experiment 2 - Impact of False Positive Penalty

The same config.json file as the Experiment 1 was used. The weights to calculate the Q-score was changed at the analysis step. This was possible as the results.csv generated by the evaluation framework stores the values of each score function.

A.2.3 Experiment 3 - Impact of False Negative Probability

This experiment was divided into two sub-experiments as well. Similar to Experiment 1, one sub-experiment has SL enabled and the other one has SL disabled. The config.json files used to replicate the experiments can be found below.

```
# config.json

{
  "experiment_name": "experiment-3/sl",
  "run_prefix": "fnp",
  "simulation_time": 2700,
  "seed": [72],
  "use_sl": true,
  "sl_operator": "Comb.",
  "adaptive_scheduling": true,
  "dirt_use_uniform_distr": true,
  "spawn_interval": 60,
  "variables": [
    {
      "name": "robot_0_fnp",
      "values": [0.1, 0.2, 0.3, 0.4,
        0.5, 0.6, 0.7, 0.8, 0.9],
      "sim": false
    },
    {
      "name": "robot_1_fnp",
      "values": [0.1, 0.2, 0.3, 0.4,
        0.5, 0.6, 0.7, 0.8, 0.9],
      "sim": false
    }
  ]
}

\begin{lstlisting}[caption={The config.json file used to configure the
    Subjective Logic disabled sub-experiment of Experiment 3},captionpos
    =b]
# config.json

{
  "experiment_name": "experiment-3/no-sl",
  "run_prefix": "fnp",
  "simulation_time": 2700,
  "seed": [72],
  "use_sl": false,
  "sl_operator": "Comb.",
  "adaptive_scheduling": true,
  "dirt_use_uniform_distr": true,
  "spawn_interval": 60,
  "variables": [
    {
      "name": "robot_0_fnp",
      "values": [0.1, 0.2, 0.3, 0.4,
        0.5, 0.6, 0.7, 0.8, 0.9],
      "sim": false
    },
    {
      "name": "robot_1_fnp",
      "values": [0.1, 0.2, 0.3, 0.4,
        0.5, 0.6, 0.7, 0.8, 0.9],
      "sim": false
    }
  ]
}
```

Listing A.11: *The config.json file used to configure the Subjective Logic enabled sub-experiment of Experiment 3*

A.2.4 Experiment 4 - Impact of Threshold

This experiment was divided into six sub-experiments to configure in the experiment designer tool. These are false positive probability of 0.2, 0.5 and 0.8. Also each part is run with SL enabled and SL disabled.

```
# config.json

{
  "experiment_name": "experiment4/fpp02/sl",
  "run_prefix": "sl_threshold",
  "simulation_time": 2700,
  "seed": [72],
  "use_sl": true,
  "sl_operator": "Comb.",
  "adaptive_scheduling": true,
  "dirt_use_uniform_distr": true,
  "spawn_interval": 60,
  "variables": [
    {
      "name": "sl_threshold",
      "values": [20, 40, 60, 80],
      "sim": true
    },
    {
      "name": "robot_0_fpp",
      "values": [0.2,0.2,0.2,0.2],
      "sim": false
    },
    {
      "name": "robot_1_fpp",
      "values": [0.2,0.2,0.2,0.2],
      "sim": false
    }
  ]
}
```

Listing A.12: *The config.json file used to configure the 0.2 false positive probability and Subjective Logic enabled sub-experiment of Experiment 4*

```
# config.json

{
  "experiment_name": "experiment4/fpp02/no-sl",
  "run_prefix": "sl_threshold",
  "simulation_time": 2700,
  "seed": [72],
  "use_sl": false,
  "sl_operator": "Comb.",
  "adaptive_scheduling": true,
  "dirt_use_uniform_distr": true,
  "spawn_interval": 60,
  "variables": [
    {
      "name": "sl_threshold",
      "values": [20, 40, 60, 80],
      "sim": true
    },
    {
      "name": "robot_0_fpp",
      "values": [0.2,0.2,0.2,0.2],
      "sim": false
    },
    {
      "name": "robot_1_fpp",
      "values": [0.2,0.2,0.2,0.2],
        "sim": false
      }
    ]
}
```

Listing A.13: *The config.json file used to configure the 0.2 false positive probability and Subjective Logic disabled sub-experiment of Experiment 4*

```
# config.json

{
  "experiment_name": "experiment4/fpp05/sl",
  "run_prefix": "sl_threshold",
  "simulation_time": 2700,
  "seed": [72],
  "use_sl": true,
  "sl_operator": "Comb.",
  "adaptive_scheduling": true,
  "dirt_use_uniform_distr": true,
  "spawn_interval": 60,
  "variables": [
    {
      "name": "sl_threshold",
      "values": [20, 40, 60, 80],
      "sim": true
    },
    {
      "name": "robot_0_fpp",
      "values": [0.5,0.5,0.5,0.5],
      "sim": false
    },
    {
      "name": "robot_1_fpp",
      "values": [0.5,0.5,0.5,0.5],
      "sim": false
    }
  ]
}
```

Listing A.14: *The config.json file used to configure the 0.5 false positive probability and Subjective Logic enabled sub-experiment of Experiment 4*

```
# config.json

{
  "experiment_name": "experiment4/fpp05/no-sl",
  "run_prefix": "sl_threshold",
  "simulation_time": 2700,
  "seed": [72],
  "use_sl": false,
  "sl_operator": "Comb.",
  "adaptive_scheduling": true,
  "dirt_use_uniform_distr": true,
  "spawn_interval": 60,
  "variables": [
    {
      "name": "sl_threshold",
      "values": [20, 40, 60, 80],
      "sim": true
    },
    {
      "name": "robot_0_fpp",
      "values": [0.5,0.5,0.5,0.5],
      "sim": false
    },
    {
      "name": "robot_1_fpp",
      "values": [0.5,0.5,0.5,0.5],
      "sim": false
    }
  ]
}
```

Listing A.15: *The config.json file used to configure the 0.5 false positive probability and Subjective Logic disabled sub-experiment of Experiment 4*

```
# config.json

{
  "experiment_name": "experiment4/fpp08/sl",
  "run_prefix": "sl_threshold",
  "simulation_time": 2700,
  "seed": [72],
  "use_sl": true,
  "sl_operator": "Comb.",
  "adaptive_scheduling": true,
  "dirt_use_uniform_distr": true,
  "spawn_interval": 60,
  "variables": [
    {
      "name": "sl_threshold",
      "values": [20, 40, 60, 80],
      "sim": true
    },
    {
      "name": "robot_0_fpp",
      "values": [0.8,0.8,0.8,0.8],
      "sim": false
    },
    {
      "name": "robot_1_fpp",
      "values": [0.8,0.8,0.8,0.8],
      "sim": false
    }
  ]
}
```

Listing A.16: *The config.json file used to configure the 0.8 false positive probability and Subjective Logic enabled sub-experiment of Experiment 4*

```
# config.json

{
  "experiment_name": "experiment4/fpp02/no-sl",
  "run_prefix": "sl_threshold",
  "simulation_time": 2700,
  "seed": [72],
  "use_sl": false,
  "sl_operator": "Comb.",
  "adaptive_scheduling": true,
  "dirt_use_uniform_distr": true,
  "spawn_interval": 60,
  "variables": [
    {
      "name": "sl_threshold",
      "values": [20, 40, 60, 80],
      "sim": true
    },
    {
      "name": "robot_0_fpp",
      "values": [0.8,0.8,0.8,0.8],
      "sim": false
    },
    {
      "name": "robot_1_fpp",
      "values": [0.8,0.8,0.8,0.8],
      "sim": false
    }
  ]
}
```

Listing A.17: *The config.json file used to configure the 0.8 false positive probability and Subjective Logic disabled sub-experiment of Experiment 4*

A.2.5 Experiment 5 - Impact of Dirt Distribution

This experiment was divided into three sub-experiments to configure in the experiment designer tool. The first sub-experiment reuses the SL disabled subexperiment of Experiment 1. The other sub-experiments are with 1 hotspot and 3 hotspots. To configure them both config.json and template parameters.yaml file was changed.

```
# config.json

{
  "experiment_name": "experiment5/hotspots1",
  "run_prefix": "fpp",
  "simulation_time": 2700,
  "seed": [72],
  "use_sl": true,
  "sl_operator": "Comb.",
  "adaptive_scheduling": true,
  "dirt_use_uniform_distr": false,
  "spawn_interval": 60,
  "variables": [
    {
      "name": "robot_0_fpp",
      "values": [0.1, 0.2, 0.3, 0.4,
        0.5, 0.6, 0.7, 0.8, 0.9],
      "sim": false
    },
    {
      "name": "robot_1_fpp",
      "values": [0.1, 0.2, 0.3, 0.4,
        0.5, 0.6, 0.7, 0.8, 0.9],
      "sim": false
    }
  ]
}
```

Listing A.18: *The config.json file used to configure the 1 hotspot sub-experiment of Experiment 5. Used with the change in the template parameters.yaml file in Listing A.7*

```
# parameters.yaml

...
dirt_number_of_random_hotspots: 1
...
```

Listing A.19: *The changed line in the template parameters.yaml file of the experiment designer. Used with the config.json in Listing A.6*

```
# config.json

{
  "experiment_name": "experiment5/hotspots3",
  "run_prefix": "fpp",
  "simulation_time": 2700,
  "seed": [72],
  "use_sl": false,
  "sl_operator": "Comb.",
  "adaptive_scheduling": true,
  "dirt_use_uniform_distr": false,
  "spawn_interval": 60,
  "variables": [
    {
      "name": "robot_0_fpp",
      "values": [0.1, 0.2, 0.3, 0.4,
        0.5, 0.6, 0.7, 0.8, 0.9],
      "sim": false
    },
    {
      "name": "robot_1_fpp",
        "values": [0.1, 0.2, 0.3, 0.4,
          0.5, 0.6, 0.7, 0.8, 0.9],
        "sim": false
      }
    ]
}
```

Listing A.20: *The config.json file used to configure the 1 hotspot sub-experiment of Experiment 5. Used with the change in the template parameters.yaml file in Listing A.9*

```
# parameters.yaml

...
dirt_number_of_random_hotspots: 3
...
```

Listing A.21: *The changed line in the template parameters.yaml file of the experiment designer. Used with the config.json in Listing A.8*