Institut für Informatik
Lehrstuhl für Programmierung und Softwaretechnik

LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor Thesis

An Extended Simulator for Motivation-Based and Fault Tolerant Task Allocation in Multi-Robot Systems

Robert Gutschale

Aufgabensteller: Prof. Dr. Martin Wirsing
Betreuer: Annabelle Klarl, Christian Kroiß
Abgabetermin: 29. März 2012

Ich versichere hiermit eidesstattlich, dass ich die vorliegende Arbeit selbstständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe.

München, den 29. März 2012

. . . .

(Unterschrift des Kandidaten)

Zusammenfassung

Die Simulation von Systemen mit mehreren Robotern spielt in der Entwicklung eine sehr wichtige Rolle. Einerseits kann die Simulation Geld einsparen, da reale Roboter sehr wertvoll sind und ihre Wartung und Reparatur hohe Kosten verursachen. Andererseits sind Experimente mit realen Robotern sehr zeitaufwändig, da in der Regel nur ein Experiment gleichzeitig durchgeführt werden kann und es auch keine Möglichkeit gibt, den Verlauf des Experiments zu beschleunigen. Ein Ausfall von einem Roboter führt hier unweigerlich zu einer Verzögerung des Experiments.

Auch ist es normalerweise nicht einfach, dynamische Umgebungen und kontrollierte Fehler zu erzeugen, wie zum Beispiel fehlerhafte Sensoren oder Aktuatoren der Roboter. Selbst wenn eine einfache Lösung zu diesen Problemen gefunden wird, ist es oft aufwändig, eine Reihe von Experimenten mit den exakt gleichen Umgebungsänderungen oder Fehlern an den Robotern durchzuführen. Häufig ist es erwünscht, ein Experimente mit identischen Gegebenheiten zu wiederholen, um beispielsweise unterschiedliche Zusammensetzungen des Roboterteams zu testen und untereinander zu vergleichen.

Diese Arbeit präsentiert einen Simulator, der es ermöglicht Simulationen mit (möglicherweise automatisierten) Umgebungsänderungen oder fehlerhaften Robotern durchzuführen. Als Grundlage wurde der von Martin Burger entwickelte *Swarmulator* [Bur12] genutzt, auf dem der Großteil der internen Logik dieses Simulators basiert. Die Funktionalität des Simulators wurde erweitert, um Experimente mit den eben erwähnten Änderungen oder Fehlern durchführen zu können. Um die Aktionen des Roboterteams zu koordinieren, wurde weiterhin die von Parker entwickelte ALLIANCE Architektur [Par98] implementiert. Zusätzlich wertet diese Arbeit Resultate verschiedener Experimente aus, die anhand einer Fallstudie durchgeführt wurden. Diese Fallstudie basiert auf Experimenten, die von Parker [Par98] mit reellen Robotern durchgeführt wurden.

Abstract

In the development of multi-robot systems, simulation plays a crucial role. On the one hand, simulation can save money, because physical robots are very valuable and damage to them results into high maintenance costs. On the other hand, experiments with physical robots are very time consuming, since usually only one experiment at a time is conducted and there is no possibility to fast forward when there is no interesting robot behavior at the moment. Here, damage to the robots results into a delay of the experiment.

Furthermore, it usually isn't easy to create dynamic environments and controlled failures such as faulty sensors or actuators within the robots. Even if a trivial solution to these problems is found, it's very laborious to conduct a number of experiments with the exact same changes in the environment and/or failures. It is often desired to repeat an experiment with identical conditions, for example to test different compositions of robot teams and compare their performance under those conditions.

This thesis presents a simulator that makes it possible to conduct simulation runs with (possibly automated) changes in the environment or failures within the robots. As a basis, the *Swarmulator*, a simulator for swarm robotics, which was developed by Martin Burger within the scope of his diploma thesis [Bur12] is used and contributed the main part of the internal logic of the simulator. However, it needed to be extended to design experiments with the aforementioned environmental changes or robotic failures. The ALLIANCE architecture, as introduced by Parker in [Par98] is implemented as the mechanism for the coordination of a team of robots. Additionally, this thesis also discusses the experimental results of a case study, which has been adopted from Parker [Par98].

Acknowledgements

I would like to thank Prof. Dr. Martin Wirsing for giving me the opportunity to write this bachelor thesis, which has introduced me to a very interesting field of research.

I would especially like to thank the supervisors of my thesis, Annabelle Klarl and Christian Kroiß, for their commendable support. Their continuous feedback in our meetings and through various emails helped me to a great extend to always improve the quality of my thesis.

Additionally I would like to thank Martin Burger for developing the Swarmulator, which I partly used as a basis for my work, as well as for supporting me whenever I had questions or problems with it.

Contents

1 Introduction **1**
1.1 Objective 1
1.2 Outline 2

2 Foundations **3**
2.1 Concepts 3
2.1.1 Intelligent Agents and Robots 3
2.1.2 Environment 4
2.1.3 Mission 4
2.1.4 Task 4
2.1.5 Multi Robot Task Allocation 5
2.2 ALLIANCE Architecture 5
2.2.1 Overview 5
2.2.2 Task Selection 6
2.3 Summary 9

3 ALLIANCE Simulator **11**
3.1 Module Overview of the Simulator 11
3.2 Module `swarmulatorCore` 11
3.2.1 Overview of the Swarmulator 12
3.2.2 Extending the Swarmulator 14
3.2.2.1 Obstacle 14
3.2.2.2 Interruption Handler Mechanism 14
3.3 Module `allianceCore` 15
3.3.1 `BehaviorSet` 15
3.3.2 `MotivationalBehavior` 16
3.3.3 Robot 17
3.3.4 Communication 17
3.4 Module `allianceWasteMission` 18
3.4.1 Requirements 18
3.4.2 Implementation 19
3.5 Summary 19

4 Case Study **21**
4.1 Waste Cleanup Mission 21
4.1.1 The Environment 21
4.1.1.1 The World and its Components 22
4.1.1.2 The Robots 23
4.1.2 The Mission 23

4.1.2.1 *find-locations* 24
4.1.2.2 *move-spill* 26
4.1.2.3 *report-progress* 26
4.2 Case-specific Implementations 26
4.2.1 Obstacles and Obstacle Avoidance 27
4.2.2 Interruption Handler 28
4.2.3 Communication 28
4.3 Experiments 29
4.3.1 No Interruption 30
4.3.2 Trapping one Robot 31
4.4 Evaluation of the Extended Simulator 33
4.5 Summary 34

5 Conclusion **35**

List of Figures **37**

Content of the CD **39**

Bibliography **41**

Chapter 1

Introduction

Since the establishment of artificial intelligence in the middle of the 20th century, a vast variety of robotic systems have been researched, designed and actually built. Nowadays, nearly every aspect of our life can be aided by some kind of robot. While the usage of robots in the private life is yet mostly marginal, the industrial and military sectors greatly use robotic systems for various applications. In a lot of cases, robots are either used because of the degree of danger or the highly repetitive nature of the application and therefore to reduce the need for human workers.

Generally, there are two approaches when designing a robotic solution for an application. On the one hand, a single robot could be used, that has all the capabilities to perform the desired tasks. On the other hand, a team of multiple robots may be used, where each robot has limited capabilities. Furthermore, the robotic team may be designed redundantly in their skills, to address the issues of uncertainty and failures within the robots' sensors and actuators. Obviously, if only one robot is used and that robot fails, it would have disastrous consequences. On the contrary, a team of robots may be able to compensate the failure of a team member to some degree to still satisfactory achieve its purpose.

One of the key issues which has to be addressed in the design of multi-robot systems is the coordination of the robotic team. Each robot needs to know when it has to execute which task. This problem is commonly referred to as *multi robot task allocation*. At some point, this mapping of robots to tasks has to be determined and ideally it should optimize the performance of the robotic team, considering some criteria such as time- or cost-efficiency.

1.1 Objective

Since real-world experiments are usually cost- as well as time-extensive, simulation plays a crucial role in the development of multi-robot systems. The goal of this thesis is to develop a simulator that uses the so called ALLIANCE architecture, developed by Parker [Par98], for task allocation. ALLIANCE is a fully distributed and behavior-based approach, that uses mathematically modeled motivations for task allocation. Due to its design, it allows the robots to effectively and efficiently respond to unexpected environmental changes or to failures within the robots, such as faulty sensors or the complete malfunction of a team member.

Since Martin Burger just finished the main work on the *Swarmulator*, a simulator for swarm robotics, which he developed within the scope of his diploma thesis [Bur12], this simulator is used as a basis. As it is of high interest to observe how the robotic

team responds to changes and failures, the simulator is extended to interrupt simulation runs during the runtime in order to create these changes or failures.

1.2 Outline

Chapter 2 establishes the necessary foundations for this thesis. First, some basic concepts of the field of multi-robot systems are introduced, followed by a presentation of the ALLIANCE architecture.

The next two chapters completely describe the developed simulator. Chapter 3 presents a general overview, while also giving a description of the underlying framework of the simulator, which is based on the *Swarmulator* by Martin Burger [Bur12] and the implementation of the ALLIANCE architecture. It also details the aforementioned extensions to the *Swarmulator*.

Chapter 4 contains the implementation of a case study. After presenting the scenario, which is adopted from an experiment conducted by Parker [Par98] and some case-specific implementations, it also discusses some experiments and their results. Based on those, an evaluation of both the simulator and the case study is given.

Finally, chapter 5 briefly summarizes this thesis and suggests future work on the simulator.

Chapter 2

Foundations

For a thorough understanding of the ALLIANCE architecture, its implementation in this thesis and its general context in artificial intelligence, it is important to take a deeper look at some concepts in the field of multi-agent/multi-robot systems. This chapter first introduces all concepts that are used later on and secondly gives a brief, but sufficiently detailed insight into ALLIANCE.

2.1 Concepts

In order to introduce the concepts, this section begins with an explanation of *intelligent agents*, followed by a specification of the *environments* agents typically live in, both based on chapter 2 of *Artificial Intelligence: A Modern Approach* [SR10]. Next, the *mission* and subsequently the agents' *tasks* are defined, corresponding with their usage in *ALLIANCE: An Architecture for Fault Tolerant Multi-Robot Cooperation* [Par98]. Lastly, a short description of *task allocation* is given.

2.1.1 Intelligent Agents and Robots

Russell and Norvig [SR10] define an *intelligent agent* as "anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators." They also state some examples: A human agent uses eyes, ears and other organs as sensors and hands, feet, mouth and other bodily parts as actuators. A robotic agent could use cameras and infrared range finders as sensors and could act using various motors, while a software agent may use keystrokes, file content and network packets as sensory input and may act by displaying something on a screen, writing files or transmitting network packets.

Furthermore, Russell and Norvig introduce two concepts to specify an agent's choice of action at any given moment: An agent's behavior is described by its *agent function*, which maps any sequence of sensory input to an action. An implementation of this mathematical function is called *agent program*.

To answer the question whether or not an agent 'does the right thing', Russell and Norvig are using *performance measures*. A sequence of actions by an agent causes its environment to go through a sequence of states. If this sequence of states is desirable, the agent performs well and therefore 'does the right thing'.

Since the focus of this thesis lies in the simulation of robots and multi-robot systems, the following terms should also be defined: A *robot* is an embodied agent that is placed

in a real-world (i.e. physical) environment. Analogical, a *multi-robot system* is a multi-agent system that uses exclusively robots as intelligent agents. As the name implies, a multi-agent system is a composition of multiple agents that interact to achieve a pre-defined goal.

2.1.2 Environment

As one can easily imagine, the range of environments agents perceive and act on is very diverse. For a scientific categorisation of environments, Russell and Norvig [SR10] identified a small number of characteristics. In the following, those characteristics that are most relevant to this thesis are introduced.

If an agent can perceive every aspect of the environment that is relevant to the action selection, the environment is called *fully observable*. Otherwise, it is called *partially observable*. Fully observability is convenient, because the agent neither has to explore the world, nor does it need to maintain an internal state to keep track of it.

The second characteristic concerns the number of agents that interactively live in the environment. If there is only one agent, the environment is called *single agent*, while it is called *multiagent* when there are multiple agents. An object is treated as an agent if its "behavior is best described as maximizing a performance measure whose value depends on [another agent's] behavior" [SR10, p. 43]. Multiagent environments can further be characterized as *cooperative*, where agents try to maximize the performance measure of all agents, or *competitive*, where agents try to maximize only their own performance measure while possibly minimizing those of other agents.

Environments can be *sequential* or *episodic*. If an agent's action influences its future decisions, the environment is sequential. In an episodic environment, the agents actions can be broken down into episodes, consisting of some sensory input and one action, where the action selection does not depend on previous decisions.

It could occur that the environment changes while the agent is deciding on an action. If that is the case, the environment is *dynamic*. On the other hand, an environment is *static*, if it only changes when an agent performs an action.

2.1.3 Mission

To intelligently determine which actions to choose, an agent needs some kind of goal to achieve. In cooperative multi-robot systems, this goal is represented by a mission the robots have to fulfill as a team. This mission is a composition of an arbitrary number of loosely coupled tasks that may have ordering dependencies. An example of coupled tasks can be found in section 4.1.2, where one task can only be processed after it has received input from another task.

2.1.4 Task

While traditional task allocation defines a task as "a subgoal that is necessary for achieving the overall goal of the system, and that can be achieved independently of other subgoals (i.e. tasks)" [BPG04, p. 1], in the context of ALLIANCE, it is sufficient to allow tasks to be loosely coupled and to have ordering dependencies. Therefore a task may be informally described as a sequence of actions (such as move to location (x, y), then pick up object o), performed by one or more robots. This slightly more lenient definition is sufficient, because the approach of the ALLIANCE architecture

(see sections 2.2.1 and 2.2.2) differs from traditional task allocation [Par98, p. 5–6], as it is described in section 2.1.5.

A differentiation between *high-level tasks* and *low-level tasks/competences* (also referred to as *lower-level robot control* and similar synonyms), can be found in various literature, such as [Par98], [BPG04], or [CHF04]. A high-level task is part of a mission, for example a task *locate food* in a *foraging* mission, whereas a low-level task describes some basic behavior that should often be constantly performed (e.g. obstacle avoidance). It is possible that a robot performs several low-level tasks at the same time. High-level tasks on the contrary, are usually performed sequentially by one robot, or simultaneously by multiple robots, where one robot can only perform one task at a time.

2.1.5 Multi Robot Task Allocation

With the concepts introduced above, it is possible to design a single agent environment and multiple tasks in order to assign a mission to that agent. However, there is one additional problem in multiagent environments, as the agents (in this case robots) need to decide "which robot should execute which task" [BPG03, p. 1], in order to properly cooperate and thereby to achieve a good performance. This issue is commonly referred to as *multi robot task allocation.*

Depending on the multi-robot system, different criteria may be used to assess the performance of the robotic team. Amongst those criteria are the minimization of the energy consumption of the robots or the time it takes the robots to accomplish the mission. Just as well, the performance of the team may be evaluated by considering the number of tasks each robot performs in a given time period. Often, a combination of multiple criteria have to be respected when designing a multi-robot system.

In order to find a suitable mapping of robots to tasks, the mission is usually decomposed into subtasks, hierarchical task trees, or roles. Traditionally, a central planner, or simply the designer then computes the robot to task mapping, based upon the robots' capabilities and the specified performance criteria. However, this approach often offers too little possibilities for re-allocation of the tasks, for example when robot failures occur.

2.2 ALLIANCE Architecture

Now that the necessary background for ALLIANCE is defined, a brief description of its architecture can be presented. The definition of *multi robot task allocation* in section 2.1.5 already mentioned the problem of re-allocation. In real-world applications, the design of a multi-robot system has to account for issues such as robot failures or a dynamic environment. The ALLIANCE architecture was developed with especially these problems in mind, as it allows the robots to respond robustly and reliable to failures at any time during a mission, for example within the robots or the communication and to adapt to a dynamic environment, changes in the team's mission or the composition of the robotic team, as stated by Parker [Par98, p. 2].

The following sections first give a general overview and then present a more detailed look at the method of task allocation, the formal model of ALLIANCE.

2.2.1 Overview

ALLIANCE is a fully distributed, behavior-based architecture for fault tolerant multi-robot cooperation. It uses two mathematically-modeled motivations – *impatience* and *acquiescence* – for task selection. The design focus was to create robot teams, that can operate successfully amidst uncertainties and failures concerning the robots' sensors and actuators, the action selection and execution, or amidst a dynamic environment, to just name a few examples. Since every robot is capable to determine its own actions, there is no need for a centralized control mechanism.

Every robot possesses several *behavior sets*, each corresponding to a high-level task achieving function. A behavior set can either be active or hibernating, but due to conflicting goals, only one behavior set can be active at a given time in one robot. Therefore, when one behavior set gets activated, it inhibits all other behavior sets within that robot from activation. But as described in section 2.1.4, the robot could continually perform a number of low-level tasks such as collision avoidance. For every behavior set, there is also a *motivational behavior*, controlling the activation of its assigned behavior set. Figure 2.1 illustrates this basic architecture.

ALLIANCE uses a fairly simple form of inter-robot communication, each robot informs the other robots only of its currently active behavior set via a broadcast message. Gerkey and Matarić [BPG03] described this as a "heartbeat message", broadcasted by each engaged robot. Hence, no two-way communications are needed and the communication overhead gets reduced.

At any given time, every high-level task the robot can perform at that moment is considered for (re)assignment. This means every motivational behavior is assessing a motivation for its behavior set, based on the current levels of impatience and acquiescence, while also considering the sensory feedback, inter-robot communication and cross-inhibition from other active behavior sets. If this motivation exceeds a given threshold, the according behavior set gets activated. For example, the motivation to execute a task that is currently performed by a teammate increases over time by the robot's own impatience and by the teammate's acquiescence. In the same way, the desire of a robot to furthermore execute a task decreases by its acquiescence and the impatience of its teammates.

This approach differs from traditional task allocation (as described in section 2.1.5), as the motivations are designed to allow robots to perform a task as long as their task execution shows the desired effect on the environment.

2.2.2 Task Selection

As mentioned in the previous section, the task selection occurs in the motivational behavior, based on a number of sources. This section, which is based on chapter III. D. of *ALLIANCE: An Architecture for Fault Tolerant Multi-Robot Cooperation* [Par98], now specifies these sources and defines a number of functions and parameters that are used to calculate the motivation to activate a behavior set. For this, Parker [Par98] defined the following problem: The set $R = \{r_1, r_2, \ldots, r_n\}$ represents the robot team, consisting of n heterogeneous robots performing the tasks $T = \{task_1, task_2, \ldots, task_m\}$, which compose the mission. The behavior sets robot r_i possesses are represented by the set $A_i = \{a_{i1}, a_{i2}, \ldots\}$. As one task could be performed differently by different robots, it is also necessary to have a way of referring to the task a robot is currently executing via its activated behavior set. The set $\{h_1(a_{1k}), h_2(a_{2k}), \ldots, h_n(a_{nk})\}$ is a set of functions, where $h_i(a_{ik})$ returns the task that behavior set a_{ik} is accomplishing.

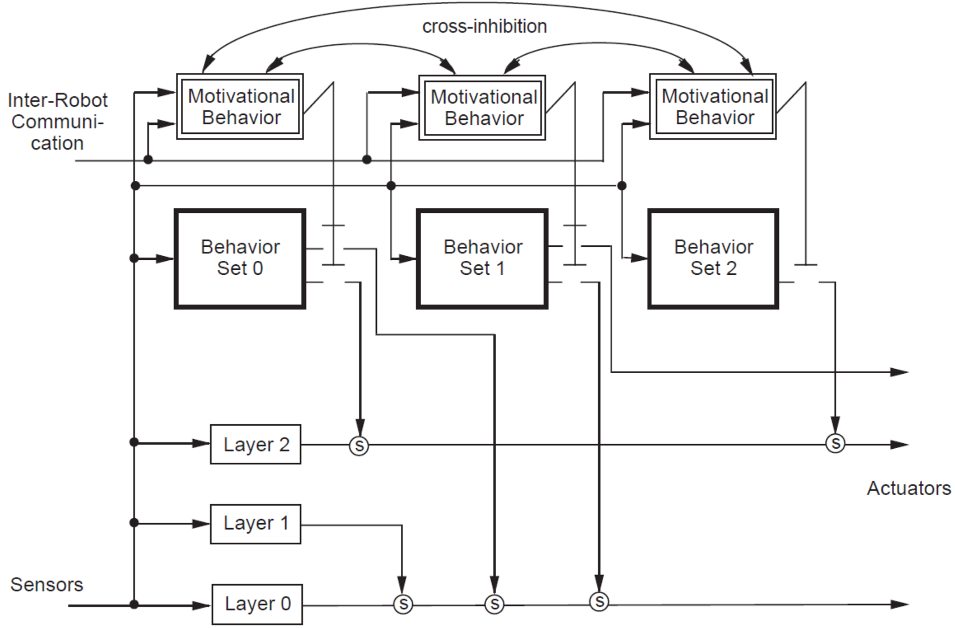

Figure 2.1: *The ALLIANCE architecture, as it is implemented on every member in the robot team. Each motivational behavior manages one behavior set, based on sensory and communicational input and on cross-inhibition when another behavior set is active. (taken from [Par98, p. 6])*

In the following, a number of parameters and functions are presented, which were all defined by Parker in [Par98].

The first parameter to be set is θ, the *threshold of activation*. If the motivation reaches or exceeds this threshold, the corresponding behavior set will become active. Only one global threshold is needed, because the rates of impatience and acquiescence can vary across behavior sets and robots.

As seen in figure 2.1, the motivational behavior needs some input concerning the *sensory feedback*, the *inter-robot communication* and the *cross-inhibition* between behavior sets. ALLIANCE defines the following functions to adress this issue:

$$sensory_feedback_{ij}(t) = \begin{cases} 1 & \text{if the sensory feedback in robot } r_i \text{ at time } t \text{ indicates that behavior set } a_{ij} \text{ is applicable} \\ 0 & \text{otherwise} \end{cases} \quad (2.1)$$

$$comm_received(i, k, j, t_1, t_2) = \begin{cases} 1 & \text{if robot } r_i \text{ has received message from robot } r_k \text{ concerning task } h_i(a_{ij}) \text{ in the time span } (t_1, t_2) \text{, where } t_1 < t_2 \\ 0 & \text{otherwise} \end{cases} \quad (2.2)$$

$$activity_suppression_{ij}(t) = \begin{cases} 0 & \text{if another behavior set } a_{ik} \text{ is active, } k \neq j \text{, on robot } r_i \text{ at time } t \\ 1 & \text{otherwise} \end{cases} \quad (2.3)$$

With function (2.1), the motivational behavior receives the necessary sensory feedback to determine whether its behavior set is applicable at a given time or not. Function (2.2) is utilized to monitor the communication messages, specifically to note when a teammate is executing task $h_i(a_{ij})$. Additionally, the parameters ρ_i and τ_i are applied for inter-robot communication. ρ_i specifies the broadcast rate, while τ_i gives the time after which r_i decides a specific team member has ceased to function, provided that r_i didn't receive any broadcasts from that robot in the time span τ_i. The cross-inhibition is achieved by the use of function (2.3). Whenever there is another active behavior set in the robot, this function returns 0.

To compute the current rate of impatience, ALLIANCE defines three parameters. $\delta_fast_{ij}(k,t)$ and $\delta_slow_{ij}(k,t)$ are the rates of impatience when no other robot is currently performing the task $h_i(a_{ij})$, respectively when one ore more team members execute the task of behavior set a_{ij}. Robot r_i may have different parameters δ_slow_{ij} for each teammate r_k. Likewise, r_i has a number of parameters $\phi_{ij}(k,t)$, which assign the time the robot is willing to let its motivation for behavior set a_{ij} be affected by communication messages from robot r_k concerning task $h_i(a_{ij})$. Two functions are defined, first (2.4), to compute the current rate of motivation, second (2.5) to reset the robots motivation of behavior set a_{ij} when he receives the first communication message from team member r_k concerning task $h_i(a_{ij})$:

$$impatience_{ij}(t) = \begin{cases} \min_k(\delta_slow_{ij}(k,t)) & \text{if } (comm_received(i,k,j,t-\tau_i,t) = 1) \text{ and } (comm_received(i,k,j,0,t-\phi_{ij}(k,t)) = 1) \\ \delta_fast_{ij}(t) & \text{otherwise} \end{cases} \tag{2.4}$$

$$impatience_reset_{ij}(t) = \begin{cases} 0 & \exists k.((comm_received(i,k,j,t-\delta t,t) = 1) \text{ and } (comm_received(i,k,j,0,t-\delta t) = 0)) \\ 1 & \text{otherwise} \end{cases} \tag{2.5}$$

As a robot shouldn't execute a task forever, there are two parameters, $\psi_{ij}(t)$ and $\lambda_{ij}(t)$, and one function, (2.6), to determine when the robot decides to acquiesce it. When a team member wants to take over task $h_i(a_{ij})$, robot r_i doesn't yield instantly, but rather continues the task execution for a period of time, given by $\psi_{ij}(t)$. Since a robot may perform a task he is not best suited for, there is a maximum period of time, given by $\lambda_{ij}(t)$, r_i wants to maintain behavior set a_{ij} active before giving up on the task, to possibly execute a task he is better suited for.

$$acquiesence_{ij}(t) = \begin{cases} 0 & \text{if ((behavior set } a_{ij} \text{ of robot } r_i \text{ has been active for more than } \psi_{ij}(t) \text{ time units at time } t) \text{ and } (\exists x.comm_received(i,x,j,t-\tau_i,t) = 1)) \text{ or (behavior set } a_{ij} \text{ of robot } r_i \text{ has been active for more than } \lambda_{ij}(t) \text{ time units at time } t) \\ 1 & \text{otherwise} \end{cases} \tag{2.6}$$

The functions defined above are now used to calculate the motivation of activation in function (2.7) of behavior set a_{ij} in robot r_i. The motivation initially is set to 0 and increases over time by some rate of impatience, given by function (2.4). It gets reset in turn to 0, when any one of the following events occur: The sensory feedback in function (2.1) indicates a_{ij} isn't applicable, r_i has activated another behavior set and therefore gets inhibited by function (2.3) to activate a_{ij}, the impatience needs to be reset as specified by function (2.5), or the robot decides to acquiesce the task in function (2.6).

$$
\begin{aligned}
m_{ij}(0) &= 0 \\
m_{ij}(t) &= [m_{ij}(t-1) + impatience_{ij}(t)] \\
&\quad * sensory_feedback_{ij}(t) \\
&\quad * activity_suppression_{ij}(t) \\
&\quad * impatience_reset_{ij}(t) \\
&\quad * acquiescence_{ij}(t)
\end{aligned}
\tag{2.7}
$$

2.3 Summary

This chapter presented the basic foundations that are needed for this thesis. First, some of the concepts from the field of multi-robot systems were introduced, followed by an insight into the ALLIANCE architecture.

The first two concepts *intelligent agent* and *environment* were defined by using the definitions of Russell and Norvig [SR10]. A robot was described as an embodied intelligent agent, which in turn can be seen as anything that interacts with its environment by using sensors and actuators. Also, several characteristics for the categorization of environments were presented, such as its degree of observability, or whether one or multiple agents live in it. As a team of robots needs a goal to achieve, the concept *mission* was defined as a composition of *task*s, that are performed by one or by multiple robots. Furthermore, tasks were classified into *high-level tasks* that are performed to accomplish a mission and *low-level tasks* such as obstacle avoidance. The last concept, *multi robot task allocation* was identified as a major problem in the design of multi-robot systems, as it needs to be specified when which task is executed by which robot.

In the second part of this chapter, the ALLIANCE architecture was presented as an motivation-based and fault tolerant approach to the problem of multi-robot task allocation. The mathematically modelled motivations were described, as well as the behavior sets and motivational behaviors, which specify how a single robot executes a task and when to activate or hibernate a behavior set respectively.

These foundations are used in the next chapter to adapt and extend an existing simulator for swarm robotics, so that experiments with the ALLIANCE architecture can be simulated.

Chapter 3

ALLIANCE Simulator

The main goal of this thesis was to implement the ALLIANCE architecture and to develop a framework that allows the design of experiments which use ALLIANCE for multi-robot task allocation. Since Martin Burger has just finished the main part of the *Swarmulator* [Bur12], a simulator to test mechanisms for task allocation in swarm robotics, this simulator is chosen as a foundation. To make experiments more realistic, it is extended in two aspects, namely the addition of obstacles and therefore the need for obstacle avoidance and the functionality to change the environment and create failures such as faulty robotic sensors or actuators during a simulation. The following sections first give a general overview of the modules which compose the simulator and then provide a deeper insight into each module.

3.1 Module Overview of the Simulator

The simulator is divided into the three modules `swarmulatorCore`, `allianceCore` and `allianceWasteMission`. This basic architecture is illustrated in figure 3.1. The module `swarmulatorCore` provides all the basic functionalities of the Swarmulator and therefore holds the internal logic of the simulator. The ALLIANCE architecture, as described in section 2.2, is contained in module `allianceCore`. Finally, the module `allianceWasteMission` contains an experiment, in this case a simulation of the laboratory version of a hazardous waste cleanup, as conducted by Parker [Par98].

Because of this architecture, it is relatively easy to extend this simulator. Additional experiments using ALLIANCE, can be either integrated into `allianceWasteMission`, or into an additional module similar to `allianceWasteMission` if this is necessary. Furthermore, it is possible to simulate experiments that don't use the ALLIANCE architecture. In this case, two modules need to be added. One, analogous to `allianceCore`, containing the basic framework that specifies the components of the world and defines their interactions, such as task allocation. The other containing one ore multiple experiments, analogous to `allianceWasteMission`.

3.2 Module `swarmulatorCore`

As mentioned earlier, the module `swarmulatorCore` contains the internal logic of the simulator and provides basic classes to design environments. With the exception of a few aspects that are presented in section 3.2.2, it is based completely on the Swarmulator, which was developed by Martin Burger as a simulator for swarm robotics within

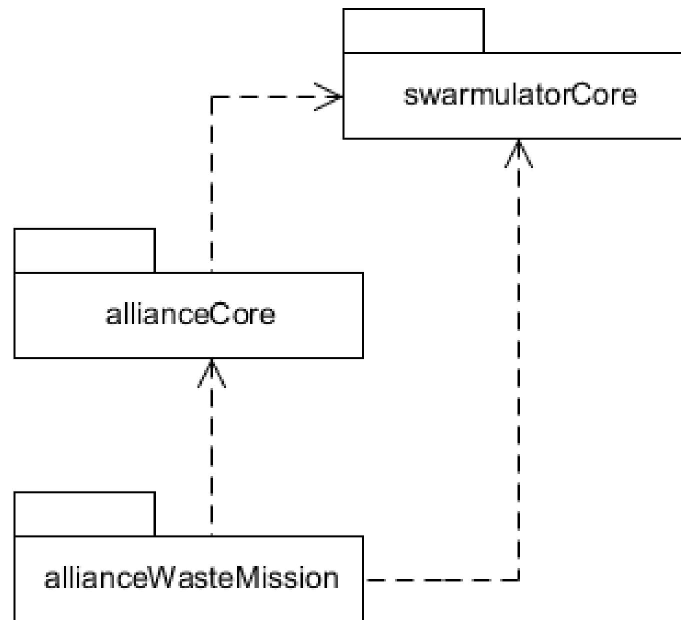

Figure 3.1: *The basic architecture of the simulator. The module* `swarmulatorCore` *provides the basic functionalities of the Swarmulator, while* `allianceCore` *and* `allianceWasteMission` *include the ALLIANCE framework and an experiment respectively.*

the scope of his diploma thesis [Bur12]. Although ALLIANCE is not an architecture for swarm type cooperation, but rather for "intentional" cooperation as Parker described it [Par98, p. 2], the Swarmulator can easily be adapted for ALLIANCE experiments, as it processes a stepwise computation of a virtual world for every simulation.

3.2.1 Overview of the Swarmulator

The Swarmulator provides the basic functionalities to simulate multiple experiments, each featuring a virtual world and an arbitrary number of objects that may interact with each other. It was developed to test mechanisms for task allocation in swarm robotics. Each simulation modifies one virtual world an its components, while optional csv-files can be produced that contain some statistical data. For this, the Swarmulator uses a model-view-controller architecture, which is illustrated in a simplified version in figure 3.2. The Swarmulator uses a factory to create a world, which is then associated to a thread, that computes steps for the world. At any point in the simulation, a view can be opened, which draws the current step of the simulated world.

The virtual world is represented in a class `World`. This class holds a reference to the floor of the world and it manages two lists, one for the `Components` that live in the world and one for the `Aspects` each `Component` may hold. These two classes are explained later on. Additionally, a `World` knows how much time has passed since the beginning of the simulation and every `World` also stores its pseudo random generator, which is instantiated at the creation of a new simulation with a user specified random seed. This random seed is also included in a separate file when csv-logging is enabled. This makes it possible to deterministically repeat simulation runs.

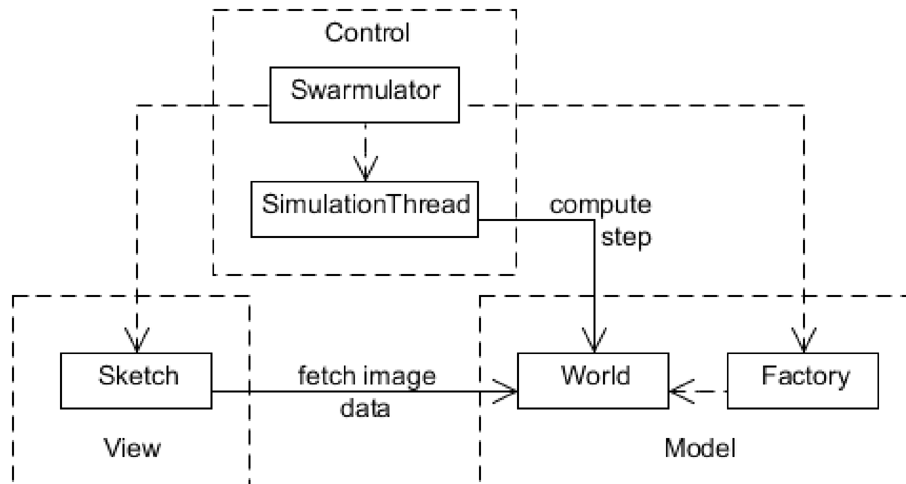

Figure 3.2: *Simplified representation of the model-view-controller pattern used by the Swarmulator. (Adapted from [Bur12])*

The class `Component` serves a basic class for all objects that live in a `World`. `Component`s have a unique id and are associated to exactly one `World`, where they reside at a specified position with a given orientation. Additionally, a `Component` manages a set of `Aspect`s, which may be used to simulate for example the energy consumption of robots or the temperature of objects, if these points should be regarded in the experiment. A `Component` provides methods to update its position and its orientation, but in order to sense other `Component`s, it needs a sensor. Hence, the Swarmulator also provides basic classes for sensors.

A simulation step gets processed in the method `computeStep(int millisGone)` of a `World`, where the parameter `millisGone` gives the time span of the step. First, all steps for the `World` are computed. For example, in a dynamic environment, this computation may create new obstacles. Then, all steps for the `Component`s are processed. This computation differs in complexity from simple objects that neither change nor interact with other objects and therefore need no computation steps, to robots, where complex mechanisms such as the ALLIANCE architecture need to be processed to decide their actions. Finally, all steps for the `Aspect`s are computed.

As illustrated by figure 3.2, a `World` gets created in a world factory, while its method `computeStep(int millisGone)` gets called by a `SimulationRun`, which is an implementation of the `Runnable` interface. To execute a simulation run, the Swarmulator starts a thread that computes the simulation steps via the run method.

It is possible to import world factories from jar-files that contain the other two modules. This makes it easy for users to "distribute their experiment's definition without having to include the simulation platform itself" ([Bur12]). Such a jar-file has to include a top-level file named "factories.cfg" that names the world factories it provides. With this, the Swarmulator is capable to locate the corresponding classes and the user can create new simulation runs with the imported world factories.

3.2.2 Extending the Swarmulator

To make experiments more realistic, the Swarmulator was extended in two aspects. Obviously, the robots should be prohibited from moving through obstacles such as walls. Additionally, there should be the possibility to intervene simulations, in order to change the environment or to create failures, such as faulty sensors or actuators within the robots. The second aspect is of particular interest for this thesis, because, as described in section 2.2, the ALLIANCE architecture is especially designed to allow the robots to respond to such environmental changes or robotic failures.

3.2.2.1 Obstacle

To incorporate obstacles and obstacle avoidance, two things have to be considered: First, every object that should be treated as an obstacle needs some property that allows other objects to identify it as such. Second, an object such as a robot needs a way to identify other objects as obstacles.

The following solution has been implemented to fulfill these requirements: Every `Component` now has a field called `penetrability`, which holds its degree of penetrability, specified in the following manner:

- The penetrability is given as a positive float value.
- If the penetrability is in the interval $[0.0, 0.1]$, the `Component` is interpreted as completely impenetrable.
- If the degree of penetrability is greater than 0.1, but less than 1.0, the `Component` is not impenetrable, but has a negative effect on the movement speed of the penetrating object. This can be used in environments that have objects such as water spills, or debris lying on the ground, which the robots can move over, but will be slowed down.
- If the penetrability equals 1.0, other objects can move over or through it with no effects on their movement speed. Usually the floor of a world is designed to have no speed-altering effects on the robots.
- If the penetrability is greater than 1.0, the penetrating `Component` will experience an accelerating effect on its movement speed.

A `Component` now also provides methods to indicate whether it is impenetrable and whether it affects the movement speed of other objects moving over it in a positive or negative way. This makes it possible to design sensors for robots, that not only are capable to sense obstacles and therefore allow the robot to avoid collisions, but also to sense objects that alter the robot's movement speed in a positive or negative way. With the latter characteristic, environments could be designed in which a robot would deliberately utilize components that have an effect on its movement speed. For example, a robot could search for routes that meet some criteria, such as finding the route with the shortest traveling time between two points.

3.2.2.2 Interruption Handler Mechanism

Every world now holds a list of `InterruptionHandler`s, which can be added in a factory. They are used to create and control changes to the environment or robotic

```
1 public interface InterruptionHandler {
2
3    public void triggerEvent(int millisGone);
4
5    public void setWorld(World world);
6 }
```

Figure 3.3: *Code-excerpt of the* `InterruptionHandler` *interface.*

failures. An `InterruptionHandler` is an interface, which contains two methods, as illustrated in figure 3.3.

One method, `public void triggerEvent(int millisGone)`, gets called in every simulation step of the world. With it, an `InterruptionHandler` decides whether to trigger its event and also specifies this event. The other method is used to set the world of an `InterruptionHandler` and gets called when one is added to a world.

Basically, two forms of `InterruptionHandler`s can be identified. One, which gets triggered by time, either periodically or at some specific time in the simulation. The other one gets triggered by some input, for example by a user via the GUI. The parameter `millisGone`, which defines the duration of the simulation step, can be used for periodically triggered `InterruptionHandler`s. For example, a time triggered interruption handler may be implemented, that changes the movement speed of a given robot at a specific point in time. The implementation of a user triggered interruption handler, which is used to set obstacles into the world during a simulation run, is presented in section 4.2.2.

3.3 Module `allianceCore`

The module `allianceCore` contains all classes that are essential for using the ALLIANCE architecture, as described in section 2.2. Combined with `swarmulatorCore`, these two modules provide all necessary functionality to design experiments using the ALLIANCE architecture and to simulate them. The following sections give an overview of the core classes.

3.3.1 `BehaviorSet`

The module `allianceCore` provides an abstract class `BehaviorSet`, which obviously is used to create specific behavior sets. An example for a concrete implementation of a behavior set is given later on. As described in section 2.2, every `BehaviorSet` is a concrete realization of a `Task`. The `Task` is mainly used just for reference in broadcasts, but it can also be prioritized and indicates whether multiple robots may perform it simultaneously.

A `BehaviorSet` can be either *active*, *hibernating* or *waiting*. The first two states are self-explanatory, an active behavior set gets executed, while the motivation of a hibernating one is below the threshold of activation. The last state is used, when robot r_i[1] wants to activate its behavior set a_{ij}, while another robot executes task $h_i(a_{ij})$. In this case, robot r_i is broadcasting messages to indicate that it wants to take over the respective task. When the other robot acquiesces the task, the `BehaviorSet` a_{ij} of robot r_i gets activated.

[1] For a specification of this notation, see section 2.2.2.

A concrete implementation of this abstract `BehaviorSet` needs to specify among other things, what the robot should do when it executes the associated `Task`. As different robots may have different ways to perform the same task, this is in no way determined by the `Task`, associated to the concrete `BehaviorSet`. For this purpose, the abstract method `public abstract void doAction(int millisGone)` is provided, where the parameter `millisGone` gives the duration of the simulation step.

3.3.2 `MotivationalBehavior`

The abstract class `MotivationalBehavior` is a basic implementation of the motivational behavior used in ALLIANCE, which was described in section 2.2.2. It obviously holds all parameters defined in that section, as well as a method for every function in section 2.2.2. Additionally, as some of the parameters are bound to other robots and as the computation of the motivation needs to check for received broadcast messages by other robots, every `MotivationalBehavior` manages a list of all team members. It is obvious, that a `MotivationalBehavior` also holds a reference to the behavior set it controls.

To compute the motivation of activation for its behavior set, the `MotivationalBehavior` provides a method, `public void computeMotivation(long time)`. Basically, it is an implementation of function 2.7 from section 2.2.2, where the parameter `time` gives the time of the simulation at which to compute the motivation. This method also checks, whether the state of the associated `BehaviorSet` needs to be changed. If the motivation exceeds the threshold of activation, the behavior set gets activated or is set waiting, when no other robot is performing the respective task or when another robot is performing the task, respectively. Otherwise, it is set hibernating.

A concrete implementation of such a `MotivationalBehavior` only has to specify the method `public boolean computeSensoryFeedback(long time)`. This method indicates whether the associated behavior set is applicable at the time given by the parameter `time`. For example, a behavior set may only be applicable when the robot possesses a specific sensor, or after a specific event has happened.

Parameter Setting

As ALLIANCE uses a lot of parameters for every `MotivationalBehavior`, the overall number of parameters which have to be set by the designer can be huge, depending on the number of robots in the environment and the amount of behavior sets of every robot. Additionally, if changes to the experiment are made, such as the exclusion of one robot, the parameters of all motivational behaviors need to be adapted, which can be fairly tedious. For this reason, the parameters can be set using a *.properties file*. The structure of this file is shown in figure 3.4.

As the specific value of the parameters have no relevance right now, the variable $\langle value \rangle$ is used. The variable $\langle motivationalBehavior \rangle$ denotes for which `MotivationalBehavior` this parameter is to set. As described in section 2.2.2, every robot may have different parameters `timeWillingToAffect` and `slowImpatience` for different team members. Therefore, those two parameters are also bound to a robot, as indicated by the variable $\langle robot \rangle$.

If now the experiment should be changed, say a robot or some motivational behavior may be excluded, the designer doesn't have to manually adapt all the parameters. As the parameters get set during the instantiation of a `MotivationalBehavior`, obviously

```
1 threshold = <value>
2
3 broadcastRate.<motivationalBehavior> = <value>
4 timeAllowedTillCeased.<motivationalBehavior> = <value>
5 timeWillingToAffect.<robot>.<motivationalBehavior> = <value>
6 slowImpatience.<robot>.<motivationalBehavior> = <value>
7 fastImpatience.<motivationalBehavior> = <value>
8 timeBeforeYielding.<motivationalBehavior> = <value>
9 timeBeforeGivingUp.<motivationalBehavior> = <value>
```

Figure 3.4: *The general structure of the properties file for the parameters.*

only those parameters will be loaded which are actually used in the experiment. Similarly, only as many parameters for `timeWillingToAffect` and `slowImpatience` are set as there are robots in the simulation. Again, every `MotivationalBehavior` holds a list of all team members.

3.3.3 Robot

The abstract class `Robot` can be extended for a concrete implementation of robots. As a `Component`, it is associated to a `World` and resides at some position in this world. Every `Robot` manages a list of its `MotivationalBehaviors`, as well as a `TeamBroadcast`, the basic class for a broadcast unit. The communication mechanism is detailed in the next section.

In the `Component`-inherited method `public void computeStep(int millisGone)`, which gets invoked in every simulation step, the motivations of all `Motivational-Behaviors` get calculated via the provided method. If a `BehaviorSet` gets activated during the computation, its method `public void doAction(int millisGone)` will be called. However, if the behavior set waits to be activated, it causes the agent to broadcast a message to indicate that the robot wants to take over a task which is currently performed by a team member. If the robot that is currently executing this task receives this broadcast message, it will acquiesce the task after the time period given by the parameter $\psi_{ij}(t)$ (the time before yielding).

3.3.4 Communication

In the ALLIANCE architecture, robots communicate by broadcasting messages. On the one hand, a robot broadcasts heartbeat messages at a pre-specified rate, to inform its team members when it takes over a task. This kind of message is used to reduce the need for perceptual abilities of the robots, as they get informed of their team members' actions. On the other hand, the broadcast messages may be used to propagate specific information between the robots, such as the discovery and the position of an object that is relevant to the mission.

To meet these requirements, an interface `IBroadcastMessage` is provided in the module `allianceCore`, which specifies the broadcast messages and also a class `Team-Broadcast`, which manages the broadcast messages and therefore serves as a basic class for a broadcast unit. As illustrated by figure 3.5, the interface contains a few methods that allow to extract specific information from the message.

These methods are self-explanatory. Obviously, `getTask()` returns the `Task` this message is about, whereas `getTime()` returns the time of the simulation this message has been broadcasted. Of course, `getRobot()` can be used to retrieve the `Robot` that

```
public interface IBroadcastMessage {

    public Task getTask();

    public long getTime();

    public Agent getRobot();

    public Object getMessageContent();
}
```

Figure 3.5: *Code-excerpt of the `IBroadcastMessage` interface.*

broadcasted the message. Lastly, a method to get the message content is provided, `getMessageContent()`. However, if the message is used to broadcast a variety of information, such as multiple coordinates, it may be easier to add methods in the concrete implementation of `IBroadcastMessage`, especially for the needed purpose.

For the heartbeat messages, a default implementation for broadcast messages is included, the `StringBroadcastMessage`. Its message content is represented by a single String. However, as the content is in essence irrelevant for the ALLIANCE functionalities for this kind of broadcast, the message content is only used for possible logging purposes.

The provided class `TeamBroadcast` is a basic implementation of a broadcast unit. In this simple form, it holds a list of `IBroadcastMessages`. Additionally, it provides methods to receive messages and to filter and retrieve the messages. This makes it possible for example, to return a list of broadcast messages from a specific robot that were received in a given time interval. Again, if messages with elaborate content are used additionally to the ALLIANCE-specific heartbeat messages, it may be necessary to extend this class for advanced filtering methods.

In the basic `TeamBroadcast`, the messages are stored forever. This obviously creates performance issues, which should be addressed in an extended broadcast unit. Although most of the messages become obsolete very quickly, it needs to be considered that some aspects of ALLIANCE require information whether a robot has received its first broadcast message concerning a given task from a team member.

3.4 Module `allianceWasteMission`

The last of the three modules contains the specific classes of one or multiple experiments. This section presents some of these classes and basically gives instructions how to implement a new experiment in this framework. The next chapter presents the concrete implementation of this module.

3.4.1 Requirements

Obviously, an experiment can only be conducted if it is sufficiently specified. In order to design a world, the environment has to be defined, according to the characteristics of section 2.1.2. Every component that should be used in the experiment has to be specified, such as obstacles or items a robot could pick up, but also the robots themselves. For the latter, a detailed description of their sensors and actuators is absolutely essential. Of course, a mission for the robot team and therefore a number of tasks have

to be created. Additionally, the concrete behavior sets and motivational behaviors have to be designed, according to the tasks and the abilities of the robots. An example of such a specification is given in chapter 4.

3.4.2 Implementation

Assuming the complete specification as described in the previous section is given, now the experiment can be implemented. First, all the behavior sets and motivational behaviors are implemented, based on their respective abstract classes that were described in section 3.3. To decide whether its behavior set is applicable, the motivational behavior can use varying criteria, such as capabilities or knowledge of a robot. A behavior set might only be applicable for example, if the robot possesses a specific sensor, or if it knows the coordinates of a specific component. The latter condition may be accomplished using extended broadcast messages, as it is the case in the experiment which was conducted in chapter 4. In such a case it might be useful to extend the functionality of the `TeamBroadcast`, as indicated in section 3.3.

Second, the sensors have to be implemented, followed by the robots, which are based on the provided abstract class `Robot`. That class obviously should hold all the sensors the specific robot should have. Additionally, it should provide methods for the actions this robot can perform. For example, this class needs to specify how a robot picks up objects, if it is designed to do so. Furthermore, since low-level tasks (such as obstacle avoidance) are not modelled as behavior sets, they also have to be implemented in this class.

Of course, if it is a dynamic environment, or if the experiment should include failures such as faulty sensors or actuators, the appropriate handlers need to be created, implementing the `InterruptionHandler` interface.

After all `Component`s of this experiment are implemented, the world itself can be realized by extending the `IWorldFactory` interface, provided by the module `swarmulatorCore`. In its method `createWorld(String logFolder, long randomSeed)`, all necessary classes for the experiment are instantiated. Due to dependencies, the tasks should be created first, followed by the robots and their respective behavior sets. As the motivational behaviors hold references of all robots and their behavior sets, they should be instantiated last.

3.5 Summary

The basic overview of the simulator has been presented in this chapter. It consists of three modules, `swarmulatorCore`, `allianceCore` and `allianceWasteMission`. The first two modules were described in detail, while the last module was only informally pictured, since it will be detailed in the next chapter.

As a basis for the simulator, the *Swarmulator* by Martin Burger [Bur12] has been used, whose core classes are found in the first module. The *Swarmulator* has also been extended, to model obstacles and to interrupt simulation runs at runtime to change some conditions.

The second module contains the basic functionality of the ALLIANCE architecture, as it was described in the previous chapter. It provides basic classes for behavior sets, motivational behaviors, robots and the communication system.

The last module, which contains an implementation of a specific case study, is presented in detail in the next chapter.

Chapter 4

Case Study

Now, that the framework of a simulator for the ALLIANCE architecture has been laid out in the previous chapter, an experiment can be implemented. The chosen scenario for that experiment is the *hazardous waste cleanup mission*, as it was conducted by Parker [Par98]. Although Parker used real robots in a laboratory version of this scenario, the results should be somewhat comparable, as the simulation closely mirrors the laboratory version. But it should be noted, that the parameters used for the motivational behaviors are not optimized. It was not a goal of this thesis to produce an elaborate set of parameters.

Corresponding to the overall architecture described in section 3.1, the implementation of this experiment is included in the module `allianceTest`.

This chapter is structured as follows: First, the waste cleanup mission is specified, as required by section 3.4. Second, some case-specific implementations are given, followed by an overview at the experiments which were conducted. Last, an evaluation of the ALLIANCE architecture is given.

4.1 Waste Cleanup Mission

Before any specific classes can be implemented, the experiment has to be specified. The following sections first define the environment, including the world and its components and in a specific section, the robots. After that, the mission is described, by breaking it down into its tasks, just as it is defined in section 2.1.3. In a nutshell, the *waste cleanup mission* demands from a team of robots to find spill locations and subsequently move spill items to a desired location, while periodically reporting the progress to a human who is monitoring the experiment.

4.1.1 The Environment

According to section 2.1.2, the environment of the *waste cleanup mission* can be characterized in the following way: As the robots' sensors have only limited range, the environment is *partially observable*. One of the assumptions made by Parker is, that the robots on a team do not lie and are not intentionally adversarial [Par98]. Since there is only one team of robots that all work together, the environment is obviously *multiagent* and clearly *cooperative*. Section 4.1.2 will further illustrate, that this environment is *episodic*, because the currently selected action of a robot does not influence its future actions. Lastly, it is *dynamic*, as the simulation can be interrupted via the *interruption handler mechanism* as described in section 3.2.2.2, for example to set obstacles.

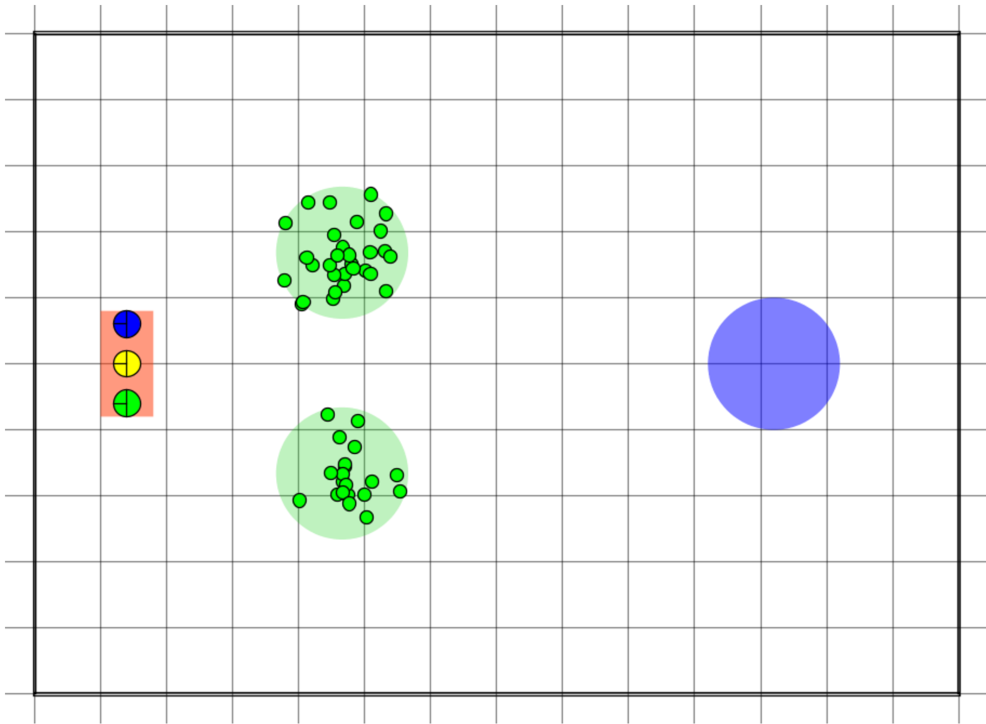

Figure 4.1: *The experimental mission at the beginning.*

4.1.1.1 The World and its Components

The world of these experiments consists of a floor and a rectangular border. The floor has no effect on other components whatsoever, while the border effectively limits the area for components to live in. The world contains three robots, which are described in more detail in the following section. Furthermore, there are two initial spill locations, which contain an arbitrary number of waste items, the desired final spill location (also referred to as waste deposit) and a site from which to report progress. This site simply changes its color in the graphical view, to indicate that a robot has just reported the progress. To retrieve actual data of the simulation, the log files can be checked. The coordinates relative to the border are as follows: The spill locations are located at one third of the border's width and at one and two thirds of the border's height respectively. The waste deposit is located at four fifths of the border's width at the center of the room. This information is used later on when defining the behavior sets of the robots, as they don't know the exact coordinates of the locations, but only these relative coordinates.

This initial world is illustrated in figure 4.1. The rectangle to the left is the report progress site, on which the three robots start their mission. They all initially face the left wall of the border. The three large circles represent the initial (in the middle) and desired final (to the right) spill locations, respectively, while the huge number of small green circles are waste items, which are located at random positions, but exclusively inside the initial spill locations.

Figure 4.2 shows a world after some simulation time. The green robot has just reported the progress at the respective site, which changed its color to indicate that a progress report has been made. The other two robots are currently carrying waste items and are on their way to the waste deposit, where already a number of waste items have been dropped. The dark rectangle to the upper-left of the blue robot, which

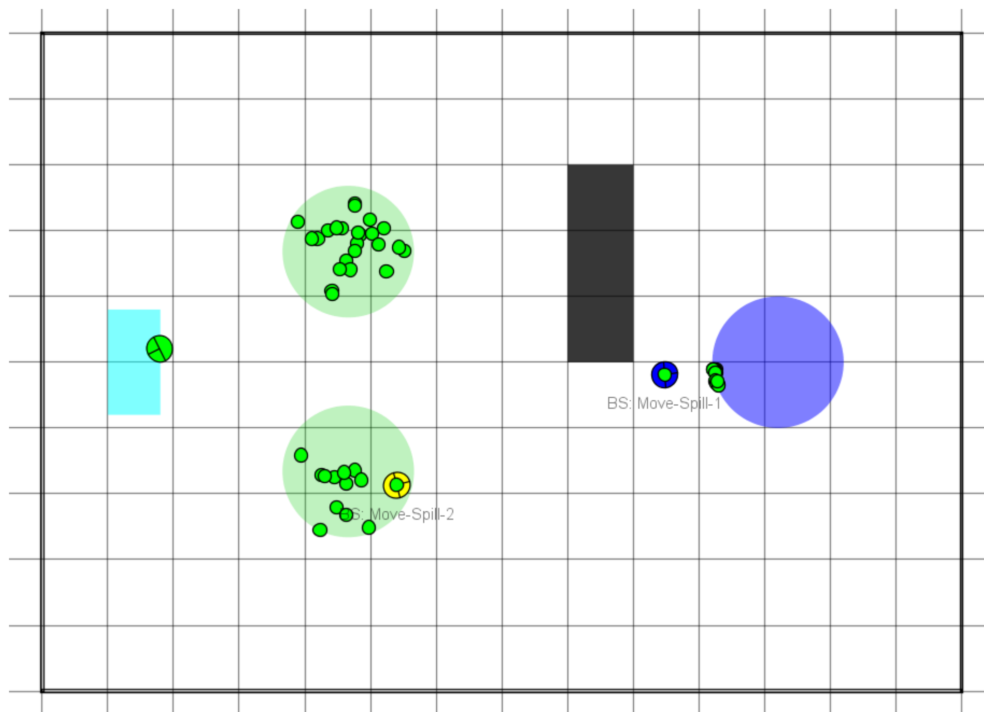

Figure 4.2: *The experimental mission after some simulation time.*

currently performs the behavior set *BS: Move-Spill-1*, is an obstacle, that obviously has to be avoided while moving between the upper initial and the final spill locations. This obstacle has been added during the simulation run by using a specific interruption handler that is presented in section 4.2.2.

4.1.1.2 The Robots

This experiment uses homogeneous robots with similar abilities. However, only one of them, the blue one, is designed to have side sensors for border detection. This grants it the ability to follow the walls of the border. This property will be utilized in the next section with a special behavior set.

The robots all share the same set of specifications, which are now defined. A robot is of circular shape and can turn on the spot, while it only can move in the direction it is looking at. Additionally, a robot can pick up a waste item from the floor, carry it and drop it onto the floor. Every robot comes with a variety of sensors, including a sensor for waste items, sensors for the spill locations and the report progress site, a sensor to detect the border and a sensor for collision avoidance. As mentioned earlier, the border detecting sensor can differ in its functionality, as one robot can detect the border to its front and to its side, while the other two only can detect the border to their front. According to sections 2.1.4 and 2.2.1, the low level *obstacle-avoidance* is not designed as a behavior set, but will be modelled within the robots. Section 4.2.1 describes the obstacle avoidance as it is used in this experiment.

4.1.2 The Mission

The mission the just defined robot team has to achieve is composed of the following distinct tasks: As the robots do not know the exact coordinates of the initial and

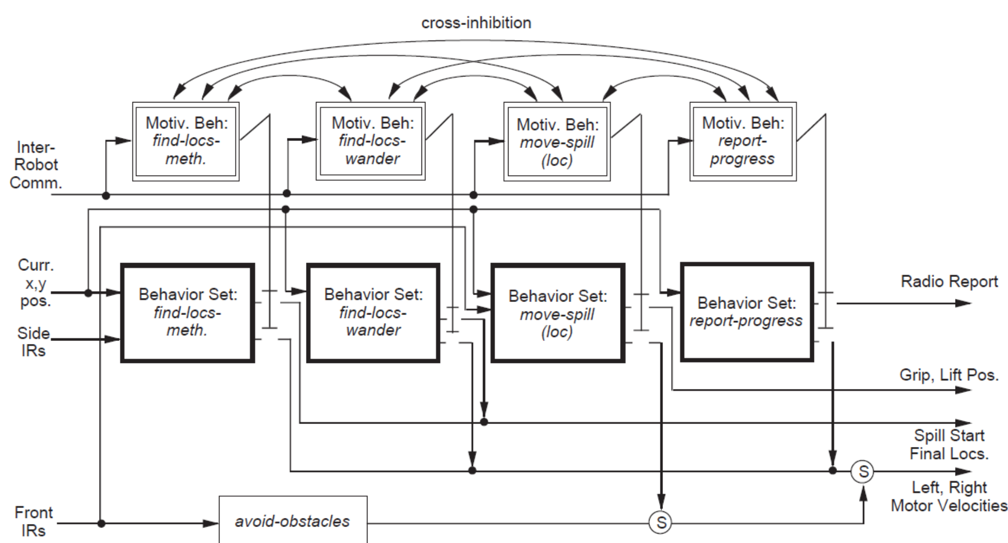

Figure 4.3: *Organization of the concrete motivational behaviors, the corresponding behavior sets, the low-level task* avoid-obstacles *and the various inputs and outputs of the hazardous waste cleanup mission. The input* Side IRs *indicates whether the robot can detect the border to its side, while the output* Radio Report *refers to the action a robot performs when it is reporting the progress. (taken from [Par98, p. 12])*

the desired final spill locations, but only their positions relative to the border, they need to explore the minimum and maximum coordinates of the room to calculate the absolute positions of the locations. This task is referred to as *find-locations*. The second task is actually split into two, *move-spill(top)* and *move-spill(bottom)*, where *top* and *bottom* refer to the according spill locations. This task, where the robots move the spill locations to the waste deposit, forms the main part of the mission. The last task, *report-progress*, requires the robots to periodically report the progress at the respective site. To minimize interference among the robots, every task can only be executed by one robot at a time.

These tasks will now be described in more detail. Additionally, for every task, the corresponding behavior sets will be defined. As the only competence of the specific motivational behaviors is only to decide whether their behavior set is applicable or not, they are not presented in the following sections. Instead, the descriptions of the behavior sets also specify their individual applicability. The emerging behavior organization for the hazardous waste cleanup is shown in figure 4.3.

4.1.2.1 *find-locations*

As mentioned earlier, the robots do not know the exact positions of the initial spill locations and the waste deposit. However, they have qualitative information about those positions, such as "the top spill is located in the upper half of the room, at one third of its width". All of this information is given relative to the border, therefore the robots first have to find out the minimum and maximum coordinates of it, before they can calculate and communicate the positions of the locations to the other team members.

Since the robots can have different sensors to detect the border, two behavior sets are designed, *find-locations-methodical* and *find-locations-wander*, to realize this task.

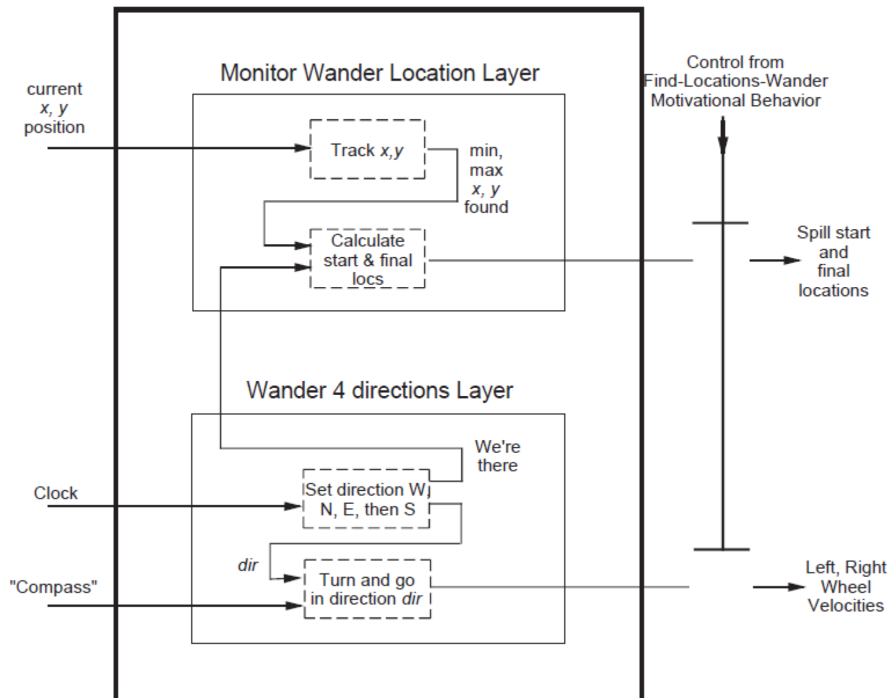

Figure 4.4: *The organization within the* find-locations-wander *behavior set, as described in section 4.1.2.1. (taken from [Par98, p. 13])*

Both of them require, that the workspace is of rectangular shape and that the walls of the border are parallel to the axes of the global coordinate system.

The methodical version to find the spill locations requires furthermore, that the robot is able to detect the border to its side. Therefore, this behavior set is only applicable if the robot in fact has intact side sensors. When activated, it causes the robot to perform the following actions: First, the robot moves to the left side of the border. Every time the robot detects a wall of the border in front, it turns 90 degrees clockwise and follows the just discovered wall with its side sensor. The robot also constantly keeps track of the minimum and maximum x and y coordinates it discovers. When all four walls of the border have been detected, the robot calculates the spill locations based on these x and y coordinates and on the qualitative information it has been given. These locations are then communicated to the other team members via a special broadcast message, which is described in 4.2.3.

The wander version of finding the spill locations avoids the need for a side sensor in the following way: When activated, it causes the robot to wander in the four directions west, north, east and south, each for a fixed time period. Again, the robot tracks the minimum and maximum x and y positions and calculates and communicates the locations when the wandering period is finished. Figure 4.4 illustrates this behavior set. As this version is less reliable than the methodical one, its impatience rate as defined in section 2.2.2 should be lower than that of the behavior set find-locations-methodical.

Both behavior sets use the same method of obstacle avoidance. When a robot discovers an obstacle at the wall, it avoids it by moving in direction of the center of the room. At the moment the obstacle is cleared, the robot continues its movement in the original direction, without returning to the specific border.

4.1.2.2 *move-spill*

After the locations have been found, the robots can begin to move waste items from the initial spill locations to the desired final location, the waste deposit. For this purpose, the behavior set *move-spill-(loc)* is defined. It is applicable whenever there are waste items at the spill location *loc*, when the initial and final spill locations are known and when no other robot currently works on that spill location. When this behavior set is activated, it causes the robot to execute the following actions: First, if the robot is not already there, it moves to the initial spill location denoted by *loc*. At the spill location, the robot activates its waste sensor to retrieve the position of a nearby waste item. Subsequently it moves to the location of that waste item and picks it up. While carrying the waste item, the robot moves to the desired final spill location. Upon arriving at the waste deposit, the waste item gets dropped. Again, to minimize interference among the robots, only one robot at a time works on the given spill location.

4.1.2.3 *report-progress*

The task *report-progress* requires the robot team to periodically report the progress of the mission to a human monitoring the system. In this experiment it is sufficient to just indicate that the report has occurred, as it is done via the change of colors of the report progress site. To gather actual information of the progress, one could simply open the view of the simulation of interest or check the various log files after the simulation is finished.

This task only needs to be done by the team as a whole, which means it is sufficient that only one robot reports the progress, while the others continue with the execution of their respective tasks. Additionally, the progress report only has to be done at an approximate time period. Therefore it is adequate, that the corresponding behavior set will only be applicable when it's time for another report, but the actual report will be done after a robot has activated that behavior set and moved to the report progress site. If that robot is at a location far away from the report progress site, the report will be delayed by the time it takes the robot to activate the behavior set and to move to the site.

To perform this task, the behavior set *report-progress* is defined, which, after the locations are discovered, is periodically applicable as specified by the given time period. It causes the robot to move to the vicinity of the report progress site and inform the humans monitoring the system of the progress of the mission by causing that site to change its color for a short period of time.

4.2 Case-specific Implementations

Section 3.4 already presented some of the classes needed for the implementation of an experiment. As the hazardous waste cleanup is now properly specified, a detailed look at some specific implementations can be given. This section focuses on the extensions that were made to the Swarmulator, namely the addition of obstacles and therefore the need for obstacle avoidance, the possibility to interrupt the simulation and change the conditions, and on the inter-robot communication, the broadcast system.

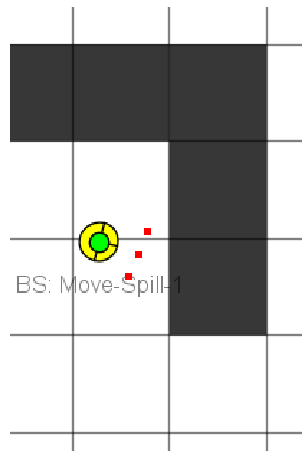

Figure 4.5: *The collision avoidance sensor checks three points in front of the robot, indicated by the three dots.*

4.2.1 Obstacles and Obstacle Avoidance

It was decided, that for the simulation of the hazardous waste cleanup mission, obstacles of a simple rectangular shape which are parallel to the axes of the global coordinate system are sufficient. This allows some simplifications for the obstacle avoidance, as can be seen later on in this section. Obviously, obstacles are extended `Components`, which are placed in the currently simulated world. They have a penetrability of 0.0, which makes them completely impenetrable. However, an obstacle may overlap other components. It should be noted that obstacles can be set anywhere in the world, including the space outside the border. The position of an obstacle is given by the coordinates of its upper left corner and its width and height which can be of arbitrary value.

As noted on multiple occasions before, the *obstacle-avoidance*, as a low-level task is not modelled as a behavior set. In this case, it is implemented directly into the robot's method `public void moveWithSpeed(float speed, int millisGone)`, which is the basic movement method and causes the robot to move in its current direction with the specified speed. Every time this method gets called to move a robot in the current direction, the following steps are regarded to ensure that it doesn't hit an obstacle.

At any time, the robot checks if there is an obstacle in front of him. To do so, the `CollisionAvoidanceSensor` is used, which looks for three points in front of the robot, if any of them touches an obstacle. The three points are located directly in front of the robot and slightly to the left, respectively to the right, depending on the robots radius. This can be seen in figure 4.5. When the sensor detects an obstacle, the robots orientation is aligned so that it is parallel to the side of the obstacle. Due to the aforementioned simplifications, this is realized by setting the orientation to one of the four directions (north, east, south, west). The robot also remembers that it is just avoiding an obstacle.

When the robot is currently avoiding an obstacle, its orientation remains aligned to it as long as the obstacle lays between the robot and its target. As soon as the robot clears the obstacle, it checks the position of the target in relation to the robots current (still aligned to the obstacle) orientation. If it lays in the back of the robot[1], the next side of the obstacle has to be avoided. Therefore its orientation is rotated 90 degrees

[1] based upon an imaginary line through the center of the robot

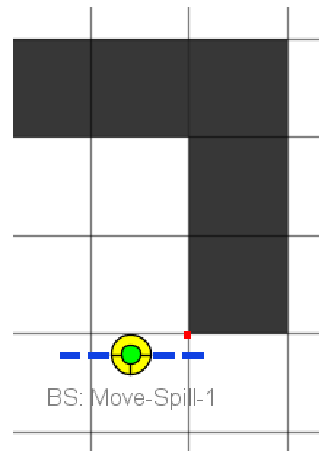

Figure 4.6: *The points that are checked by the side sensor of the collision avoidance include the one indicated by the dot directly at the edge of the obstacle. If that point is outside of the obstacle, the robot determines its new orientation, based on whether the target lays in its back or in its front. This is illustrated by the dashed line, which is drawn from left to right through the robot. In this case, if the target is somewhere above that line, the robot turns 90 degrees counterclockwise as it still needs to avoid the obstacle, otherwise it can freely move to the target.*

to align it to that side of the obstacle. This is shown in figure 4.6.

4.2.2 Interruption Handler

The implementations of the `InterruptionHandler` can be divided into two categories. The first type are interruption handlers that get triggered via some time related criteria, whereas the second type of interruption handlers get triggered via a user input. A time triggered interruption handler can either count the time with the parameter `millisGone`, to possibly trigger its event periodically. Alternatively, the world time may be queried.

In the following, a closer look at the class `UTObstacleInterruption` is presented, as shown by figure 4.7. It realizes a user triggered handler that sets obstacles. This interruption handler holds a list of rectangles, which represent the obstacles that are to set when it gets triggered and a variable, `trigger`, to indicate that the event should be triggered. Two methods are of particular high interest, since they determine the behavior of this handler.

The specific values for the rectangles are entered via a graphical user interface, which then calls the method `setTrigger(...)`. This graphical interface can be seen in figure 4.11. The user may add an arbitrary number of obstacles, which are stored in the list of rectangles, managed by this class. In the next step of the simulation, the inherited method `triggerEvent(int millisGone)` is called, where the obstacles are instantiated. This means, when the simulation is paused, the user who set the obstacles can't see them instantly, as they are not yet instantiated and therefore no image data has been updated for the view.

4.2.3 Communication

Section 3.4 already described the typical steps when implementing the broadcast system for an experiment. This section takes a closer look at the broadcast unit and the

```
1 public void triggerEvent(int millisGone) {
2    if (trigger) {
3       trigger = false;
4       for (Rectangle2D rect : rects) {
5          new ObstacleRect(world, rect, 0);
6       }
7       rects = new ArrayList<Rectangle2D>();
8    }
9 }
10
11 public void setTrigger(float posX, float posY,
12                        float width, float height) {
13    trigger = true;
14    rects.add(new Rectangle2D.Float(posX, posY, width, height));
15 }
```

Figure 4.7: *Code of the class `UTObstacleInterruption`.*

broadcast messages that were implemented for the hazardous waste cleanup mission.

As mentioned above, the behavior sets *find-locations-methodical* and *find-locations-wander* require a special broadcast message, that enables the robots to broadcast information about the positions of the spill locations. The default implementation for broadcast messages, `StringBroadcastMessage` could be used, but it would be laborious to extract the coordinates for the specific locations. Therefore, another implementation of the `IBroadcastMessage` has been devised, the `PositionUpdateMessage`. In addition to the basic information set (which robot broadcasted it, which task is concerned and at which time the message has been broadcasted), this message stores the coordinates of the locations that were calculated by the given robot and also provides the necessary methods to retrieve these coordinates.

`PositionUpdateMessages` are also used to decide whether the behavior sets are applicable or not. The two behavior sets to find the locations are only applicable when the exact positions are unknown, i.e. when no `PositionUpdateMessage` has been broadcasted, whereas the other two behavior sets are only applicable after the locations have been found. As the specific motivational behaviors – which each decide the applicability of their associated behavior set – remember whether the locations are known, the `PositionUpdateMessage` (only one message of this kind is broadcasted in this mission) may be deleted shortly after it has been broadcasted.

A specific `WasteTeamBroadcast` also has been implemented, which extends the basic `TeamBroadcast` to provide a method that returns a list of all `PositionUpdateMessages` that were broadcasted. These two classes now enable the robots to effectively broadcast and retrieve the calculated coordinates of the relevant locations.

It should be noted, that in this implementation, the robots all share the same broadcast unit. Also, there remains one minor issue with the broadcast mechanism, concerning the point of time at which broadcast messages become visible for other team members. This will be discussed in section 4.4.

4.3 Experiments

Two experiments were conducted to show the functionality of the developed simulator and the ALLIANCE framework. The first one without any interruption, in the second one, one robot has been trapped. The following sections present these experiments and

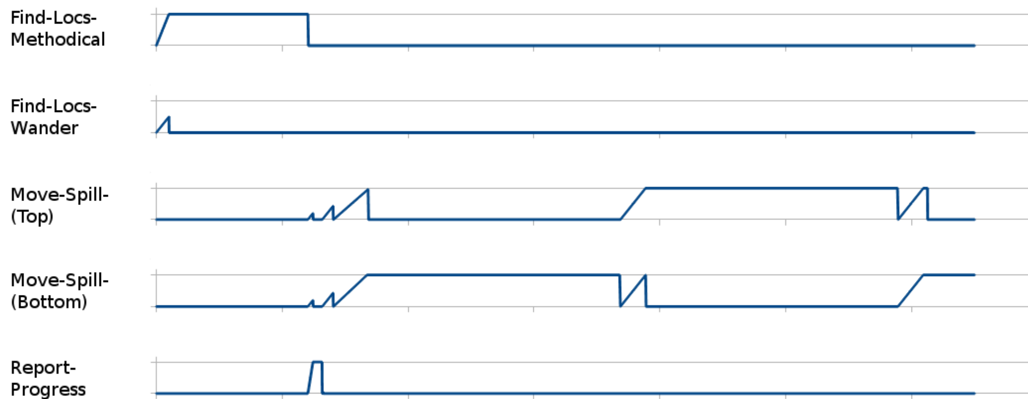

Figure 4.8: *The motivational levels of all the behavior sets in robot* blue.

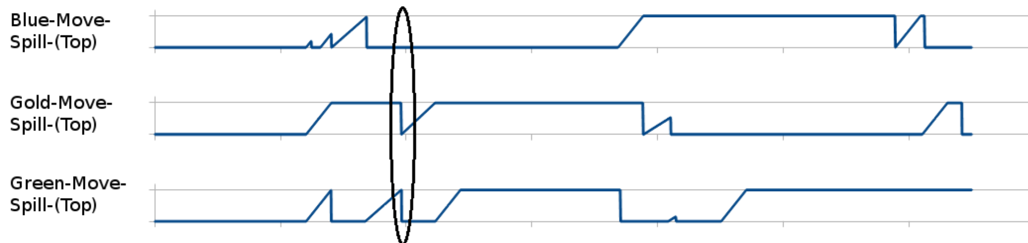

Figure 4.9: *The motivational levels of the behavior sets of all robots for moving the upper spill location.*

offer a short discussion of the results, backed by a graphical analysis of some of the log files. Note that no elaborate parameter settings for the motivational behaviors and no adaption mechanism for the parameters were used.

4.3.1 No Interruption

The first experiment features a simulation run with no interruptions. At the beginning of the mission, only the two behavior sets for finding the locations are applicable. After some time, the blue robot activates its behavior set *find-locations-methodical*. Upon discovering all walls of the border and calculating the positions of the spill locations, the blue robot broadcasts a `PositionUpdateMessage` and the robots take turns at moving the spill locations and periodically reporting the progress.

Figure 4.8 shows the motivational levels of all behavior sets of robot *blue*. It can be observed that its behavior set *find-locations-methodical* is activated shortly after the beginning of the mission, which causes the motivations of its behavior set *find-locations-wander* to be reset to 0 and to be inhibited as long as the first behavior set remains active. After broadcasting the `PositionUpdateMessage`, the applicability of the behavior sets change, as there is no need to find the locations anymore. Therefore, the motivational levels for the other three behavior sets rise and are reset, depending on the task selection of the robots.

Figure 4.9 compares the motivational levels of the behavior set *move-spill-(top)* of all robots. After the locations have been discovered, the motivational levels begin to increase for all three robots. The motivation for the blue robot gets reset three times in

the beginning. First, because the robot has activated its behavior set *report-progress*. The second time, because the gold robot has activated its behavior set *move-spill-(top)* and the third time, because the blue robot has activated its behavior set *move-spill-(bottom)*. The gold robot is the first one to activate this behavior set, but gets interrupted after a short time, although neither its time before yielding, $\psi_{ij}(t)$, nor its time before giving up, $\lambda_{ij}(t)$, has been reached. But as specified by function 2.5 in section 2.2.2, the motivation of a behavior gets reset to 0 when another robot takes over the task for the first time. In this case, the green robot activates its behavior set *move-spill-(top)* for the first time. The ellipse in figure 4.9 highlights this event.

At that point a curiosity can be observed, as the motivational level of the green robot immediately returns to 0, which means it stops the execution of the task as soon as the motivation for its behavior set *move-spill-(top)* reaches the threshold of activation. As the gold robot currently performs the according task, this behavior set actually is not activated, but set waiting. Therefore, the green robot broadcasts a communication message to indicate it wants to take over the task. In the next simulation step, this causes the gold robot to stop the execution of its behavior set *move-spill-(top)*, which can be observed in the highlighted area of figure 4.9, as its motivational level gets reset to 0. Again, the gold robot yields the task, because another robot wants to execute it for the first time. The green robot would now be able to activate the behavior set, but in the same simulation step it broadcasts the aforementioned message, it also activates its behavior set *report-progress* (which is not illustrated in figure 4.9). This obviously causes the motivation of the behavior set *move-spill-(top)* of the green robot to be reset to 0, which produces the curiosity that the green robot interrupts the execution of the gold robot, although it actually doesn't perform the task afterwards.

It can be observed, that there are occasions where the motivation of two behavior sets from different robots both reach the threshold of activation, for example shortly after the highlighted area of figure 4.9. In such cases, the behavior set that reaches the threshold later is set waiting, while the other one continues to be active until the according robot decides to acquiesce the task.

4.3.2 Trapping one Robot

In this experiment, the blue robot gets trapped while performing its behavior set *find-locations-methodical*. This is done by surrounding it with obstacles, using the `UTObstacleInterruption`. Figure 4.10 shows the trapped robot and the graphical interface of the interruption handler.

It can be seen in figure 4.11, that the other two robots get increasingly impatient. After some time, the gold robot activates its behavior set *find-locations-wander* and therefore takes over the task. This causes the motivational levels of the other two robots to be reset to 0. Afterwards they rise again, for a time period with the slow impatience rate[2] and after that with the fast impatience rate. This is highlighted in figure 4.11, where a definite bend in the motivational levels can be observed, shortly before the behavior sets of the blue and the gold robots reach the threshold for a second time.

Upon reaching the threshold of activation again, *find-locations-methodical* of the blue robot is set waiting for a short time, because it is not the first time the robot wants to execute the task and the gold robot doesn't want to acquiesce the task yet. The green robot however is permitted to take over the task, as it is the first time this robot wants to perform it. Concurrently, the motivation of the other two robots

[2]This time period is given by the parameter $\phi_{ij}(k, t)$, as defined in section 2.2.2.

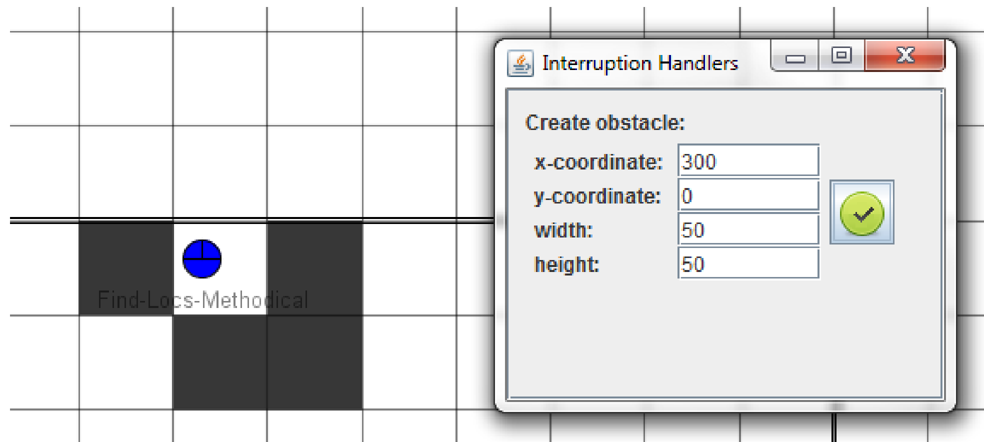

Figure 4.10: *The blue robot has been trapped at the border by surrounding it with obstacles, set with the graphic interface of the* `UTObstacleInterruption`.

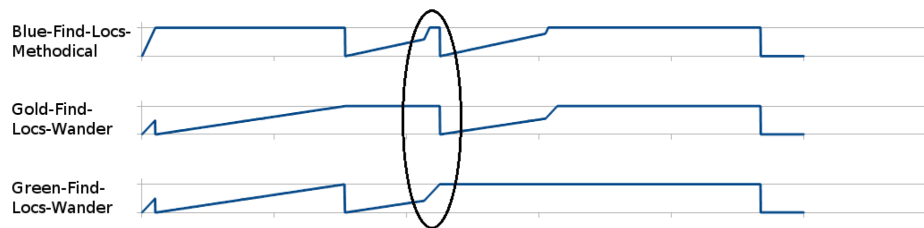

Figure 4.11: *The motivational levels of the behavior sets* find-locations-methodical *of the blue robot (which gets trapped) and* find-locations-wander *of the other two robots. This figure only shows the period of time where these behavior sets are applicable (i.e. while the locations need to be found).*

get reset to 0 and afterwards increase in the same manner as just described. They eventually both reach the threshold again, but as both the blue and the gold robot have executed the task at some point in time, they are set waiting. Therefore the green robot can finish its behavior set *find-locations-wander* without any interruptions, as it doesn't decide to acquiesce the task. After finding all locations, the green robots sends a `PositionUpdateMessage`, which in turn causes a reset of all motivational behaviors, because they are not applicable anymore.

After the green robot has completed the task *find-locations*, the robots again take turns at moving the spill locations and periodically reporting the progress. But since the blue robot is still trapped, it cannot properly execute any of these tasks and therefore decreases the overall efficiency of the robot team. It can also be observed, that the task *report-progress* is not accurately performed anymore, which can be attributed to poor parameter settings: The two non-blocked robots only consider to activate their behavior set *report-progress*, when they aren't currently executing another task. In situations where both robots (gold and green) currently move waste items, the report only can be done when one of the robots has acquiesced the task. This issue could be solved by adapting the parameters $\psi_{ij}(t)$ and $\lambda_{ij}(t)$ to the time period the progress reports should be made. This would ensure, that a robot is available at approximately the time a new progress report has to be done.

4.4 Evaluation of the Extended Simulator

Now, that experimental results of the *hazardous waste cleanup mission* are available, the developed simulator and the implemented case study can be evaluated. Since Parker conducted a laboratory version of this experiment, the results from section 4.3 can be compared to those presented in [Par98].

Section 4.3 shows obviously, that the implementation of the ALLIANCE architecture is working with the implemented case study. In the experiments, the motivation of a behavior set only increases when it is applicable and it gets reset to 0 when another behavior set in a given robot is activated, when another robot takes over the task for the first time, or when the robot acquiesces the task. This meets exactly the specifications from section 2.2.2.

With the extended functionality of the simulator to create obstacles and to change conditions of a simulation at runtime, it is also possible to create failures within the robots or for example to trap a robot, as described in section 4.3.2. In such experiments, the robots showed the ability, depending on the parameter settings, to furthermore execute the different tasks and therefore to accomplish the mission despite the given problems. However, the overall performance of the team suffers, which is no surprise, as the robot team has been reduced to two functional robots in the conducted experiments.

In comparison with the results Parker achieved, some differences can be observed. In the first experiment in section 4.3, the robots took turns at moving the spills and reporting the progress, whereas in Parkers experiment, every robot specialized on one task. Two robots continually each worked on one spill location, while the remaining robot exclusively reported the progress. This behavior could be achieved by properly adjusting the parameters. One major difference that occurred is, that in the experiments from section 4.3, the robots interrupt each other quite quickly whenever a behavior set becomes applicable for the first time. This behavior is specified by the ALLIANCE architecture in the function *impatience_reset*[3], but isn't observed in the experimental results from Parker.

As mentioned earlier, one issue remains concerning the point in time when a broadcast message is sent by a robot. A robot instantly broadcasts a message, as soon as it has a reason to do so. In combination with the stepwise simulation, this means that the robots may see different conditions in the same simulation step, which computes all actions of a fixed point in time. As described in section 3.2, the `Components` of a world get computed in the order they have been initialized. In the experiments in section 4.3, first the motivational levels of all behavior sets of the blue robot get processed, then those of the gold and lastly those of the green robot. If for example the gold robot takes over a task, it instantly broadcasts a message. The green robot now reacts to this message, whereas the blue robot – whose simulation step already has been computed – couldn't react to it.

This issue could be solved by broadcasting all the messages at the end of a simulation step, but it was chosen not to do so. The reason the current implementation has been kept is, because as a side effect, it is ensured that only one robot can activate a behavior set in a given simulation step, as the subsequent robots already received the broadcast messages sent in that step and therefore reacted to possible cross-inhibition.

[3] See [Par98] or section 2.2.2.

4.5 Summary

This chapter presented an implementation of a case study for the ALLIANCE framework. First, the case study has been defined, based on chapter 2, followed by a presentation of some case-specific implementations. Additionally, a few experiments were conducted and their results were discussed. Lastly, the simulator and the case study were evaluated.

As a scenario for the case study, the *hazardous waste cleanup mission*, conducted in a laboratory version by Parker [Par98] has been chosen: A team of three robots need to locate two spill locations and one waste deposit in a room with a border. Afterwards, the robot team has to move waste items from the spill locations to the waste deposit, while also periodically reporting the progress of the mission. This scenario was characterized on the basis of the concepts introduced in section 2.1. The environment could be identified as *partially observable*, *cooperative multiagent* and *dynamic*. Four behavior sets were defined and implemented, two to find the locations, one for moving the waste items and one for reporting the progress.

The experiments that were conducted and evaluated, demonstrated the usability of the developed simulator and the implemented case study. Simulation runs without interruptions, as well as simulation runs with the specified `InterruptionHandler`s showed desirable outcomes. However, it could also be seen that a lot of parameters need to be set for a simulation. This problem can be addressed by using an extension of the ALLIANCE architecture, which is briefly discussed in chapter 5.

With the implementation of a case study, the development of the simulator within the scope of this thesis has been completed. The next chapter gives a conclusion and presents aspects for future work.

Chapter 5

Conclusion

This thesis presented a simulator that is capable to simulate experiments featuring multiple robots in a dynamic environment. Additionally, depending on the experiment, the robotic team might have to deal with uncertainty in the sensory feedback and the task selection and task execution, as well as failures of some parts of a robot, or the complete malfunction of a team member.

The core of the presented simulator was extracted from the *Swarmulator*, developed by Martin Burger [Bur12], which makes it highly extendable for any simulation of a stepwise computed virtual world. To coordinate the robotic team, the simulator provides an implementation of the ALLIANCE architecture, which especially addresses the problems the robots experience amidst a dynamic environment and with uncertainties and failures within the robots' sensors and actuators. The implemented case study makes it possible to conduct experiments right away. These experiments may also be evaluated with the help of the detailed logfiles for each simulation run.

Although the developed simulator is fully capable to simulate experiments using the ALLIANCE framework, there is still room for improvement. Most prominently, there is the issue of the parameter setting for the motivational behaviors. Since a lot of parameters have to be adjusted, it would be preferable to have an automated way of finding a reasonable parameter setting. Parker addressed this need with the development of the so called L-ALLIANCE system, presented in [Par95], [Par97] and [Par98], which is an extension to the ALLIANCE architecture. The basic idea is, that the robots monitor the performance of other robots, measured in task execution time and therefore have a means of evaluating their own capability for the specific tasks. With this knowledge, the impatience rates and the time a robot wants to maintain its behavior set active before acquiescing it to another robot can be updated. An implementation of this extension would be very useful, as it would simplify the process of finding a proper parameter setting.

Another aspect which may be addressed in future work on the simulator is the point in time at which broadcast messages are sent, as described in section 4.4. The problem with the current implementation is, that different robots may see different conditions during the same simulation step. As stated in section 4.4, this could be solved by broadcasting all messages at the end of a simulation step, but then multiple behavior sets performing the same task may be activated in the same step, which is a problem that would have to be addressed too.

But even without the just mentioned aspects, the simulator is completely usable for experiments that utilize the ALLIANCE architecture for task allocation. As Parker stated in [Par98, p. 10], "finding the proper parameter settings in ALLIANCE has

not proved to be difficult”. This statement was affirmed in section 4.3, where a set of parameters, that wasn’t thoroughly elaborated, produced reasonable results. Additionally, although the issue with the broadcast system makes the simulations a little bit less realistic, this is not a major problem, as the current implementation ensures a flawless execution of the ALLIANCE architecture.

List of Figures

2.1 The ALLIANCE Architecture 7

3.1 Module overview of the ALLIANCE simulator 12
3.2 Simplified representation of the MVC pattern used in the Swarmulator . 13
3.3 Code of the `InterruptionHandler` interface 15
3.4 The structure of the parameter's properties file 17
3.5 Code of the `IBroadcastMessage` interface 18

4.1 The experimental mission at the beginning 22
4.2 The experimental mission after some time 23
4.3 Hazardous Waste Cleanup: Behavior Organization 24
4.4 Behavior Set: Find-Locations-Wander 25
4.5 Front sensor of the obstacle avoidance 27
4.6 Side sensor of the obstacle avoidance 28
4.7 Code-excerpt of the class `UTObstacleInterruption` 29
4.8 Motivation of Robot Blue, no Interruption 30
4.9 Motivation Move-Spill-Top, no Interruption 30
4.10 Trapping the Blue Robot 32
4.11 Motivation Find-Locations, Trapping one Robot 32

Content of the CD

The attached CD contains the following:

- This bachelor thesis in pdf format,
- Three Java Projects "AllianceCore", "AllianceTest" and "SwarmulatorCore" that contain the sourcecode of the three modules of the simulator, including the properties file for the parameters that was used for the experiments,
- A jar-file "factory.jar" that can be imported into the simulator,
- The logfiles of the conducted experiments.

Bibliography

[BPG03] Maja J Matarić Brian P. Gerkey. Multi-Robot Task Allocation: Analyzing the Complexity and Optimality of Key Architectures. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2003)*, 2003.

[BPG04] Maja J Matarić Brian P. Gerkey. A Formal Analysis and Taxonomy of Task Allocation in Multi-Robot Systems. *International Journal of Robotics Research*, 23(9):939–954, September 2004.

[Bur12] Martin Burger. Mechanisms for Task Allocation in Swarm Robotics. Diploma thesis, Ludwig-Maximilians-Universität München, 2012.

[CHF04] Khiang Wee Lim Cheng-Heng Fua, Shuzhi Sam Ge. BOAs: BackOff Adaptive scheme for Task Allocation with Fault Tolerance and Uncertainty Management. In *Proceedings of the 2004 IEEE International Symposium on Intelligent Control*, 2004.

[Par95] Lynne E. Parker. L-ALLIANCE: A Mechanism for Adaptive Action Selection in Heterogeneous Multi-Robot Teams. Technical report, 1995.

[Par97] Lynne E. Parker. L-ALLIANCE: Task-Oriented Multi-Robot Learning In Behavior-Based Systems. In *Advanced Robotics, Special Issue on Selected Papers from IROS'96*, pages 305–322, 1997.

[Par98] Lynne E. Parker. ALLIANCE: An Architecture for Fault Tolerant Multi-Robot Cooperation. *IEEE Transactions on Robotics and Automation*, 14(2):220–240, April 1998.

[SR10] Peter Norvig Stuart Russell. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2010.