Université Libre de Bruxelles
Faculté des Sciences Appliquées

Année académique 2007-2008

Fault Detection in Autonomous Robots

Endogenous fault detection through fault injection and learning - exogenous fault detection based on firefly-inspired synchronization

Anders Lyhne Christensen

Directeur de Thèse: Prof. Marco Dorigo

Thèse présentée en vue de l'obtention du
titre de Docteur en Sciences de l'Ingénieur

Statement

This dissertation has been submitted in partial fulfilment of the requirements for an advanced degree at Université Libre de Bruxelles. The dissertation describes an original research carried out by the author. It has not been previously submitted to the Université Libre de Bruxelles or to any other university for the award of any degree. Nevertheless, some chapters of this dissertation are partially based on articles that, during his doctoral studies, the author, together with a number of co-workers, submitted for publication in the scientific literature. Details can be found in Section 1.2.

Brief quotations from this dissertation are allowed without special permission, provided that accurate acknowledgement of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in part or in whole may be granted by the copyright holder.

Abstract

In this dissertation, we study two new approaches to fault detection for autonomous robots. The first approach involves the synthesis of software components that give a robot the capacity to detect faults which occur in itself. Our hypothesis is that hardware faults change the flow of sensory data and the actions performed by the control program. By detecting these changes, the presence of faults can be inferred. In order to test our hypothesis, we collect data in three different tasks performed by real robots. During a number of training runs, we record sensory data from the robots both while they are operating normally and after a fault has been injected. We use back-propagation neural networks to synthesize fault detection components based on the data collected in the training runs. We evaluate the performance of the trained fault detectors in terms of the number of false positives and the time it takes to detect a fault. The results show that good fault detectors can be obtained. We extend the set of possible faults and go on to show that a single fault detector can be trained to detect several faults in both a robot's sensors and actuators. We show that fault detectors can be synthesized that are robust to variations in the task. Finally, we show how a fault detector can be trained to allow one robot to detect faults that occur in another robot.

The second approach involves the use of firefly-inspired synchronization to allow the presence of faulty robots to be determined by other non-faulty robots in a swarm robotic system. We take inspiration from the synchronized flashing behavior observed in some species of fireflies. Each robot flashes by lighting up its on-board red LEDs and neighboring robots are driven to flash in synchrony. The robots always interpret the absence of flashing by a particular robot as an indication that the robot has a fault. A faulty robot can stop flashing periodically for one of two reasons. The fault itself can render the robot unable to flash periodically. Alternatively, the faulty robot might be able to detect the fault itself using endogenous fault detection and decide to stop flashing. Thus, catastrophic faults in a robot can be directly detected by its peers, while the presence of less serious faults can be detected by the faulty robot itself, and actively communicated to neighboring robots. We explore the performance of the proposed algorithm both on a real world swarm robotic system and in simulation. We show that failed robots are detected correctly and in a timely manner, and we show that a system composed of robots with simulated self-repair capabilities can survive relatively high failure rates.

We conclude that i) fault injection and learning can give robots the capacity to detect faults that occur in themselves, and that ii) firefly-inspired synchronization can enable robots in a swarm robotic system to detect and communicate faults.

Acknowledgements

I first visited IRIDIA on a grey and windy Monday morning in late-September 2004. The place was quite a sight. It was 8.45 a.m. and nobody was around. There was construction material everywhere and the wind was coming in through cracks in the plastic that sparsely covered the window frames where glass used to be. I was unsure if I had come to the right place – and even if I had, I was seriously considering getting out of there as fast as I could. But then a great mixture of people started dropping in and before I knew it, I had given a presentation, met the "IRIDIA gang" and seen the robots. IRIDIA was an extremely inspiring place: I met people who were working on all sorts of projects such as ant-inspired algorithms for route planning, the modeling of biological networks, computational chemistry and swarm robotics. The place had a special buzz and I was sold. When I was about to leave after less than 24 hours in Bruxelles, Prof. Marco Dorigo offered me a Ph.D. position. "When should I start?" - "Next week!". And so I did.

Now, three and a half years later, I look back at my time at IRIDIA with a lot of fond memories: it was fun, challenging, interesting and full of experiences. I would first of all like to deeply thank Prof. Marco Dorigo for his supervision and for finding the funding that allowed me to complete my doctoral studies. I would also like to thank the people from IRIDIA who made my stay at ULB so enjoyable: Alexandre Campo, Bruno Marchal, Carlo Pinciroli, Carlotta Piscolo, Christophe Philemotte, Elio Tuci, Federico Vincentini, Giovanni Pini, Hughes Bersini, Javier Martinez, Jodelson Sabino, Krzysztof Socha, Marcello Cirillo, Marco Montes de Oca, Mauro Birattari, Max Manfrinn, Muriel Decreton, Navneet Bhalla, Paola Pellegrini, Prasanna Balaprakash, Roderich Gross, Shervin Nouyan, Thijs Urlings, Thomas Halva Labella, Thomas Stuetzle, Tom Lenaerts, Utku Salihoglu, Vito Trianni, and Yann-Aël Le Borgne. Special thanks go to Christos Ampatzis, Francisco Santos, and Rehan O'Grady - I have had numerous interesting discussions with you guys, and without our collaboration, this dissertation would still have been far from completion.

I thank all the people from the *swarm-bots* project for their hard work and especially Francesco Mondada and his group at the EPLF for designing and building the robots without which the work presented in this dissertation would not have been possible.

I would like to thank Prof. Francisco Cercas and the people at Departamento de Ciências e Tecnologias da Informação, ISCTE, University of Lisbon, who have given me a warm welcome and who have been kind enough to give me time to finish this dissertation.

Andreia, thank you so much for your love and for your ever optimistic and playful nature!

Mona, your friendship is extraordinary - it really means a lot to me. You showed me a different side of Brussels and I wish you and your soon-expanding family all the best!

I would like to acknowledge support from COMP2SYS, a Marie Curie Early Stage Research Training Site funded by the European Community's Sixth Framework Program (grant MEST-CT-2004-505079). The information provided in this dissertation is the sole responsibility of the author and does not reflect the European Commission's opinion. The European Commission is not responsible for any use that might be made of data appearing in this dissertation.

To my family.

Contents

1 Introduction 1

1.1 Problem Statement . . . 3

1.2 Thesis Structure and Contribution of Research . . . 3

1.3 Other Scientific Contributions . . . 5

1.3.1 Self-Assembly, Morphology Control and Self-Reconfiguration . . . 5

1.3.2 Evolutionary Robotics . . . 8

1.3.3 Simulation Tools . . . 10

1.4 Summary . . . 14

2 Related Work 15

2.1 Endogenous Fault Detection . . . 16

2.2 Exogenous Fault Detection . . . 18

3 Robotic Hardware 21

3.1 The *swarm-bot* Platform . . . 21

3.1.1 The *s-bot* Camera and Image Processing . . . 24

3.1.2 Examples of Studies Conducted with *S-bots* . . . 25

3.2 Other Multi-Robot and Modular Robotic Systems . . . 26

3.3 Summary . . . 28

4 Fault Detection based on Fault Injection and Learning 29

4.1 Methodology . . . 30

4.1.1 Formal Definitions . . . 33

4.1.2 Faults . . . 35

4.1.3 Software Architecture . . . 36

4.2 The Three Experimental Setups . . . 36

4.3 Data Collection, Training and Performance Evaluation . . . 38

4.3.1 Data Collection . . . 38

4.3.2 Training and Evaluation Data 38
4.3.3 Performance Evaluation 38
4.4 Results 40
4.4.1 Tuning the Input Group Distance 40
4.4.2 Fault Detection Performance in the *follow the leader* and *connect to s-bot* setups 45
4.4.3 Reducing the Number of False Positives 46
4.4.4 Faults in Both Sensors and Actuators 49
4.4.5 Robustness to Variations in the Task 51
4.4.6 Exogenous Fault Detection in a Cooperative Task 52
4.5 Extensions and Limitations 55
4.6 Summary 59

5 Fault Detection in Swarms of Robots **61**
5.1 Motivation 62
5.2 Synchronization in Natural and Artificial Systems 65
5.3 Synchronization in Robots 67
5.3.1 Discrete Oscillators 67
5.3.2 Synchronization Experiments in Simulation 68
5.3.3 Synchronization Experiments with Real Robots 72
5.4 Fault Detection in Swarms of Robots 73
5.4.1 Detecting Faults in Non-Synchronized Robots 73
5.4.2 Time Overhead 75
5.4.3 Implementation 76
5.4.4 Fault Detection Experiments with Real Robots 77
5.4.5 Fault Tolerance Experiments with Real Robots 78
5.4.6 Limitations of the Approach 79
5.4.7 Limitations of the Current Implementation of the Approach 79
5.5 Summary and Directions for Future Work 80

6 Summary and Future Work **83**
6.1 Summary of Contributions 83
6.2 Challenges for the Future 84

7 Conclusions **89**

A Appendix **91**

A.1 Software Architecture for Fault Detection based on Fault Injection and Learning 91

A.2 Summary 98

List of Figures **99**

List of Tables **103**

References **105**

Introduction

Chapter 1

Imagine a future in which we are surrounded by robots: Robots to clean, robots to garden, robots to cook, robots to take out the trash, and so on. From the beginning of civilization, we have been preoccupied with constructing tools and modifying the environment around us to make our lives easier, better, more efficient, and so on. The construction of intelligent machines to perform more sophisticated tasks including domestic chores is the next step on this quest. We do, however, face a number of challenges that must be overcome before we can sit back, relax and let the robots do the rest.

As many have pointed out [Pollack, 1981, Kurzweil, 2005, Butterfield, 2006, Gates, 2006], the field of autonomous robotics is in its infancy and in many ways resembles the field of computers 30 to 40 years ago. In the 1970s, every computer manufacturer had to custom-design and custom-build everything including the boards to hold the electronics and software such as the operating system (if any). As a result, the computers were expensive, they were not very reliable and they could only perform basic tasks compared to today's standards. Currently, the field of robotics is in a similar phase: Robot manufacturers usually have to design a significant amount of new hardware and develop a custom software stack for every new robot that they produce. Although there have been recent efforts to standardize interfaces and control software for autonomous robots, there is still no de facto method for object identification, map construction, speech recognition, and so on. As a result, robots are expensive, they break easily, and they are often only of interest to researchers and hobbyists.

Over the past four decades, we have witnessed how hardware and software components for personal computers have been standardized. As a result, we have much cheaper computers that can perform a much broader range of tasks. Arguably, computers have also become more reliable despite the many-fold increase in complexity that hardware and software have undergone. Still, few of us would like to let our lives depend on our personal computer. And the upside is that we don't really have to: in case we accidentally spill a cup of coffee over our keyboard in an absent moment or if our harddisk crashes, it may feel like the end of the world, but after all, we (usually) survive. For robots the same may not hold true. If a robot stops working in the way it is supposed to, the consequences could be dire and easily go beyond a lost document and the cost of a new harddisk: because robots operate in the physical world, an undetected fault in a domestic service or leisure robot could result in human injury and/or material damage. Imagine, for instance, that a robot experiences a fault while ironing clothes and that the fault prevents the robot from moving a joint - the situation could result in a fire.

Mobile robots used in labs today have been found to be rather unreliable: in a recent paper [Carlson et al., 2004], the reliability of fifteen mobile robots from three different manufacturers was tracked over a period of three years and the average mean time between failures was found to be 24 hours. The result suggests that faults in mobile robots are quite frequent. As we encounter more and more robots in our daily lives, and as more businesses and government organizations start to rely on robots, it becomes increasingly important to make intelligent machines dependable and safe. It can, however, be both difficult and costly to make a system detect faults and respond correctly and safely to their presence. If we consider mechanical systems such as cars, air planes, washing machines and robots, we find that fault tolerance has only been implemented in places where it has been absolutely necessary. An unnoticed leak of oil from a hydraulic pump controlling the flaps of a commercial airliner can have fatal consequences, while a similar leak in a washing machine merely causes a wet floor and ruins the laundry. The main reason that fault detection and identification is not a common built-in feature in all systems is that it often requires special software and hardware to monitor the system. This prolongs the development time and increases complexity and cost.

In this thesis, we study new ways of detecting faults in autonomous robots. Discovering that a fault has occurred is the first step in the process of ensuring that a robot remains safe and dependable even if something stops functioning correctly. We first study a method that enables a robot to discover that a fault, such as a broken wheel, has occurred in itself. There are various ways of detecting such faults: we could for instance add sensors for proprioception such as encoders. If an encoder detects that a wheel is not turning when it should, we could interpret that as a symptom of a fault. Alternatively, we could build a model of how the robot is supposed to behave and compare the actual behavior to the behavior predicted by the model. We discuss some of the established techniques in more detail in Chapter 4. We present an alternative method for detecting faults, namely through fault injection and learning. We collect data while a robot is operating normally and after faults have been injected. Based on the collected data, we train the robot to detect the presence of faults. Our method has the advantage that no additional sensors are needed and we do not need to build an analytical model of how a robot should behave (which is a non-trivial task in many cases).

Some faults are, however, hard to detect in the robot in which they occur. These faults include software bugs that cause the on-board software to hang, sensor failures that prevent a robot from detecting that something is wrong and mechanical faults such as an unstable connection to a power source. Alternatively, a robot might be able to detect a fault, but the fault itself might still render the robot unable to alert a human operator or another robot. When multiple robots are present or even working together on some task, it can therefore be advantageous to give robots the capacity to detect faults in each other. Based on this premise, we show how one robot can learn to detect faults that occur in another robot: we record training data from one robot while the other robot is operating normally and while it is subject to a fault. There appears, however, to be no way to scale this approach to larger groups or swarms of robots. We discuss this issue and we go on to propose a different method for detecting non-operational robots in swarms of robots. Our method enables robots in a multi-robot system to detect faults in one another: operational robots emit periodic flashes of light that nearby robots can detect. Non-operational robots do not emit flashes periodically and they can thus be detected. Inspired by the behavior observed in some species of fireflies, the robots are able to synchronize their periodic flashing. When the robots are synchronized, it is straightforward to detect non-operational robots: whenever a synchronized robot flashes it can detect faults by looking for non-flashing robots.

1.1 Problem Statement

We study the activity known as *fault detection* for autonomous robots. Fault detection is a binary decision process confirming whether or not a fault has occurred in a system. A fault is an unexpected change in system function which hampers or disturbs normal operation, causing unacceptable deterioration in performance [Isermann and Ballé, 1997]. A *fault tolerant* system is capable of continued operation, possibly at a degraded performance, in the event of faults in some of its parts. Fault tolerance is a sought-after property for critical systems due to economic and/or safety concerns. In most systems, the capability to detect faults is a prerequisite for ensuring that proper action is taken when faults occur. Other aspects of fault tolerance include *fault identification*, namely determining the type and location of faults, and *fault accommodation* which comprises any steps necessary to ensure continued safe operation of the system.

In this thesis, we focus on fault detection: we first tackle the problem of how to give robots the capacity to detect *endogenous faults*, that is, the capacity of a robot to detect faults in itself. Not all faults can be detected in the robot in which they occur: catastrophic faults, such as complete failure, render a robot incapable of detecting faults and/or taking any deliberate action. We therefore also study how robots in a multi-robot system can detect *exogenous faults* – that is, how robots can detect faults that occur in one another.

1.2 Thesis Structure and Contribution of Research

In this section, we provide an overview of the thesis structure and the scientific publications produced over the past three and a half years leading to this thesis.

In Chapter 2, we review the history and the state-of-the-art of fault detection and fault tolerance in robotics. The chapter is divided into two sections: one dedicated to endogenous fault detection and one dedicated to exogenous fault detection.

In Chapter 3, we go on to present the *swarm-bot* robotic platform that we have used for the work presented in this thesis. The platform consists of a number of autonomous mobile robots that have the ability to physically connect to each other to form larger robotic entities. We review the hardware and present some of the studies conducted on the *swarm-bot* platform.

In Chapter 4, we focus on automatic synthesis of fault detection modules. More specifically, we propose a methodology based on fault injection and learning for obtaining fault detection modules. We let one or more robots perform a given task while we record the sensory data and the control signals sent to the actuators of the robot(s). During the recording phase, we inject simulated hardware faults in the robots' actuators. In this way, we obtain a set of recorded sensor readings and actuator control signals corresponding both to when a robot is operating normally and to when faults are present. We train neural networks to detect faults based on the data collected. We first introduce the methodology (see Section 4.1). We then present the three tasks in which we test the performance of the approach (see Section 4.2). We go on to present how training data is collected and how the performance of a fault detector can be measured (see Section 4.3). In Section 4.4, we present the results of experiments with real robots. The results show that the proposed method enables the accurate and timely detection of faults. This work was published in:

- A. L. Christensen, R. O'Grady, M. Birattari, and M. Dorigo, "**Automatic Synthesis of Fault Detection Modules for Mobile Robots**", *Proceedings of the NASA/ESA conference on Adaptive Hardware and Systems (AHS-2007)*, IEEE Computer Society, Los Alamitos, CA, pages 693–700, 2007

We then go on to show that the method is applicable when faults in both sensors and actuators are considered (see Section 4.4.4). Autonomous mobile robots often navigate in environments in which the conditions and task parameters are unknown and sometimes change over time. A fault detection approach has to be robust to such changes in order to be generally applicable. In Section 4.4.5, we show that faults can be correctly and timely detected in a task that varies from trial to trial.

One of the three tasks for which we choose to evaluate fault detection based on fault injection and learning is a leader and follower navigation task. Two robots move around in an arena enclosed by walls. One robot is assigned the leader role while the other robot is assigned the follower role. The leader performs a random walk in the environment while the follower simply follows the leader. In a set of additional experiments, we record sensory data and actuator control signals from the leader robot while we inject faults in the follower. Based on the data collected, we train a neural network that gives the leader the capacity to detect faults in the follower robot. We show that the leader is able to detect faults that occur in the follower, that is, exogenous faults. This work was presented in:

- A. L. Christensen and R. O'Grady and M. Birattari and M. Dorigo, "**Exogenous Fault Detection in a Collective Robotic Task**", *Proceedings of the 9th European Conference on Artificial Life (ECAL2007)*, Springer Verlag, Berlin, Germany, pages 555–564, 2007

The studies related to fault detection through fault injection and learning were extended with experiments involving more complex setups such as varying environmental conditions and a larger set of faults. The result was a publication in the journal *Autonomous Robots*:

- A. L. Christensen, Rehan O'Grady, Mauro Birattari and Marco Dorigo, "**Fault Detection in Autonomous Robots Based on Fault Injection and Learning**", *Autonomous Robots*, 24(1), pages 49–67, 2008

In Section 4.5, we conclude the chapter with a discussion of the limitations of fault detection based on fault injection and learning. We discuss how some of these limitations potentially could be overcome and how the methodology could be extended to include fault identification. We argue that there appears to be no general way of synthesizing modules for exogenous fault detection using our methodology when larger groups or swarms of robots are considered.

In Chapter 5, we propose a method that gives the constituent robots in a multi-robot system the capacity to detect faults that occur in one another (exogenous faults). We exploit some of the high-level principles underlying synchronizing systems found in Nature to obtain a robust, simple, distributed approach to fault detection in groups or swarms of robots. Through local interactions, a group of robots is able to synchronize and reach a state in which they flash periodically in unison. When a robot breaks down, it ceases to flash. By detecting the absence of flashes, operational robots can effectively detect failed robots. We first motivate our approach and discuss alternatives (see Section 5.1). We then discuss synchronization among pulse-coupled oscillators in natural and artificial systems (see Section 5.2). We explore different parameter settings of a model adapted to our robots and present synchronization results obtained in simulation (see Section 5.3.2). We go on to demonstrate synchronization in a group of 10 real robots (see Section 5.3.3). In Section 5.4, we discuss how faults can be detected by detecting non-synchronized (non-flashing) robots and we test the approach on real robots. In one experiment, we give the robots the ability to simulate repair of one another. We show that a group of robots with these capabilities can detect faults and survive a relatively high rate of failure.

The work on synchronization and fault detection presented in Chapter 5 has been submitted for publication at ICRA and the IEEE Transactions on Evolutionary Computation:

- A. L. Christensen, R. O'Grady and M. Dorigo, "**Synchronization and Fault Detection in Autonomous Robots**", *Submitted to the IEEE/RSJ 2008 International Conference on Intelligent Robots and Systems*, under review
- A. L. Christensen, R. O'Grady and M. Dorigo, "**From Fireflies to Fault Tolerant Swarms of Robots**", *Submitted to IEEE Transactions on Evolutionary Computation*, under review

In Chapter 7, we conclude and discuss directions for future work.

All of the publications discussed in this chapter and the work presented in this thesis are based either partly or completely on experiments with real robotic hardware.

1.3 Other Scientific Contributions

We have conducted a number of other studies that are not directly related to the topic of this thesis. These studies fall into two categories: One that is concerned with *self-assembly*, *morphology control*, and *self-reconfiguration*, while the other is concerned with a controller design methodology called *evolutionary robotics*, which relies on artificial evolution. In the following two sections, we present in brief our scientific contributions not directly related to the main topic of this thesis.

1.3.1 Self-Assembly, Morphology Control and Self-Reconfiguration

The *swarm-bot* robotic platform used in this thesis belongs to a class of multi-robot systems in which the individual units can physically connect to one another and form larger structures (see Chapter 3 for details on the robotic hardware). Self-assembly is a mechanism that allows teams of cooperating robots to overcome the physical limitations of the individual team members. Figure 1.1 shows two examples of physically connected robots carrying out tasks impossible for a single robot: on the left, a group of robots is crossing a trough impassable by a single robot and on the right, a group of robots is navigating rough terrain on which a single robot would topple over. Self-assembly is challenging because it requires numerous autonomous robots with limited sensory capabilities to coordinate their actions. Decentralized control is usually favored in multi-robot systems for the benefits it confers of scalability, robustness and flexibility [Cao et al., 1997, Bonabeau et al., 1999, Shen et al., 2004]. However, distributed control renders the problem of coordination more difficult, as the individual agents usually have only a partial view of the system and have to act based on local information alone.

We first demonstrated that self-assembly can increase the task-execution performance of a real multi-robot system in the following publications:

- R. O'Grady, R. Gross, A. L. Christensen, F. Mondada, M. Bonani and M. Dorigo, "**Performance Benefits of Self-Assembly in a Swarm-Bot**", *Proceedings of the 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '07)*, IEEE Computer Society, Los Alamitos, CA, pages 716–725, 2007
- R. O'Grady, R. Gross, A. L. Christensen, and M. Dorigo, "**Self-Assembly Strategies in a Group of Autonomous Mobile Robots**", *Submitted to Autonomous Robots*, under review

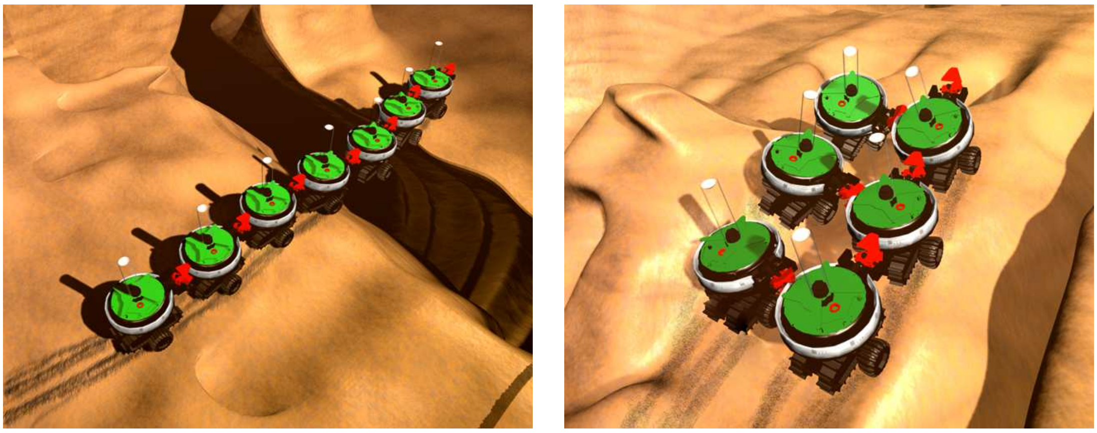


Figure 1.1: Two examples of robotic entities self-assembled into morphologies appropriate for the task. Left: A connected robotic entity crosses a trough. A line formation is well-suited to this task, since it allows the entity to stretch further and requires only a minimum number of robots to be suspended over the trough at any one time. Right: A more dense structure provides greater stability for rough terrain navigation.

The examples in Figure 1.1 highlight the importance of the morphology of the self-assembled robotic entities. The elongated structure (left) allows the robots to reach across the trough while the dense structure (right) provides stability on rough terrain.

We developed a novel *directional self-assembly* mechanism. This mechanism allows the robots to specify the location and orientation of the connections made during the self-assembly process. We built a set of local pattern extension rules on top of the directional self-assembly mechanism. In the following publications, we demonstrated self-organized growth of specific morphologies on a real-world self-assembling multi-robot system:

- A. L. Christensen, R. O'Grady and M. Dorigo, "**A Mechanism to Self-Assemble Patterns with Autonomous Robots**", *Proceedings of the 9th European Conference on Artificial Life (ECAL2007)*, Springer Verlag, Berlin, Germany, pages 716–725, 2007
- A. L. Christensen, Rehan O'Grady and Marco Dorigo, "**Morphology Control in a Self-Assembling Multi-Robot System**", *IEEE Robotics & Automation Magazine*, 14(4), pages 18–25, 2007
- R. O'Grady, A. L. Christensen and M. Dorigo, "**Self-Assembly and Morphology Control in a Swarm-Bot**", *Proceedings of the 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE Computer Society, Los Alamitos, CA, pages 2551-2552, 2007
- A. L. Christensen, R. O'Grady, and Marco Dorigo, "**Morphogenesis: Shaping Swarms of Intelligent Robots**, *AAAI-07 Video Proceedings*, 2007, **Best Video Award**
- R. O'Grady, A. L. Christensen, and M. Dorigo, "**SWARMORPH: Morphology Control with a Swarm of Self-Assembling Robots**". *Extended abstract accepted for the Workshop on Self-Reconfigurable Robots/Systems and Applications, held as part of 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems, San Diego, CA*, 2007, unpublished
- R. O'Grady, A. L. Christensen, and M. Dorigo, "**SWARMORPH: Multi-Robot Morphogenesis Using Directional Self-Assembly**", *Submitted to IEEE Transactions on Robotics*, under review

We extended the work listed above and showed how arbitrary morphologies can be formed in a way that could enable the adaptive use of the morphologies in practical task-execution scenarios. We showed how morphology size can be regulated. We demonstrated how multiple identical morphologies can be assembled. Finally, we showed how robots with no a priori knowledge of a task can form morphologies based on instructions from robots already engaged in task-execution:

- A. L. Christensen, R. O'Grady, and M. Dorigo, "**SWARMORPH-script: A Language for Arbitrary Morphology Generation in Self-Assembling Robots**". *Swarm Intelligence*, accepted for publication

In another study, we showed for the first time how real robots capable of self-assembly can autonomously self-reconfigure between different morphologies:

- R. O'Grady, A. L. Christensen, and M. Dorigo, "**Autonomous Reconfiguration in a Self-Assembling Multi-Robot System**", *Submitted to the Sixth International Conference on Ant Colony Optimization and Swarm Intelligence (ANTS 2008)*, accepted for publication

Our ongoing research concerns leveraging the morphology generation approach to add functional value to a group of robots. More specifically, we want to give the robots the capability to identify different types of obstacles, assemble into appropriate morphologies and then to overcome the obstacles. In an all-terrain navigation task, for example, the group could self-assemble into a line morphology in order to cross a ditch, while uneven or hilly terrain could trigger self-reconfiguration into a dense morphology that provides stability (see the examples in Figure 1.1).

1.3.2 Evolutionary Robotics

The field in which evolutionary techniques are applied in order to design robotics hardware and/or control software is called *evolutionary robotics* [Nolfi and Floreano, 2000]. One direction of studies in this field is concerned with cognitive science and psychology [Harvey et al., 2005], while another direction focuses on the use of evolutionary techniques as an engineering tool. Our interest falls in the latter category. A robotics setup where artificial evolution can be applied usually starts off with one or more robots and a task that we want the robot(s) to solve. A fitness function is defined, which, given a behavior, assigns a score reflecting the goodness of that behavior with respect to the task. An evolutionary algorithm is then used to search for a good controller. The controllers themselves may consist of rule sets, decision trees or similar, but it has become common to use artificial neural networks (ANNs) due to their versatility and tolerance to noisy sensory input. If the controller is represented as an ANN, an evolutionary algorithm can be applied in order to optimize the weights, and possibly the morphology, of the network. Solutions found in this way can exploit subtle environmental features as they are perceived through the robot's sensors. Therefore, artificial evolution might not only be a time-saving approach for synthesizing controllers: better controllers than those hand-crafted by human developers can be obtained in some cases [Nolfi and Floreano, 2000].

Although promising, evolutionary methods have to our knowledge only been successfully applied to relatively simple tasks and are not yet used extensively in industry as a tool for automatic controller design. This is most likely explained by the fact that reaching the point where artificial evolution produces a controller that solves a given task is a difficult, tedious and time-consuming process, which involves a large amount of trial-and-error. First of all, it can be difficult to define a fitness function for the task that we wish our robots to perform. Secondly, the fitness function has to assign scores in such a way that gradients in the fitness landscape push evolution towards good solutions. Thirdly, it can be difficult to start the evolutionary machinery, that is, to initially find regions of the fitness landscape with gradients that lead the search towards better solutions (this is also know as the *bootstrapping problem*). Fourthly, determining a good type, size and morphology of a neural network (unless under evolutionary control) often have to be done on a trial-and-error basis. If evolutionary robotics is to be used in more complex scenarios, we need to improve our understanding and methods for designing suitable evolutionary setups. Based on this premise, we suggested a structured method for applying evolutionary robotics methods to synthesize robot controllers. We chose a collective navigation task in which multiple physically connect robots should safely navigate an arena containing holes. This work was published in:

- A. L. Christensen and M. Dorigo, "**Evolving an Integrated Phototaxis and Hole-Avoidance Behavior for a Swarm-bot**", *Artificial Life X: Proceedings of the Tenth International Conference on the Simulation and Synthesis of Living Systems*, MIT Press, Cambridge, MA, pages 248–254, 2006

In a continuation of our work, we applied two different types of *incremental evolution* to the same task. Incremental evolution is a method in which evolution begins with a population that has already been trained for a simpler, but in some way related task [Harvey et al., 1994]. This is done by changing the fitness function and/or the environment during evolution in order to make the task progressively more complex. In this way, bootstrapping problems can possibly be overcome and evolution can be sped up. The use of incremental evolution can, however, require a substantial engineering effort, because the goal-task has to be organized into a number of sub-tasks of increasing complexity. Our research concerning incremental evolution was published in:

- A. L. Christensen and M. Dorigo, "**Incremental Evolution of Robot Controllers for a Highly Integrated Task**", *From Animals to Animats 9: 9th International Conference on Simulation of Adaptive Behavior, SAB 2006*, Springer Verlag, Berlin, Germany, pages 473–484, 2006

In a social scenario, establishing whether collaboration is required to achieve a certain goal is a complex problem that requires decision making capabilities and coordination among the members of the group. Depending on the environmental contingencies, solitary actions may result more efficient than collective ones and vice versa. If each robot in a group has only limited knowledge about the environment, estimating the opportunity to persevere individually or to engage in collaboration may be very difficult or expensive. We used artificial evolution to synthesize neural controllers that let robots decide when to switch from solitary to collective actions based on the information gathered through time. This work was presented in:

- V. Trianni, C. Ampatzis, A. L. Christensen, E. Tuci, M. Dorigo and Stefano Nolfi, "**From Solitary to Collective Behaviours: Decision Making and Cooperation**", *Proceedings of the 9th European Conference on Artificial Life (ECAL2007)*, Springer Verlag, Berlin, Germany, pages 575–584, 2007

Recently, we have managed to evolve homogeneous controllers that let real robots self-assemble. We demonstrated how robots are able to allocate roles between them without the use of explicit signals. This work has been submitted for publication in:

- E. Tuci, C. Ampatzis, V. Trianni, A. L. Christensen, and M. Dorigo, (2008), "**Self-Assembly in Physical Autonomous Robots: the Evolutionary Robotics Approach**". *Submitted to Artificial Life XI, the 11th Conference on the Simulation and Synthesis of Living Systems*, accepted for publication
- C. Ampatzis, E. Tuci, V. Trianni, A. L. Christensen and M. Dorigo, "Evolving Autonomous Self-Assembly in Homogeneous Robots". *Submitted to the Artificial Life Journal*, under review

All of the studies listed in this chapter were carried out with the aid of a robot simulation tool that we developed. We briefly discuss the motivation behind the simulator and its main design philosophies in the next section.

1.3.3 Simulation Tools

The significance of suitable simulation tools is seldom emphasized in scientific publications on robotics. The robots themselves tend to steal the spot light. However, the majority of the software developed in many robotics projects (including the *swarm-bots* project[1]) is not for execution on real robots, but rather for execution on workstations. Simulators allow researchers to develop and test controllers on their own workstations before the controller is executed on real robotic hardware. In simulation, ideas can easily be tested. Development can start before the hardware is available. A large number of experiments can be conducted in simulation with little or no manual intervention, and so on. We have developed our own simulator called *TwoDee* which started out as a fast, specialized simulator for evolutionary robotics. The simulator itself grew over time to include *the Common Interface*, which is essentially an implementation of the robot API in TwoDee. The Common Interface allows hand-coded control programs to be run both in simulation and on the real robots without any change to the source code. TwoDee has formed the basis of simulator for the *e-puck* robotic platform and parts of TwoDee have been included in the simulator currently being developed as part of the *swarmanoid* project[2].

The TwoDee Simulator

TwoDee is a fast, specialized multi-robot simulator for the *swarm-bot* robotic platform. TwoDee has a custom rigid body physics engine, specialized to simulate only the dynamics in environments containing flat terrain, walls and holes. This restriction allows for certain optimizations in the computation of the physics and thereby reduces the computational resources necessary for running simulations significantly (see [Christensen, 2005] for more details on TwoDee). TwoDee is written in C++. It compiles and runs on POSIX.1 compatible operating systems such as

---

[1] The *swarm-bots* project was sponsored by the Future and Emerging Technologies program of the European Commission. The project aimed to study new approaches to the design and implementation of self-organizing and self-assembling artifacts. This novel approach found its theoretical roots in recent studies in swarm intelligence, that is, in studies of the self-organizing and self-assembling capabilities shown by social insects and other animal societies. For more information see http://www.swarm-bots.org.

[2] The *swarmanoid* project is a Future and Emerging Technologies project funded by the European Commission. The main scientific objective of this research project is the design, implementation and control of a novel distributed robotic system. The system will be made up of heterogeneous, dynamically connected, small autonomous robots. For more information see www.swarmanoid.org.

Linux and under CygWin in Windows. With relatively few change it should be compilable and able to run natively on other operating systems such as Microsoft Windows. Autoconf and automake from the GNU Autotools project are used for dependency checking and build management. At the time of writing, TwoDee comprises approximately 85.000 lines of source code.

The Common Interface and Hand-Coded Controllers

Experimentation on real robotic hardware is often a time-consuming and tedious process. The robots to which we had access were specially designed and built for the *swarm-bots* project. The robots suffered from a number of nuisances that can be partly attributed to the prototypic nature of the hardware. Before each experiment, every robot had to be tested and re-calibrated and a robot would often break down during experiments. Hence, running experiments on real robots is time consuming.

Before TwoDee was developed, control programs would either be developed directly on the real robots or they would first be developed in an initial version for some particular simulator. Code is very rarely correct the first time it is executed and usually significant amount of testing and debugging is required in order to locate and fix design errors and bugs. Testing and debugging directly on the real robots is slow and tedious: each new version of the control program needs to be compiled for, transferred to and tested on one or more real robots. When the control program is run, the behavior of the system has to be observed in real-time. The debugging facilities on the real robots are minimal and it is hard to locate the source of an error. Often, encountered issues are difficult to replicate. Furthermore, it can sometimes be hard to determine if the problem is due to a bug in the control program code, an error in the design of the control algorithm, or if the incorrect behavior is due to a hardware hick-up.

When an initial version of the control program would be developed in simulation first, it would have to use the particular interface and representations for simulator of choice. When a satisfactory performance had been reached in simulation, the control program would be re-implemented to interface with the sensors and actuators on real robots. Although some of the logical errors in the code usually had been caught in the initial version developed for the simulator, the new version for the real robots would usually still have to undergo a non-negligible debugging phase. In case experimentation on real robotic hardware led to changes concerning the high-level logic of the controller, those changes had to be implemented in the simulator version of the control program if results obtained in simulation were needed. In this way, two versions – one for a particular simulator and one for the real robots – had to be developed and maintained.

The Common Interface is a simple idea that alleviates the need for reimplementing or changing code in order to move a control program from simulation to real hardware and back. The Common Interface is basically the real robot API encapsulated in a C++ interface (an abstract class) and some scaffolding code to startup, initialize and run control programs. Control programs read sensors and control actuators through the Common Interface. On real robots, a call to a Common Interface method is mapped to the corresponding real robot API call. In simulation, on the other hand, arguments and return values are converted and/or computed in order to comply with the representations and the interface of the simulator. In this way, all the simulator-specific code lives inside the Common Interface and is invisible to the control program. The same control programs can thus be compiled and run in simulation and on real robots.

The Common Interface does not remove the need for testing and debugging code on real robots since there will always be differences between simulation and reality. However, since the same control program runs in both simulation and on real robots, only one version of the control program needs to be maintained. It is hard to quantify the benefits of the approach, but we believe that the Common Interface has significantly shortened the development cycles for control programs.

Simulation and Evolutionary Robotics

Our initial work in the field of evolutionary robotics moved us to develop TwoDee. Artificial evolution of robot controllers is often carried out in simulation for practical reasons. It is common to evaluate thousands of controllers in order to find a good solution. Running the evolution in a software simulator can be orders of magnitude faster than on real robots. Furthermore, there is no risk of damaging hardware, no need to re-calibrate sensors and actuators, and robots do not run out of battery[3] in simulation.

Evolving controllers in software simulators is, however, not a perfect solution. We can run into a number of problems when we try to transfer controllers that have been evolved in software simulators to real, physical robots. An evolved controller might rely on subtle cues and symmetries present in a simulated environment, which are slightly different in the real world. Real sensors and actuators are not ideal, meaning that they are not always as precise and reliable as one could hope for. The complex dynamics of the real world cannot be simulated with perfect accuracy. These aspects have to be taken into account if the controllers evolved in software are meant to be used on physical robots. Unfortunately, there is no general method to ensure that controllers evolved in software simulation are transferable to physical robots. An obvious approach is to try to narrow the gap between simulation and reality as much as possible – that is, to develop or use software that attempts to simulate reality faithfully. This is typically done by modeling robots and the environment in high detail and by using a software dynamics engine, such as Vortex and Open Dynamics Engine (ODE).

During the life-time of the *swarm-bots* project a number of simulators have been built and a quick web search shows that several generic and architecture-specific mobile robot simulators exist. We chose to develop an entirely new simulator for multiple reasons, the two major ones being *performance* and *flexibility*. When the work described in this thesis began, we tested a number of simulators. We evaluated one of the more mature simulators called *MISS* which uses the Vortex dynamics engine. However, with the available hardware resources at the time[4] an average evaluation consisting of 100 individuals per generation, each evaluated for 400 control cycles (40 seconds), for 100 generations, took on average between 10 and 12 hours. Now in order to draw any general conclusions on the feasibility of a given evolutionary setup, it is necessary to run multiple evolutions with different initial random seeds: Running at least 10 and often 20 or more evolutions is common practice in literature, which means that a given evolutionary strategy takes many days to evaluate. If only few strategies need to be evaluated, this is not a major problem. However, a significant amount of trial-and-error is often necessary in order to shape evolution so that the solutions found are satisfactory and thus the evaluation time makes the whole process tedious.

---

[3] Unless, of course, the battery power plays a role for the controllers being evolved and therefore is modelled explicitly in software; for instance if a robot should learn to move to a special area to recharge when its battery level gets low.

[4] Available hardware resources: 15 single AMD Athlon 1800-2800+ nodes and 16 dual Opteron 1.8 GHz nodes shared between the researchers in our lab. Estimates are based on the assumption that we have exclusive access to one third of the processing power, that is, 16 CPUs.

Another widely used, relatively inexpensive strategy for easing the transfer of controllers from simulation to reality involves adding noise to sensory reading and to actuator outputs. The effect of adding noise is two-fold: Real sensors and actuators are noisy by nature and controllers should be robust enough to handle this; furthermore, small differences between the simulated and real versions of the sensors and actuators can be masked by the noise. Jakobi advocates an extreme approach in which the parts of the simulated world (e.g. the presence of a corridor) that a robot should not rely on are hidden by a noise [Jakobi, 1997a,b, Miglino et al., 1995]. While adding noise to sensors and actuators does not have a significant effect on a simulator's performance, the use of detailed robot and environment models and rigid body engines has a huge impact on the number of CPU cycles required for running evolution in simulation.

We built TwoDee to be flexible in the sense that different versions of the same sensors could easily be implemented. Different researchers and different experiments require different implementations of sensors and actuators. Combined with our custom rigid body physics engine, TwoDee performs quite well: we benchmarked TwoDee against another simulator, NS, developed towards the end of the *swarm-bots* project. NS is built on a general-purpose rigid-body dynamics engine ODE. The benchmark results showed that TwoDee outperforms NS by several orders of magnitude, and that TwoDee scales close to linearly, both time-wise and space-wise, in the number of physically connected robots simulated (see [Christensen, 2005] for detailed benchmark results). With TwoDee, we have successfully been able to simulate up to 1.000.000 connected robots on a normal workstation[5].

TwoDee, TwoDeePuck and the Autonomous Robots Go Swarming Simulator

Parts of TwoDee have been used in two other simulators:

- *TwoDeePuck* [Bury, 2007] is a simulator for the *e-puck* robotic platform. The structure and non-*s-bot* specific code of TwoDee was used and in some cases modified and extended in order to simulate groups of *e-puck* robots. The resulting simulator, TwoDeePuck, has been used by several undergraduate and master students.
- Autonomous Robots Go Swarming (ARGoS) is a simulator under development as part of the *swarmanoid* project (see http://www.swarmanoid.org/simulation). The goal of ARGoS is to allow for the simulation of different types of interacting robots in complex man-made environments, e.g. a kitchen or a living room. The simulator is adaptive

---

[5] A normal workstation in this case is an Athlon XP 1800+ with 512 MB of RAM running Debian Linux. When TwoDee simulates 1.000.000 physically connected robots the memory usage is approximately 500 MB of RAM.

in the sense that the user can replace or choose between various implementations of central components, such as the physics engine. The simulator also allows the user to use different physics engines at the same time, e.g. a simple 2D physics engine for robots and items on the floor plane combined with a specialized 3D physics engine for flying robots. Parts of the code for TwoDee has been modified and is used in ARGoS, namely its specialized physics engine, some sensor and actuator implementations, components for rendering and the concept of the Common Interface.

At the time of writing, TwoDee is still being used as a platform for controller development and test.

1.4 Summary

In this chapter, we have motivated why fault detection and fault tolerance are important subjects to study before autonomous robots can be widely adopted. As we entrust more and more tasks to robots, it becomes increasingly important to ensure that the robots remain safe – even in the event of partial or complete failure. We stated the main topic of this thesis, namely endogenous and exogenous fault detection for autonomous robots. We listed the original research and the publications on which this thesis is based. We briefly presented original research related to two other topics in which we have made scientific contributions: morphology control and evolutionary robotics. Finally, we presented TwoDee, a fast and flexible simulator that was initially developed with evolutionary robotics in mind. Over the past years, the simulator has grown, it has been used by several researchers, and it has been used in part in two other simulators, namely TwoDeePuck and ARGoS.

Related Work

Chapter 2

Fault detection is based on observations of a system's behavior (for an introduction see [Isermann, 1997]). Deviations from normal behavior can be interpreted as symptoms of a fault in a system. Fault detection is an important activity in many automated systems ranging from car engines to complex chemical plants [Gertler, 1998]. A specific fault detection approach is a concrete method for *observation processing*. Observations can for instance be the current temperature of a car engine or the measured flow of a solution through a certain part of a chemical plant. Some faults can be detected by checking the observations against some predefined limits, e.g. 100 degrees Celsius or 0.2 l/s. Under normal operation, these limits should not be exceeded and in case they are, it may indicate the presence of a fault.

Fault detection can be achieved by adding special-purpose hardware such as torque sensors and encoders [Terra and Tinos, 2001]. However, the sensors responsible for detecting faults are subject to faults themselves. Thus, we are faced with a dilemma: the more hardware we add in order to facilitate fault detection, the more faults we have to consider. Adding hardware also increases cost, complexity and power consumption. It is, therefore, something that we would like to avoid in many cases. Reducing cost and complexity would, for example, be crucial in projects such as the National Aeronautics and Space Administration's (NASA) swarm missions, in which cooperating swarms of hundreds to thousands of small-scale autonomous robots explore the solar system [Hinchey et al., 2004]. Given the high number of robots, simplicity and small size are high priorities. Similarly, for domestic adoption of service and leisure robots, the number and complexity of components have to be kept low in order to reach a price point that allows for high market penetration [Kochan, 2005].

If a wheel on a robot breaks, how can the robot discover this? We could add rotary encoders that measure the angular rotation of the robot's wheels and verify that the shafts are turning as they are supposed to. As mention above, this increases the cost and complexity of the robot. Furthermore, if the motors that drive the robot are operating normally but if one of the wheels has fallen off, we will be unable to detect this if we rely on the readings from the encoders only. Alternatively, we could construct a model of how the robot is supposed to move given the speeds of the left wheel and the right wheel, respectively. Determining how an autonomous robot actually moves is however difficult, especially when a robot is operating in an unknown environment. Even when sensors, such as a GPS receiver, are available, noise and poor resolution of the readings can make it hard to determine if a robot is operating normally or if it is subject to a fault. In Section 2.1, we discuss these issues further and we review existing methods for endogenous fault detection in autonomous robots.

Some faults cannot be detected by the robot in which they occur: For instance a dead battery, a short-circuit on the main board, or a bug that causes the on-board software to hang.[1] Systems composed of multiple, homogeneous robots can potentially exploit their inherent redundancy in order to achieve a high degree of tolerance to individual failures. To realize this potential, it is in general necessary that robots can detect faults in each other. In Section 2.2, we discuss proposed techniques for exogenous fault detection and fault tolerance in multi-robot systems.

---

[1] Some tolerance to these types of faults can be achieved by adding redundant software and hardware components

2.1 Endogenous Fault Detection

Endogenous fault detection generally falls into two categories: *model-based* methods (analytical) and *model-free* methods (data-driven). A large body of research in model-based fault detection approaches exists [Gertler, 1988, Isermann and Ballé, 1997]. In model-based fault detection, some model of the system or of how it is supposed to behave is constructed. The actual behavior is then compared to the predicted behavior and deviations can be interpreted as symptoms of faults. A deviation is called a *residual*, that is, the difference between the predicted and the observed value.

In the domain of mobile autonomous robots, accurate analytical models are often not feasible due to uncertainties in the environments, noisy sensors, and imperfect actuators. A number of methods have been studied to deal with these uncertainties. Many of these methods are model-free since they are not based on analytical models of the systems but instead data-driven. Fault detection is nevertheless often performed based on residuals. Artificial neural networks and radial basis function networks have, for instance, been used for fault detection and identification based on residuals [Vemuri and Polycarpou, 1997, Terra and Tinos, 2001, Patton et al., 2000]. In Skoundrianos and Tzafestas [2004], the authors train multiple local model neural networks to capture the input-output relationship for the components in a robot for which faults should be detected. The authors focus on detecting faults in the wheels of a robot, and the input and the output are the voltage to the motor driving a wheel and the speed of the wheel, respectively. Supervised learning is used to train the local model neural networks. During operation, the speeds predicted by the local models are compared to the actual speed and the residuals are computed.

Another popular approach to fault detection is to operate with multiple global models of the system concurrently. Each model corresponds to the behavior of the system in a given fault state, for example a broken joint, a flat tire, and so on. The fault corresponding to a particular model is considered to be detected when that model's predictions are a sufficiently close match to the currently observed behavior. Banks of Kalman filters have been used for such state estimation [Roumeliotis et al., 1998, Goel et al., 2000]. In their basic form, Kalman filters are based on the assumption that the modeled system can be approximated as a Markov chain built on linear operators perturbed by Gaussian noise [Kalman, 1960]. Robotics systems are, like many other real-world systems, inherently nonlinear. Furthermore, discrete fault state changes can result in discontinuities in behavior. Extensions, such as the *extended Kalman filter* (EKF) and the *unscented Kalman filter* (UKF), overcome some of these limitations. In EKFs, the state transitions and the models can be non-linear functions, but they must be

---

as it is often done in critical systems such as aircraft and spacecraft. However, as we have discussed, adding hardware components and increasing development cost are not always feasible options.

differentiable so that the Jacobian matrix can be computed. In UKFs, a few sample points are picked and propagated through the model allowing the mean and covariance to be estimated even for models comprised of highly non-linear functions [Julier and Uhlmann, 1997]. EKFs and UKFs have been extensively used for localization for mobile robots [Smith and Cheeseman, 1986, Leonard and Durrant-Whyte, 1991, Ashokaraj et al., 2004], but in the domain of fault detection and fault identification for autonomous robots these techniques are often used in combination with other methods.

Dynamic Bayesian networks represent another technique that does not require that the underlying phenomenon can be reasonably modeled as a linear system [Lerner et al., 2000]. Recently, computationally efficient approaches for approximating Bayesian belief using *particle filters* have been studied as a means for fault detection and identification [Dearden et al., 2004, Verma et al., 2004, Li and Kadirkamanathan, 2001]. Particle filters are Monte Carlo methods capable of tracking hybrid state spaces of continuous noisy sensor data and discrete operation states. The key idea is to approximate the probability density function over fault states by a swarm of points or particles. One of the main issues related to particle filters is tracking multiple low-probability events (faults) simultaneously. A scalable solution to this issue has recently been proposed [Verma and Simmons, 2006].

Artificial immune-systems (AIS) are a biologically inspired approach to fault detection. An AIS is a classifier that distinguishes between *self* and *non-self* [Forrest et al., 1994]. In fault detection, “self” corresponds to fault-free operation while “non-self” refers to observations resulting from a faulty behavior. AIS have been applied to robotics, see for example [Canham et al., 2003] in which fault detectors are obtained for a Khepera robot and for a control module of a BAE Systems Rascal robot. The two systems are first trained during fault-free operation and their capacity to detect faults is then tested.

Marsland et al. [2005] have suggested using a *novelty filter* based on a clustering neural network and *habituation* for inspection and fault detection. Through unsupervised learning, a novelty filter learns to ignore sensory data similar to data previously perceived. The authors evaluated the approach in various configurations using a Nomad 200 robot placed in different environments and the robot correctly detected environmental differences. Novelty detection has the potential to be used for fault detection assuming that a fault would have some impact on what a robot perceives. However, when a novelty filter detects a new situation some supervisor has to determine if the novelty of the situation is due to external factors such a new type of environment or due to an internal fault. Fault detection through novelty detection is therefore best suited for cases in which robots operate semi-autonomously, such as in inspection tasks in which an operator is notified whenever a robot encounters a novel situation.

The approach that we propose in Chapter 4 relies on artificial neural networks that are trained to discriminate between behaviors when a robot is operating normally and behaviors when the presence of a fault is affecting the robot’s performance. We rely on a single neural network only, as opposed to the multiple local model neural networks and the multi-model approaches mentioned above. We do not use explicit or analytical modeling of the system (which can be complicated for all but the simplest systems). The approach that we suggest for endogenous fault detection in Chapter 4 shares features with artificial immune system-based fault detection approaches in the sense that a classifier discriminates between normal and incorrect behavior without the use of any explicit model. The aim is to detect the difference between normal and abnormal behaviors, based on the flow of information from a robot's sensors and the subsequent flow of control signals from the control program to the robot's actuators. However, in contrast with the studies on AIS and novelty detection, we use supervised learning. When an artificial neural network is trained, we include training data with both positive and negative examples, which potentially allows the proposed method to be extended to fault identification.

2.2 Exogenous Fault Detection

In robotics, exogenous fault detection is the process through which one robot detects faults that occur in other, physically separate robots.

Parker [1998] has demonstrated that cooperating teams of robots based on the ALLIANCE software architecture can achieve a high degree of fault tolerance. Fault tolerance is obtained by modeling "motivations" mathematically and by adaptive task selection based on these motivations. When a robot fails to register satisfactory progress in its current task (for instance due to the presence of a fault), it decreases its motivation for performing the task. Eventually, the robot will switch to another task that it may still be able to perform. Alternatively, another robot will discover that there is limited or no progress in the task undertaken by the failed robot, and take over. Other approaches such as MURDOCH [Gerkey and Matarić, 2002b,a] and TraderBots [Dias et al., 2004] have been proposed. In both MURDOCH and TraderBots, explicit communication is used to negotiate task allocation. Fault detection and fault tolerance are built into this negotiation process.

The approaches mentioned above are all for multi-robot systems which consist of relatively complex robots. The robots often allocate tasks and track the progress of the mission by communicating over a network. Other multi-robot systems consist of multiple, relative simple robots. In these systems, the behavior of a robot is typically determined by what robot can perceive in its immediate surroundings (as opposed to what it has negotiated over a network). Each robot acts according to a limited set of rules. Multi-robot systems in this category are often referred to as swarm robotic systems (the multi-robot system used in the research covered in this thesis, the *swarm-bot* platform, is a swarm robotic system, see Chapter 3).

In some cases, fault tolerance is an inherent property of the swarm robotic system and is not handled explicitly. Lewis and Tan [1997] have shown that their control algorithm for maintaining geometric formations exhibits correct behavior even if one of the robots fails. Winfield and Nembrini [2006] performed failure mode and effect analysis (FMEA) on a containment task for a swarm of robots connected through local wireless links. The authors found that their system exhibited a high level of tolerance to individual failures, but at the same time that certain types of faults could effectively anchor the swarm at the failed robot's position. In their conclusion, they envisage a new behavior in which the swarm can identify failed members and isolate them from the rest of the swarm. In both [Lewis and Tan, 1997] and [Winfield and Nembrini, 2006] mentioned above, fault tolerance is a consequence of the simple and adaptive nature of the controller design and not of explicit fault detection, identification and accommodation. Although attempts to generalize fault tolerance by design have been made (see for instance [Winfield et al., 2005]), it is unknown whether such designs are feasible (or even theoretically possible) for all systems and all tasks.

In [Li and Parker, 2007], the authors use sensor analysis to facilitate fault detection in a tightly-coupled multi-robot team. Their "sensor analysis for fault detection" (SAFDetection) is based on data collection during normal operation and subsequent abnormality detection. The approach combines data clustering techniques with the generation of a probabilistic state diagram to model normal operation of the multi-robot system. The whole multi-robot team is regarded as a single monolithic entity with a unified set of sensors. This puts limits on the size of the teams since the amount of data communicated and processed centrally is proportional to the number of robots in the team. In [Li and Parker, 2007], the authors state that they intend to extend their approach to distributed monitoring.

In Chapter 5, we study a completely distributed approach that builds on the principles behind synchronization observed in fireflies to implement a heartbeat-like fault detection scheme for swarms of robots. Fault tolerance is achieved by explicit exogenous fault detection (as opposed to implicit fault tolerance by design). Unlike many previously studied approaches, the approach that we propose does not depend on radio communication, a centralized coordination mechanism, or a strict definition of tasks and progress.

Robotic Hardware

Chapter 3

In this chapter, we present the multi-robot platform, *swarm-bot*, which has been used for the research covered in this thesis. We discuss how it compares to other multi-robot systems, and we provide examples of studies conducted on this platform. In multi-robot systems fault detection and fault tolerance are central issues, especially when the robots often work in close collaboration, e.g. to transport a heavy object. In case one robot fails, the whole system could stop functioning if the other robots are unable to detect the fault and take the necessary steps to exclude and/or repair the failed robot. On the other hand, the robots in such systems can benefit from redundancy: in case one robot breaks down, another robot can take steps to repair the failed robot or take over the failed robot's task. In order to leverage the redundancy, however, the robots need to have the capacity to detect and act upon faults that occur in one another.

3.1 The *swarm-bot* Platform

The research covered in this thesis was conducted on the *swarm-bot* robotic platform [Mondada et al., 2005]. This innovative platform was designed and built by Mondada's group at the École Polytechnique Fédérale de Lausanne in Switzerland. The *swarm-bot* system consists of a number of mobile autonomous robots called *s-bots*. An *s-bot* with indications of its sensors and actuators is shown in Figure 3.1.

The *s-bot* is 10 cm high without the perspex tube housing its camera mirror and has a diameter of 11.6 cm without its gripper. Thanks to a traction system that combines tracks and wheels, the *s-bot* can rotate efficiently about its own axes and navigate on uneven terrain. The body of the *s-bot* contains the majority of the *s-bot* sensory and processing systems. The body is mounted above the chassis. A motorized axis allows the *s-bot* body to rotate with respect to the chassis.

S-bots have the capacity to form physical connections with each other. Physical connections between *s-bots* are established by a gripper-based connection mechanism, see Figure 3.2. The entity formed by two or more connected *s-bots* is called a *swarm-bot* (in the examples shown in Figure 1.1 on page 6, the *s-bots* form *swarm-bots*). Each *s-bot* is surrounded by a transparent ring that contains 8 sets of RGB-colored LEDs. This LED ring can be grasped by other *s-bots*. An optical light barrier inside the *s-bot* gripper indicates when another *s-bot*'s LED ring (or another graspable object) is between the jaws of the gripper. The LEDs in the ring can be individually controlled. The two examples in Figure 3.3 show an *s-bot* with all its LEDs off and illuminated, respectively.

The *s-bot* has 15 proximity sensors distributed around its body that allow for the detection of obstacles. A 3-axis accelerometer provides information on the *s-bot*'s inclination that can be

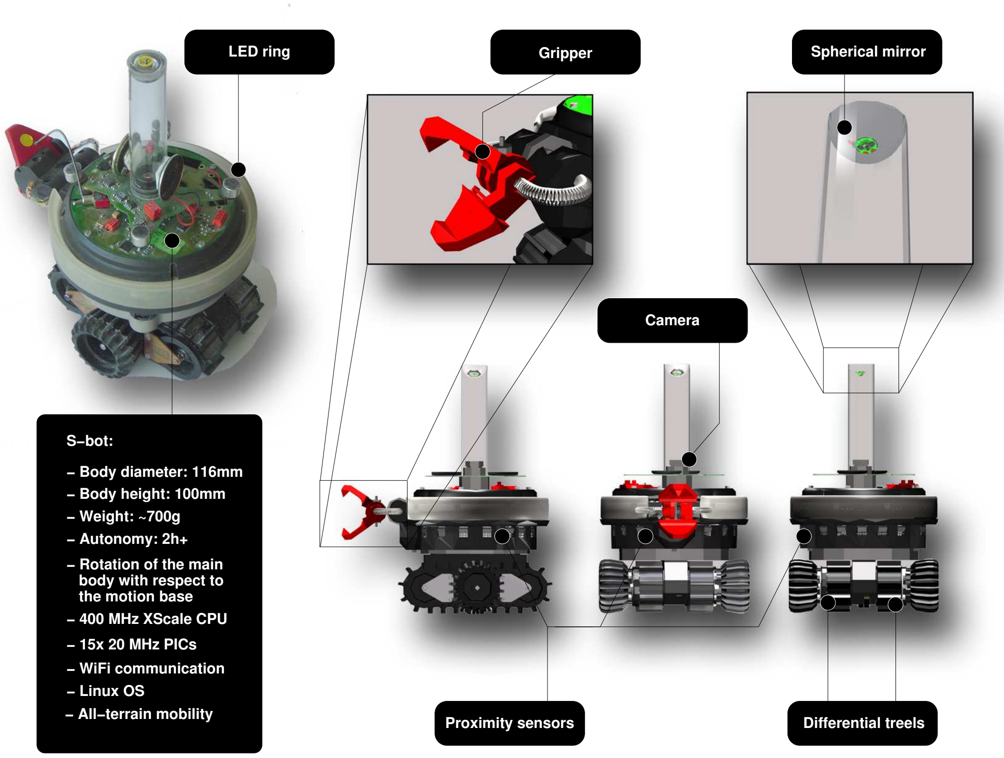

LED ring
Gripper
Spherical mirror
Camera
S–bot:
– Body diameter: 116mm
– Body height: 100mm
– Weight: ~700g
– Autonomy: 2h+
– Rotation of the main body with respect to the motion base
– 400 MHz XScale CPU
– 15x 20 MHz PICs
– WiFi communication
– Linux OS
– All–terrain mobility
Proximity sensors
Differential treels

Figure 3.1: The *s-bot*: An autonomous, mobile robot capable of self-assembly. Processor: 400 MHz XScale CPU, operating system: Linux, weight: ∼700 g, battery allows for ∼2 h of operation between recharges.

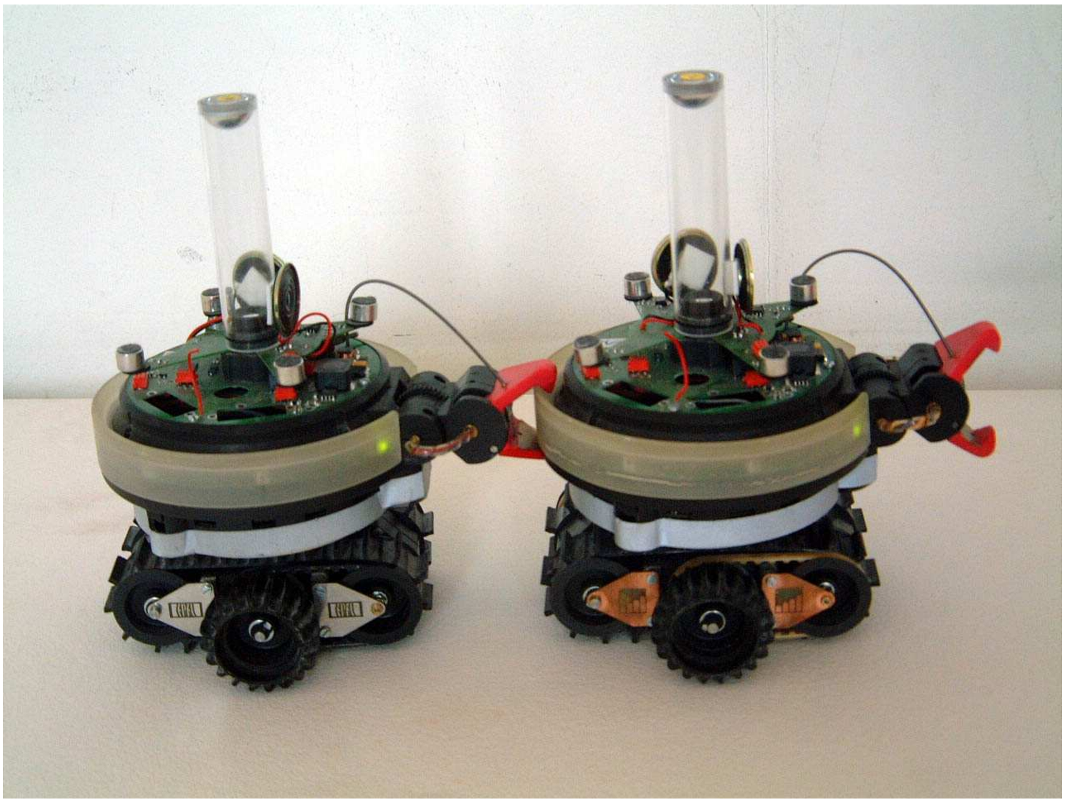


Figure 3.2: Illustration of the gripper-based connection mechanism: One *s-bot* grasping the transparent LED-right of another *s-bot*.

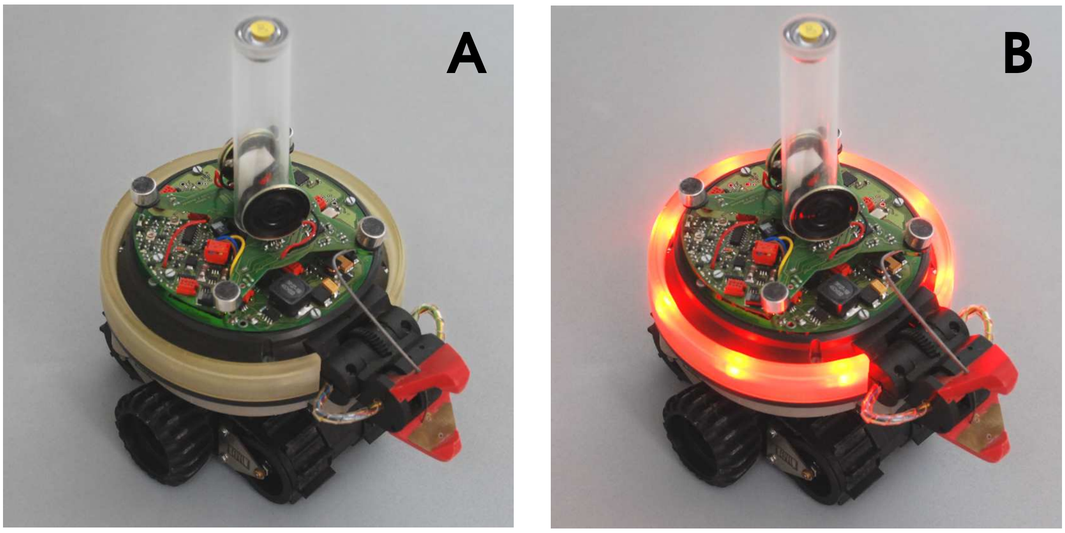


Figure 3.3: A: an *s-bot* with all its LEDs off. B: an *s-bot* with its red LEDs illuminated.

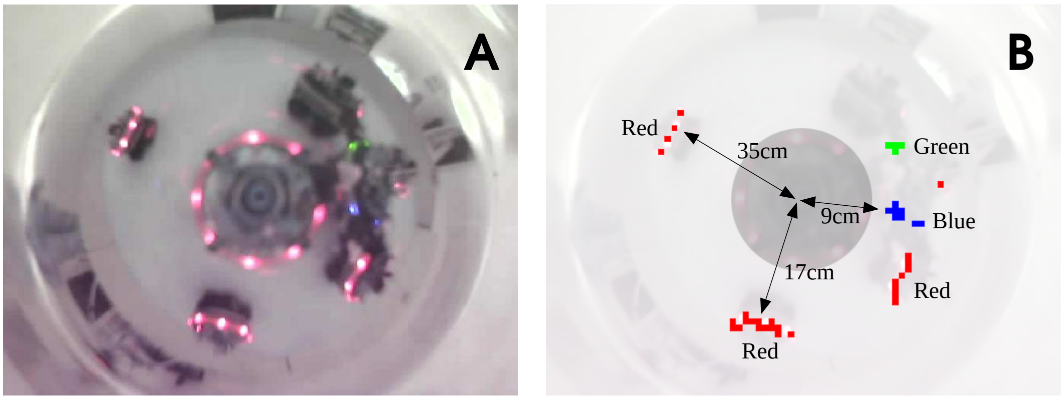


Figure 3.4: An image captured by a robot's omni-directional camera and the processing steps to obtain information about the LEDs of nearby robots. A: The captured image. B: After color segmentation with indications of the distance estimates from the robot that captured the image to some of the LEDs detected.

used to detect if the *s-bot* is in danger of toppling over. Other sensors provide the *s-bot* with proprioceptive information about its internal motors. This includes positional information (e.g., of the rotating turret) and torque information (e.g., of forces acting on the traction system). Traction sensors are mounted in the junction between the chassis and the main body. The traction sensors enable an *s-bot* to perceive forces acting on the chassis, such as when another physically connected robot pulls or pushes the chassis. The traction sensors can help coordinate physically connected *s-bots* because each robot can perceive when the rest of the *swarm-bot* attempts to move or change direction.

Each *s-bot* is equipped with a 400 MHz XScale CPU and runs a customized version of the Linux operating system. Control programs for *s-bots* are usually written in C or C++. The sensors can be read and actuators can be controlled through an application programming interface (API). Control programs for *s-bots* operate in a discrete manner: a control program is run as a succession of sense-think-act cycles. In each cycle, the control program reads data from sensors such as the on-board camera, infrared proximity sensors, and so on, processes the data, and sends control signals to the actuators such as the motors that drive the robot. A control cycle period of 0.10-0.15 s is common.

3.1.1 The *s-bot* Camera and Image Processing

The *s-bot* has omni-directional vision, achieved using a camera that points upwards at a hemispherical mirror. A transparent perspex tube holds the mirror in place. The camera captures images of the robot's surroundings reflected in the hemispherical mirror. When the *s-bots* operate on flat terrain, the distance in pixels from the center of an image to a perceived object can be used to estimate the physical distance between the robot and the object. An example is shown in Figure 3.4.

The camera sensor records 640x480 pixel color images. The *s-bots* have sufficient on-board processing power to scan entire images and identify objects based on color information. The image processor is configured to detect the location of the colored LEDs of the *s-bots* and discard any other information. The image processor divides the image into a grid of multi-pixel blocks and returns the color prevalent in each block (or indicates the absence of any color). Depending on light conditions, the camera can detect illuminated LEDs on other *s-bots* up to 50 cm away.

3.1.2 Examples of Studies Conducted with *S-bots*

The *swarm-bot* platform was novel given that it was one of the first multi-robot systems in which the units can self-assemble into larger robotic entities (*swarm-bots*) while still be sophisticated enough to carry out meaningful tasks individually.

A number of studies have been dedicated to the subject of self-assembly on the *swarm-bot* platform. The challenge lies in coordinating a number of autonomous *s-bots* so that they connect physically and form a connected robotic entity - a *swarm-bot*. It has been demonstrated that control programs – partly evolved and partly hand-coded – can steer the *s-bots* to self-assemble [Gross et al., 2005, 2006a]. One of the fundamental issues is deciding how to initiate the self-assembly process. In [Gross et al., 2005], different colors were used to distinguish between the robot (or object) that seeds the self-assembly process and the rest of the group. A robot operating individually would illuminate its blue LEDs, while a robot that was part of a robotic entity would illuminate its red LEDs. When blue robots connect to the structure, they themselves become red in order to attract other robots.

Recently, control programs based almost entirely on neural networks were synthesized through artificial evolution [Tuci et al., 2008, Ampatzis et al., 2008]. The neural controllers had no control over the colored LEDs and the robots did not change color depending on their role. Evolution found a solution to the role allocation problem (that is, the problem of deciding who should grip who) in which the robots perform repetitive oscillating movements – swinging from left to right. At one point, the robots make a collective decision: one moves and initiates the self-assembly process, while the other turns its gripper away from the approaching robot in order to let itself be grasped.

Once assembled, physically connected *s-bots* need to coordinate when they have to move, e.g. to overcome a steep hill or transport a heavy object. The robots basically have to agree on a common direction of movement. Trianni *et al.* have evolved neural network-based controllers for coordinated motion and hole avoidance [Trianni et al., 2004, Trianni and Dorigo, 2006]. Coordination between the robots is facilitated by the traction sensors: the robots can detect forces acting on their turret in the horizontal plane. In subsequent work, we have evolved neural network-based controllers capable of coordinated motion while moving towards a light source and avoiding holes [Christensen and Dorigo, 2006a,b].

It has been demonstrated that physically connected robots are able to carry out tasks that a single robot could not. O'Grady *et al.* have demonstrated how cooperating *s-bots* are able to overcome a steep hill on which an *s-bot* operating alone would topple [O'Grady et al., 2005, 2007c, 2008b]. Gross et al. [2006a], Tuci et al. [2006] and Gross and Dorigo [2008a] have demonstrated how *s-bots* can attach to a heavy object (and to each other) and transport the object. Nouyan et al. [2006] and Nouyan et al. [2008] have shown how a group of *s-bots* is able to form logical chains of robots from a predefined location (the nest) to an object (the prey) and to transport the object along the chain to the nest.

As mentioned in Section 1.3.1, the morphology formed when robots self-assemble is important. In collaboration with Marco Dorigo and Rehan O'Grady, we have demonstrated how the morphology generated through self-assembly can be controlled [Christensen et al., 2007a,c, O'Grady et al., 2007a, Christensen et al., 2007b, O'Grady et al., 2007b, 2008a, Christensen et al., 2008]. The goal of our ongoing research is to let robots identify different types of obstacles and then assemble into appropriate morphologies in order to overcome the obstacles.

3.2 Other Multi-Robot and Modular Robotic Systems

Research in multi-robot systems began in the domain of modular self-reconfigurable systems. A modular self-reconfigurable robotic system is composed of numerous units and is able to change its own shape by rearranging the connectivity between its units. A system can reconfigure in order to adapt to new circumstances, perform new tasks, or recover from faults. In order to recover from a fault and self-repair, the units need to be able to detect faults.

The past 20 years have produced a large body of research in self-reconfigurable modular robotic systems and associated connection mechanisms. The individual modules vary in autonomy of control from system to system. However, the individual modules are usually simple and can seldom – as opposed to *s-bots* – carry out meaningful tasks independently.

Fukuda *et al.*'s CEBOT system [Fukuda et al., 1991, Kawauchi et al., 1993] is one of the first realizations of a reconfigurable modular robotic system. The architecture consists of heterogeneous modules with different functions, e.g. to rotate, move, and bend. Various prototypes of the CEBOT system comprising different shapes and connection mechanisms have been studied.

Hirose *et al.*'s Gunryu concept [Hirose et al., 1996] consists of a number of autonomous mobile units each equipped with an actuator that allows the modules to form physical connections with each other. However, only two prototype modules connected by a passive arm have been built.

PolyBot [Yim et al., 2003, 2000] is a modular chain robot in which each module has one degree of freedom. It has been demonstrated that an arm consisting of multiple PolyBot modules is capable of operating in 3D space and that such an arm can grasp and dock with additional modules. The PolyBot system uses hermaphroditic connection plates with four grooved pins and four holes on each plate. When two modules connect, the grooved pins on the connection plate of one module match up with the four holes on the connection plate of the other module. Connections are formed and released by a shape-memory alloy latching mechanism.

The CONRO [Castano et al., 2000] system consists of a number of roughly shaped rectangular boxes with a female connector on one face and male connectors on three of the other faces. Rubenstein *et al.* have recently shown that CONRO is capable of autonomous docking (self-assembly) [Rubenstein et al., 2004].

In the Millibot Train system [Brown et al., 2002], multiple mobile robots can physically connect into a line formation. The objective is to enhance the mobility of the system, for instance, enabling a train of connected robots to cross obstacles that a single robot would not be able to cross. So far, only a teleoperated train composed of seven prototype modules has been demonstrated [Brown et al., 2002].

The systems presented above all employ connection mechanisms based on penetration and shape matching. This requires precise alignment of the modules when connections are formed. Furthermore, connections between modules can only be made at predefined locations on the module bodies, sometimes only at a single location.

Super-Mechano Colony (SMC) [Damoto et al., 2001, Yamakita et al., 2003] is composed of one parent unit and several child units that are responsible for locomotion when they are attached to the parent unit. In the SMC Rover [Motomura et al., 2005], the child units are called Uni-Rovers. Each Uni-Rover is composed of a wheel and a single manipulator. The Uni-Rovers can attach to the parent unit at one of six locations. When six Uni-Rovers connect to the parent unit, the artifact effectively becomes a six wheeled rover. For another prototype of the SMC, Gross et al. [2006b] showed self-assembly of two child units.

M-TRAN [Murata et al., 2002] is a hybrid system in which each module consists of three parts: an active block, a passive block and a link between them. A module is equipped with an on-board microprocessor, inter-module communication/power transmission devices and inter-module connection mechanisms. Each block has three connection surfaces composed of a combination of permanent and electrical magnets. M-TRAN is able to metamorphose into different robotic configurations. Examples include a legged machine for which a coordinated walking motion was generated.

SuperBot [Shen et al., 2006] is a new deployable self-reconfigurable system for real-world applications outside laboratories. The system takes inspiration from CONRO and M-TRAN. SuperBot combines the advantages of chain-based and lattice-based robotic systems to accomplish multi-modal locomotion. Each module can dock with other modules in six different positions. A prototype consisting of six modules has been built [Salemi et al., 2006].

ATRON [Østergaard et al., 2006] is a cubic lattice-based reconfigurable robot. It is composed of self-contained modules that can attach to each other, share power, communicate to form a larger robotic entity and reconfigure into different shapes. Each module is composed of two hemispheres, connected by a slip-ring. The two hemispheres can rotate independently while still transfer power and communicate with each other.

For detailed overviews of self-reconfigurable systems and self-assembling systems see [Yim et al., 2007], [Gross et al., 2006a] and [Gross and Dorigo, 2008b]

Flexibility is one of the key potential advantages of multi-robot systems in which the units have the capabilities to carry out tasks individually. Multi-robot systems composed of units that are capable of carrying out tasks individually is a topic of increasing interest to researchers. Multi-robot systems of this flavor have a number of advantages over single robot systems. In particular, the robots in such systems can either execute tasks in parallel, or can carry out more demanding tasks by cooperating with each other. Such systems can be used for surveillance [Roman-Ballesteros and Pfeiffer, 2006], physical manipulation [Matarić et al., 1995, Gerkey and Matarić, 2002b], exploration [Burgard et al., 2000], and so on.

Multi-robot systems of this type can be composed of robots that were designed to operate alone, for instance Sony's AIBO [Fujita et al., 2000], or they can be composed of robots that were designed to cooperate such as the *s-bots*. Examples of other multi-robot systems that consist of autonomous mobile robots are I-SWARM, [Woern et al., Sept. 2006, Seyfried et al., 2005], iRobot's SwarmBots [McLurkin, 2004, McLurkin and Smith, 2004], and Pherobots [Payton et al., 2001]. The Khepera [Mondada et al., 1994] is a miniature robot designed to allow researchers to efficiently (thanks to its small size and relatively low cost) investigate control algorithms. The Khepera robot has been quite successful and numerous studies in collective robotics have been conducted on this platform. A new miniature robot architecture, the *e-puck* (see www.e-puck.org), has recently been developed at Ecole Polytechnique Fédérale de Lausanne. The e-puck is an open and extendable hardware platform. Similar to the Khepera, the e-puck is mainly for education and research.

Multi-robot systems can also consist of different types of robots: In the *swarmanoid* project (see http://www.swarmanoid.org), the aim is to build a heterogeneous swarm of robots consisting of three different types of robots – foot-bots, hand-bots and eye-bots – that can cooperate in order to carry out tasks in indoor environments. The eye-bots (flying robots) can guide foot-bots (operating on the ground) and hand-bots (operating on the ground, on tables, shelves, etc.).

3.3 Summary

In this chapter, we presented the *swarm-bot* platform used for the research presented in this thesis. The platform is a multi-robot system which is composed of a number of autonomous robots called *s-bots*. *S-bots* can physically connect and form larger robotic entities – *swarm-bots*. We discussed some of the studies that have been conducted on the *swarm-bot* platform such as self-assembly, coordinated motion, cooperative transport and morphogenesis.

Over the past 20 years researchers have been increasingly interested in building and controlling self-reconfigurable and multi-robot systems. These systems have a number of potential advantages over single robot systems, such as flexibility, adaptability, and inherent redundancy at the unit level. Fault detection and fault tolerance is particularly important in these systems, since robots often have to work together in order to complete tasks. If one robot fails, it could compromise the whole group and its mission. However, when faults are correctly detected, these systems could achieve a high level of tolerance to individual failures due to the redundancy at the unit level.

Fault Detection based on Fault Injection and Learning

CHAPTER 4

In this chapter, we propose a new method for synthesizing fault detection modules for autonomous robots. The method requires no special fault detection hardware and relatively little computational resource to run the fault detection software. Fault detection has been a topic of many previous studies and a popular approach for detecting faults is to build a model of how a system or a robot is supposed to behave. If the difference between the actual behavior and the behavior predicted by the model differ, it could be due to the presence of a fault. However, building an analytical model of how an autonomous robot is supposed to behave is a non-trivial task: due to noisy sensors, imperfect actuators and general uncertainty about the environment, this approach is difficult to apply in practice.

The method that we propose is model-free since is does not rely on an explicit model of the system. Instead, we record sensory data, firstly over a period of time when a robot is operating normally, and secondly over a period of time when various types of hardware faults are present in the robot. We then train a neural network to detect faults based on the data recorded.

In the following section, we provide an overview and a formal description of the proposed approach. In Section 4.2, we present three different tasks in which we test our fault detection approach. We then go on to present how we collect training data and how we evaluate the performance of fault detectors (see Section 4.3). In Section 4.4, we present results. We first consider only faults in the traction system of the robots and evaluate the performance of fault detectors for the three tasks (see Section 4.4.1 to 4.4.3) and we then go on to test the performance when faults in both sensors and actuators are considered (see Section 4.4.4). Since autonomous mobile robots often navigate in uncertain environments, we test the performance of our approach when a task varies from trial to trial (see Section 4.4.5). In a leader and follower task involving two robots, we show that the leader robot is able to detect faults in the follower robot (see Section 4.4.6). Although we demonstrate that the leader can detect exogenous faults, it is unlikely that the approach presented in this chapter can be used for exogenous fault detection in larger groups or swarms of robots. In Section 4.5, we discuss the cause of this limitation and we discuss various other aspects such as how to extend the approach to include fault identification, improve its scalability, and how fault detectors could be trained on data obtained in simulation. Finally, we conclude with a summary of the proposed method and the results obtained (see Section 4.6).

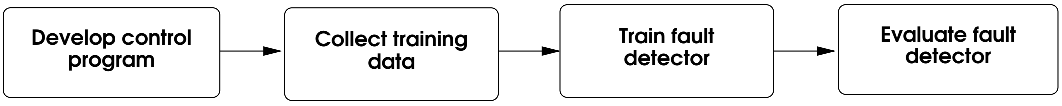


Figure 4.1: The four steps of our methodology for obtaining and evaluating fault detectors based on fault injection and learning.

4.1 Methodology

An overview of our method for obtaining and evaluating fault detectors based on fault injection and learning is illustrated in Figure 4.1. First the control program specifying the desired behavior for a fully operational robot is developed. We then collect training data during a phase in which the control program is run on real robotic hardware. During the training runs, sensory data and actuator control signals are collected while the robot is operating normally and after faults have been injected. We then train a fault detector on the data collected. Finally, the performance of the fault detector is evaluated.

The fault detection problem is to determine if the robot performs its task correctly, or if some fault in the hardware or in a software sub-system (but not a bug in the control program itself) is degrading the robot's performance. If a fault is detected, a signal can be sent to the control program itself, another robot, or a human operator. In our design, the fault detector is an isolated software component that passively monitors the performance of the robot through the information that flows in and out of the control program. A conceptual illustration of the relationship between the control program, the robotic platform, and the fault detection module can be seen in Figure 4.2.

The control program is run as a succession of sense-think-act cycles. In each cycle, the control program reads data from sensors such as the on-board camera, infrared proximity sensors, light sensors, and so on, processes the data, and sends control signals to the actuators such as the motors that drive the robot. A control cycle period is typically between 0.10 s and 0.15 s, depending on the task and the amount of computation needed to extract the relevant information from the sensory data.

Our hypothesis is that knowledge of the flow of sensory inputs describing the state of the world as perceived by the robot, and the resulting flow of control signals that steer the robot, are sufficient to discriminate between situations in which the robot operates normally and situations in which the presence of one or more faults hamper its performance. An intuitive motivation for why this is often true for autonomous robots is that a control program already requires a certain amount of continuously updated information about the state of the world to successfully steer the robot to perform its task in an unstructured environment. If we, for instance, design a control program to steer a robot between two arbitrary points A and B, we would need some sensory data telling us if we have left A and if we have reached B, perhaps also the current relative direction to B, proximity sensor readings for avoiding obstacles, and so on. If such information were not used, we would have to rely on a blind random walk and chance. Thus, the continuously updated information used by a control program in many cases contains some indications on how a task is progressing. We should therefore be able to determine if a robot

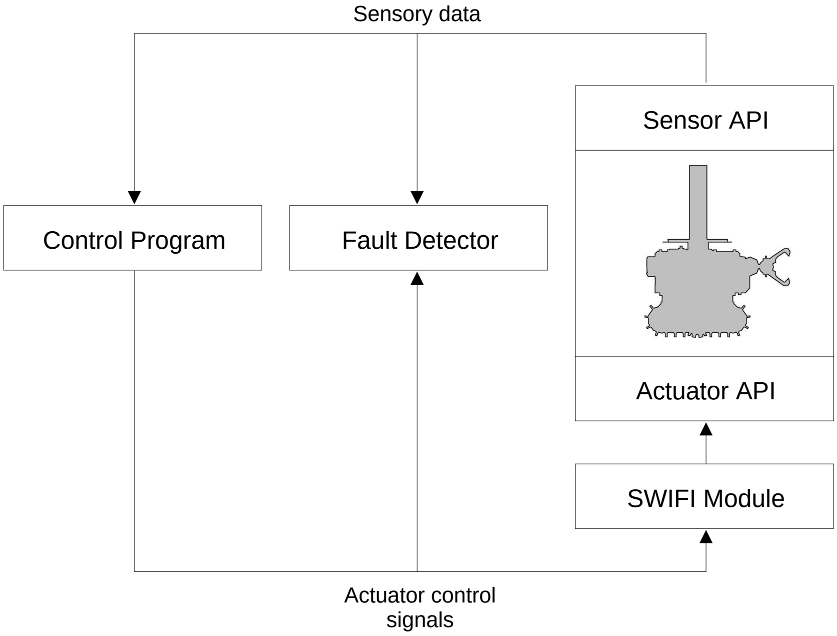


Figure 4.2: The fault detection module monitors the sensory data read by the control program and the consequent control signals sent to the actuators. The fault detection module is passive and does not interact with the robot hardware or the control program. The SWIFI Module facilitates fault injection (see text).

is operating normally or not based on the sensory data used by the control program.

The actions that a control program performs, that is, the signals which a control programs sends to the robot's actuators, can help us to determine if a robot has as fault. When a fault occurs, e.g. an actuator or a sensor breaks, the resulting behavior of the robot is an interaction between the fault and the control program. To illustrate this, assume that the robot in our example from above is propelled by a pair of differential drive wheels and with some means of sensing the relative direction θ to the point B. A simple way of steering the robot towards B would be to let the speeds of the left and right wheels, respectively, depend on θ in the following way, assuming that θ is in the interval $[-\pi, \pi]$ and that $x, k > 0$:

$$s_l = x - k\theta \tag{4.1}$$

$$s_r = x + k\theta \tag{4.2}$$

If the motor driving one of the wheels breaks causing the wheel to stop turning, then the robot will either turn continuously about the broken wheel if $x > k\pi$ or it will turn until $k|\theta| = x$ and then stop if $x \leq k\pi$. Hence, the subsequent flow of actuator control signals (and sensory inputs) depends not only on the type and location of the fault, but on the interplay between the control program and the fault.

If we take a white-box approach, we can use our knowledge of the control program and possible faults to build a fault detection module (for instance, if we know that $x \leq k\pi$ and if $k|\theta| = x$ for a number of consecutive control cycles, one of the wheels is probably broken). However, this approach does not scale well neither with the complexity of the original control program nor with the number of faults considered. Many interactions between a fault and the actions taken by a control program can also be significantly more complicated than our example above.

In our method, we take a black-box approach and consider only the inputs and the outputs of the control program, that is, the robot's flow of sensory data into the control program and the resulting flow of control signals sent from the control program to the actuators. Our hypothesis is that this information alone is sufficient to discriminate between situations in which the robot operates as normally and situations in which the presence of one or more faults hampers its performance. We record the flow of sensory data and control signals in situations where a robot is operating normally and where the robot is subject to a fault, respectively.

There are several methods for obtaining flows of sensory data and control signals for robots with faults: A broken robot can be used, readings can be obtained using a detailed software simulator, or faults can be purposefully provoked by the experimenter. In this study, we apply the latter technique: in a modified version of the on-board software, we provoke (simulated) hardware faults on real robots. We apply a well-established technique known as *software implemented fault injection* (SWIFI) used in dependable systems research. The technique is usually applied to measure the robustness and fault tolerance of software systems [Hsueh et al., 1997, Arlat et al., 1990]. In our case, we inject faults to discover how sensory data and the control signals change when faults are present. The idea is that by actively controlling the state of the robot (for instance by injecting faults or by using a broken robot) and recording the flow of sensory data and control signals, we can use supervised learning techniques and obtain a classifier that, based on that flow, can determine the state of the robot.

Some methods for fault detection base classification on the most recent observations only. The approach presented in this chapter allows classification based on both current and past observations, since many faults can only be detected if a system is observed over time. This is especially true for mechanical faults in mobile robots; a fault causing a wheel to block, for instance, can only be detected once the robot has tried to move the wheel for a period of time long enough for the absence of movement to be detectable. This period of time could be anywhere from a few milliseconds if, for example, dedicated torque sensors or encoders in the wheels are used, to several seconds if the presence of a fault has to be inferred based on information from non-dedicated sensors.

We use time delay neural networks (TDNNs) as classifiers [Waibel et al., 1989, Clouse et al., 1997]. TDNNs are feed-forward networks that allow reasoning based on time-varying inputs without the use of recurrent connections. In a TDNN, the values for a group of neurons are assigned based on a set of observations from a fixed distance into the past. The TDNNs used in this study are normal multilayer perceptrons for which the inputs are taken from multiple, equally spaced points in a delay-line of past observations. TDNNs have been extensively used for time-series prediction due to their ability to make predictions based on data distributed in time. A large variety of other classifiers exist[1] such as linear classifiers, Bayesian networks, hidden Markov models, support vector machines, and so on. The main reason why we chose artificial neural networks in this study is that this type of classifier is often used in robotics and generally enough for our concerns.

---

[1] For introductions to these topics see [Duda et al., 2000, Devroye et al., 1996, Jensen, 1996, Rabiner, 1989, Cristianini and Shawe-Taylor, 2000].

4.1.1 Formal Definitions

Our aim is to obtain a function that maps from a set of current and past sensory data, S, and control signals, A, to either 0 or 1 corresponding to *no-fault* and *fault*, respectively:

$$\chi : S, A \rightarrow \{0, 1\} \tag{4.3}$$

We assume that such a function exists and we approximate it with a feed-forward neural network. We let $I \subseteq (S \cup A)$ be the inputs to the network. We choose a network that has a single output neuron whose output value is in the interval $[0, 1]$. The output value is interpreted in a task-dependent way. For instance, a threshold-based classification scheme can be applied where an output value above a given threshold is interpreted as 1 (*fault*), whereas an output value below the threshold is interpreted as 0 (*no-fault*).

Sensory Data, Control Signals and Fault State: We perform a number of runs each consisting of a number of control cycles (sense-compute-act loops). For each control cycle, c, we record the sensory data and control signals to and from the control program. We let i_c^r denote a single set of control program inputs and outputs (CPIO), that is, the CPIO for control cycle c in run r. We let s denote the number of values in a single CPIO set, that is $s = |i_c^r|$. We let I^r be the ordered set of all CPIO sets for r. Similarly, for each control cycle we let f_c^r denote the fault state for control cycle c in run r, where $f_c^r = 1$ if a fault has been injected and 0 otherwise. Hence, $f_c^r = 0$ when the robot is operating normally and $f_c^r = 1$ otherwise.

Tapped Delay-Line and Input Group Distance: The CPIO sets are stored in a tapped delay-line, where each tap has size s. The input layer of a TDNN is logically organized in a number of *input groups* $g_0, g_1, \ldots, g_{n-1}$ and each group consists of precisely s neurons, that is, one neuron for each value in a CPIO set. The activation of the input neurons in group g_t are then set according to $g_t = i_{c-t \cdot d}^r$, where c is the current control cycle and d is the *input group distance*. See Figure 4.3 for an example. If we choose an input group distance $d = 1$, for example, the TDNN has access to the current and the $n - 1$ most recent CPIOs, whereas if $d = 2$, the TDNN has access to the current and every other CPIO set up to $2(n - 1)$ control cycles into the past, and so on. In this way, the input group distance specifies the relative distance in time between the individual groups and (along with the number of groups) how far into the past a TDNN "sees".

TDNN Structure and Activation Function: The input layer of the TDNN is fully connected to a hidden layer, which is again fully connected to the output layer. The output layer consists of a single neuron whose value reflects the network's classification of the current inputs. The activations of the neurons are computed layer-by-layer in a feed-forward

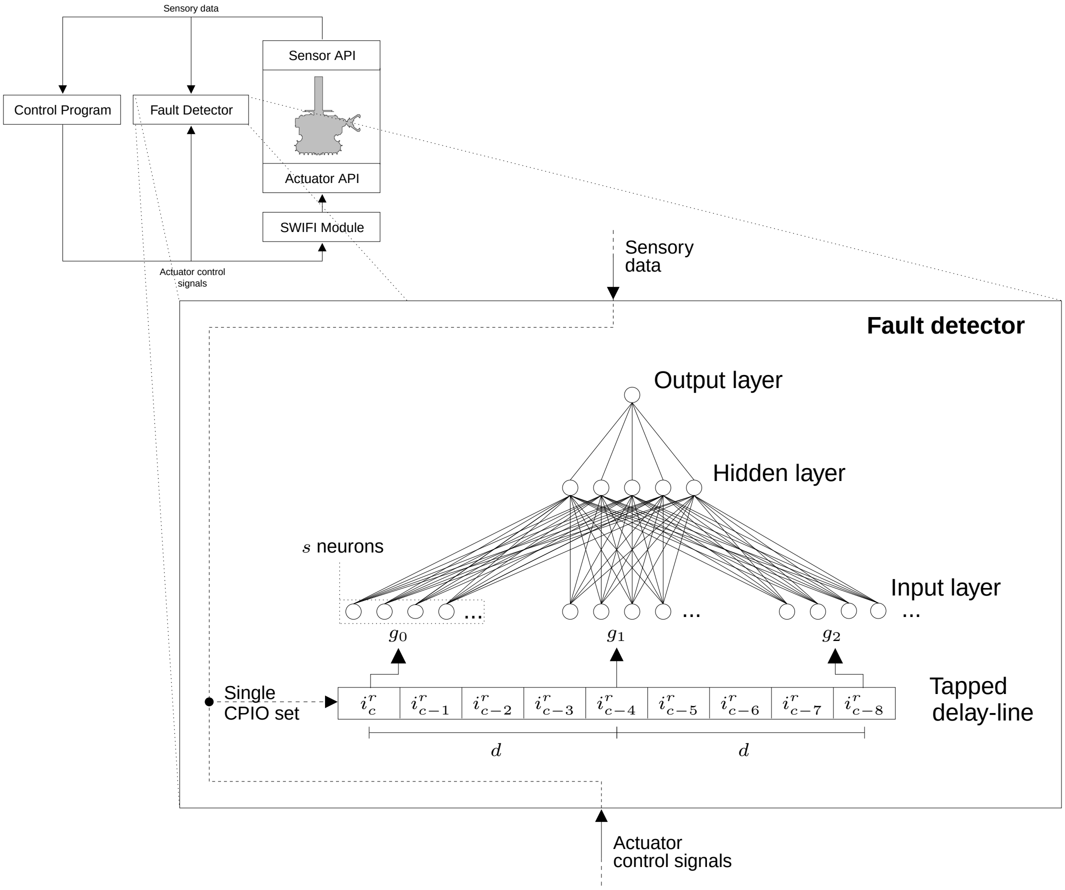

Sensory data
Sensor API
Control Program
Fault Detector
Actuator API
SWIFI Module
Actuator control signals
Sensory data
Fault detector
Output layer
Hidden layer
s neurons
Input layer
g_0
g_1
g_2
Single CPIO set
i^r_c
i^r_{c-1}
i^r_{c-2}
i^r_{c-3}
i^r_{c-4}
i^r_{c-5}
i^r_{c-6}
i^r_{c-7}
i^r_{c-8}
Tapped delay-line
d
d
Actuator control signals

Figure 4.3: An illustration of a fault detection module based on a TDNN. The current control program input and output (CPIO) is stored in the tapped delay-line and the activations of the neurons in the logical input groups are set according to the current and past CPIOs. In the example illustrated, there are 3 input groups and the input group distance d is 4.

manner and the following sigmoid activation function is used to compute the neurons' outputs in the hidden and the output layers:

$$f(a) = \frac{1}{1 + e^{-a}}, \tag{4.4}$$

where a is the activation of the neuron.

Classification and Learning: The output of the TDNN has a value between 0 and 1. The error factor used in the back-propagation algorithm is computed as the difference between the fault state f_c^r and the output o_c:

$$E_c = f_c^r - o_c. \tag{4.5}$$

The neural networks are all trained by a standard batch learning back-propagation algorithm to minimize the absolute value of the error factor E_c in (4.5) [Rumelhart et al., 1986].

In summary, sensor and actuator data is collected from a number of runs on real robots and different types of faults are injected. A TDNN is trained to discriminate between normal and faulty operation. By storing past observations in a tapped delay-line, the TDNN can base classification on how the flow of information changes over time.

4.1.2 Faults

Faults in the mechanical system that propels the robot can be hard to detect when no special hardware to facilitate fault detection is used. Unlike faults in sensors, which are usually immediately observable due to inconsistencies or abrupt changes in the sensory values, faults in the mechanical system have to be inferred from the unexpected consequences (due to the presence of faults) of the actions performed by the robot. There is often a latency associated with the detection of faults in the mechanical systems of a robot because the consequences of the robot's actions need to become apparent before the presence of a fault can be inferred.

We focus principally on faults in the mechanical system that propels the *s-bots*. This system consists of a set of differential *treels*, that is, combined tracks and wheels (see Chapter 3). Given that the treels contain moving parts and that they are used continuously in most experiments, they are the components in which the majority of faults occur.

We analyze two types of faults. Both types can either be isolated to the left or the right treel or they can affect both treels simultaneously. The first type of fault causes one or both treels to stop moving. This usually happens if the strap that transfers power from the electrical motors to the treels breaks or jumps out of place. Whenever this happens, the treels stop moving. We denote this type of fault as *stuck-at-zero*. The second type of fault occurs when an *s-bot's* software sub-system crashes leaving one (or both) motor(s) driving the treels running at some undefined speed. The result is that a treel no longer can be controlled by the on-board software. We refer to this type of fault as *stuck-at-constant*.

To collect training data, a number of runs are conducted. In each run, the *s-bot* starts in a nominal state and during the run, a fault is injected. The fault is implemented in the on-board software by discarding the control program's commands to the failed part and by substituting them according to the type of fault injected. If, for instance, a *stuck-at-constant* fault is injected in the left treel, the speed of that treel is set to a random value, and all future changes in speed requested by the control program are discarded.

4.1.3 Software Architecture

The software architecture has been designed in a way that allows faults to be injected and a fault detector to run without making any changes to an existing controller. In order to take advantage of our fault detection approach, an existing control program only needs to incorporate code for accommodating detected faults. This is made possible by a layered architecture that makes extensive use of the Common Interface (see page 11). One layer allows training data to be collected and allows a fault detector to passively monitor the flow of sensory data and actuator control signals. Another layer implements the fault injection and simulation logic. The software architecture is detailed in Appendix A.1.

4.2 The Three Experimental Setups

We have chosen three setups in which to study fault detection based on fault injection and learning. The setups are called *find perimeter*, *follow the leader*, and *connect to s-bot*, respectively, and they are described in Figure 4.4. In all setups, we use a 180x180 cm^2 arena surrounded by walls.

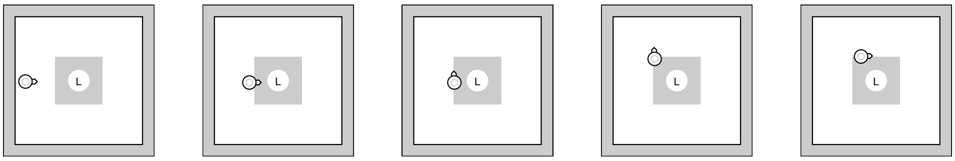
In the *find perimeter* setup, an *s-bot* follows the perimeter of a dark square drawn on the arena surface. In this setup, the four infrared ground sensors are used to discriminate between the normal light-colored arena surface and the dark square.

In the *follow the leader* setup, an *s-bot* (the *leader*) performs a random walk in the environment and another *s-bot* (the *follower*) follows. The two robots perceive one another using their omnidirectional cameras. The infrared proximity sensors are used to detect and avoid walls. Objects up to 50 cm away can be seen reliably through the camera. Infrared proximity sensors have a range from a few centimeters up to 20 cm depending on the reflective properties of the obstacle or object. Faults are injected in the *follower* only.

In the *connect to s-bot* setup, one *s-bot* tries to connect to another, stationary *s-bot*. The connection is made using the gripper. The connecting *s-bot* uses the camera to perceive the location of the other robot. Faults are only injected in the *s-bot* that is trying to form the connection.

Readings from sensors such as infrared ground sensors are straightforward to normalize and feed to the input neurons of a neural network. The camera sensor, in contrast, captures 640x480 color images. For these more complex sensor readings to serve as input to a neural network, relevant information must be extracted and processed beforehand. As discussed in Chapter 3, the *s-bots* have sufficient on-board processing power to scan entire images and identify objects

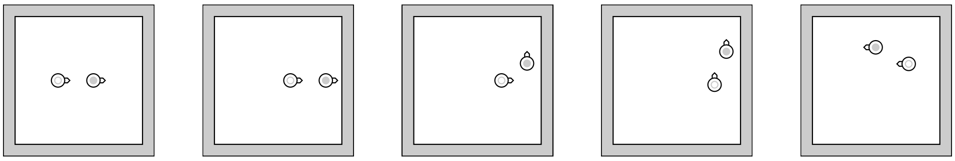
Find perimeter: An *s-bot* is situated in an arena with a dark colored square drawn on an otherwise light floor. A light source is placed in the center of the square. The task for the *s-bot* is to follow the perimeter of the square.



(1) (2) (3) (4) (5)

Sensors: IR ground (4 inputs), light (8 inputs)
Control period: 100 ms

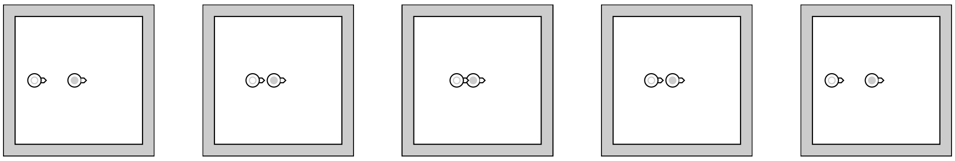
Follow the leader: Two *s-bots* are placed in a square environment bounded by walls. One of the *s-bots* has been preassigned the *leader* role, while the other has been preassigned the *follower* role. The *leader* moves around in the environment. The *follower* tails the *leader* and tries to stay at a distance of 35 cm. If the *follower* falls behind, the *leader* waits. Faults are injected in the *follower* only.



(1) (2) (3) (4) (5)

Sensors: Camera (16 inputs), IR proximity (15 inputs)
Control period: 150 ms

Connect to s-bot: One *s-bot* attempts to physically connect to a stationary *s-bot* using its gripper. When a successful connection has been made, the *s-bot* waits for 10 seconds, disconnects, moves back, and tries to connect again. Faults are injected in the connecting *s-bot* only.



(1) (2) (3) (4) (5)

Sensors: Camera (16 inputs), optical sensors in gripper (4 inputs)
Control period: 150 ms

Figure 4.4: Description of the three setups: *find perimeter*, *follow the leader*, and *connect to s-bot*. For each setup a list of sensors used and the control cycle period for the controllers are shown. The number in brackets after each sensor listed corresponds to the number of input values the sensor provides to the fault detector at each control cycle.

based on color information. The image processor is configured to detect the location of colored LEDs of the *s-bots* only, and discard any other information. The *s-bot* camera captures images of the robot's surroundings reflected in a hemispherical mirror. Since the robots operate on flat terrain, the distance in pixels from the center of an image to a perceived object corresponds to the physical distance between the robot and the object. In order to make this information available to a neural network, we divide the image into 16 non-overlapping slices of equal size in terms of the field of view they cover. Each slice corresponds to a single input value. The value is set depending on distance to the closest object perceived in the slice. If no object is perceived, the value for a slice is 0. Used in this way, the camera sensor effectively becomes a range sensor for colored LEDs.

4.3 Data Collection, Training and Performance Evaluation

4.3.1 Data Collection

A total of 60 runs on real *s-bots* are performed for each of the three setups. In each run, the robot(s) start in a nominal state, and at some point during the run a fault is injected. The fault is injected at a random point in time after the first 5 seconds of the run and before the last 5 seconds of the run according to a uniform distribution. Hence, a robot spends on average 50% of the time that a run lasts in a nominal state. When a fault is injected, there is a 50% probability that a fault affects both treels instead of only one of the treels, and faults of the type *stuck-at-zero* and *stuck-at-constant* are equally likely to occur. Each run consists of 1000 control cycles and for each control cycle the sensory data, control signals, and the current fault state are recorded. In the *find perimeter* setup, 1000 cycles correspond to 100 seconds, while for the *follow the leader* and in the *connect to s-bot* setups 1000 cycles correspond to 150 seconds, due to the longer control cycle period.

4.3.2 Training and Evaluation Data

The data sets recorded in each setup are partitioned into two subsets, one consisting of data from 40 runs, which is used for training; and one consisting of the data from the remaining 20 runs, which is used for a final performance evaluation. The TDNNs all have a hidden layer of 5 neurons and an input layer with 10 input groups.

4.3.3 Performance Evaluation

The performance of the trained neural networks is computed based on the 20 runs in each setup reserved for evaluation. A network is evaluated on data from one run at a time, and the output of the network is recorded and compared to the fault state.

The two main performance criteria for a fault detection approach are reliability of detection and speed of detection. In our approach, the interpretation of the output of the trained neural network has an important impact on both criteria. The simplest interpretation mechanism is to define a threshold. An output value above this threshold is considered an indication that a fault is present, whereas an output value below the threshold is considered an indication that

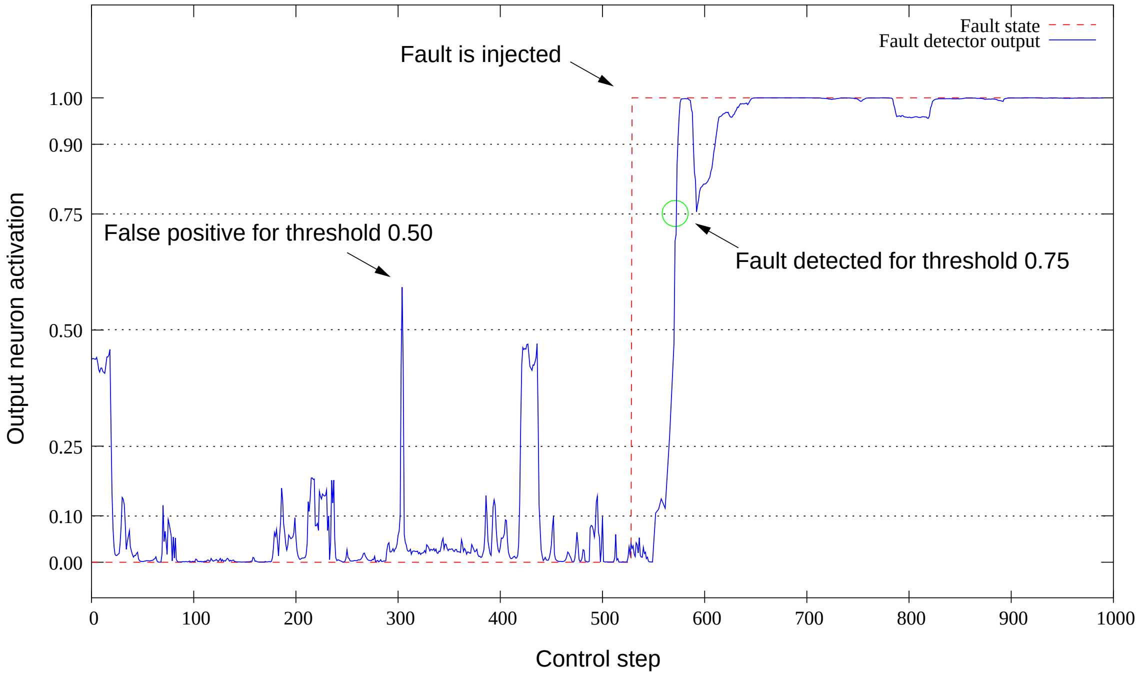


Figure 4.5: An example of the output of a trained TDNN during a run. The dotted line shows the optimal output. At control cycle 529 a fault is injected. Five different thresholds are indicated, 0.10, 0.25, 0.50, 0.75, and 0.90, and a false positive for threshold 0.50 is shown at control cycle 304 (the output has a value greater than 0.50 *before* the fault was injected at control cycle 529). The latency for a threshold is the number of control cycles from the moment the fault is injected till the moment the output value of the TDNN becomes greater than the threshold. In the example above, the latency for threshold 0.75 is 43 control cycles because the output of the TDNN reaches 0.75 only at control cycle 562, that is, 43 control cycles after the fault was injected.

the robot is in a nominal state. In the next section, we present results for five such thresholds: 0.10, 0.25, 0.50, 0.75, and 0.90.

A graphical representation of TDNN's output during an evaluation run is shown in Figure 4.5. In the run shown, a fault was injected at control cycle 529. The number of false positives is the number of control cycles before a fault is injected for which the output of the TDNN exceeds the given threshold. Choosing a threshold of 0.50 would, for example, result in one false positive, since the output of the network is higher than 0.50 for one control cycle (cycle 304) *before* the fault was injected. If we choose a higher threshold, either 0.75 or 0.90, false positives are avoided. However, choosing a higher threshold has a negative impact on another aspect of a fault detector's performance, namely its *latency*. Latency is the number of cycles between the occurrence and detection of a fault. In the example in Figure 4.5, the fault is detected at control cycle 553, 561, 570, 572, and 574 for the thresholds 0.10, 0.25, 0.50, 0.75, and 0.90, yielding latencies of 24, 32, 41, 43, 45 control cycles, respectively.[2]

For some tasks, the recovery procedure is costly, and fewer false positives might be desirable even at the cost of a higher latency. For other tasks, undetected faults can have serious consequences and a low latency is more important than reducing the number of false positives. We can gain fine control over the balance between latency and number of false positives by choosing an appropriate threshold.

4.4 Results

We first evaluate the effect of the input group distance on the performance of a fault detector with respect to its latency and number of false positives (Section 4.4.1). The input group distance determines how far into the past a fault detector "sees". We then evaluate the performance of the fault detectors in the *follow the leader* and *connect to s-bot* setups (Section 4.4.2). In some situations, false positives can be costly and we show how the output of a TDNN can be reinterpreted to avoid nearly all false positives (Section 4.4.3). We test if the proposed method is applicable when faults in both sensors and actuators are considered (Section 4.4.4). We show that it is possible to obtain a fault detector that can generalize if the task varies between runs (Section 4.4.5). Finally, we demonstrate fault detection through fault injection and learning can be extended to *exogenous fault detection*, that is, the capacity of one robot to detect faults in another, physically separate robot.

4.4.1 Tuning the Input Group Distance

To find an input group distance that performs well, we trained fault detectors with input group distances ranging from 1 to 10. Figure 4.6 and Figure 4.7 show respectively a box-plot of the latencies[3] and a box-plot of the number of false positives observed during 20 evaluation runs

---

[2] It is important to note that a latency of 24 control cycles may seem long, but the faults that we are trying to detect do not always have an immediate impact on the performance of a robot. If, for instance, the fault injected causes a treel to block (a fault of the type *stuck-at-zero*), the fault can only be detected if the control program tries to set the treel to a non-zero speed. In particular, if the control program is setting low speeds (values close to zero) it might take a long time before a fault can be detected.

[3] For the latency results, we only include data from runs in which the fault was detected. See Table 4.3 for the number of undetected faults for different input group distances and thresholds.

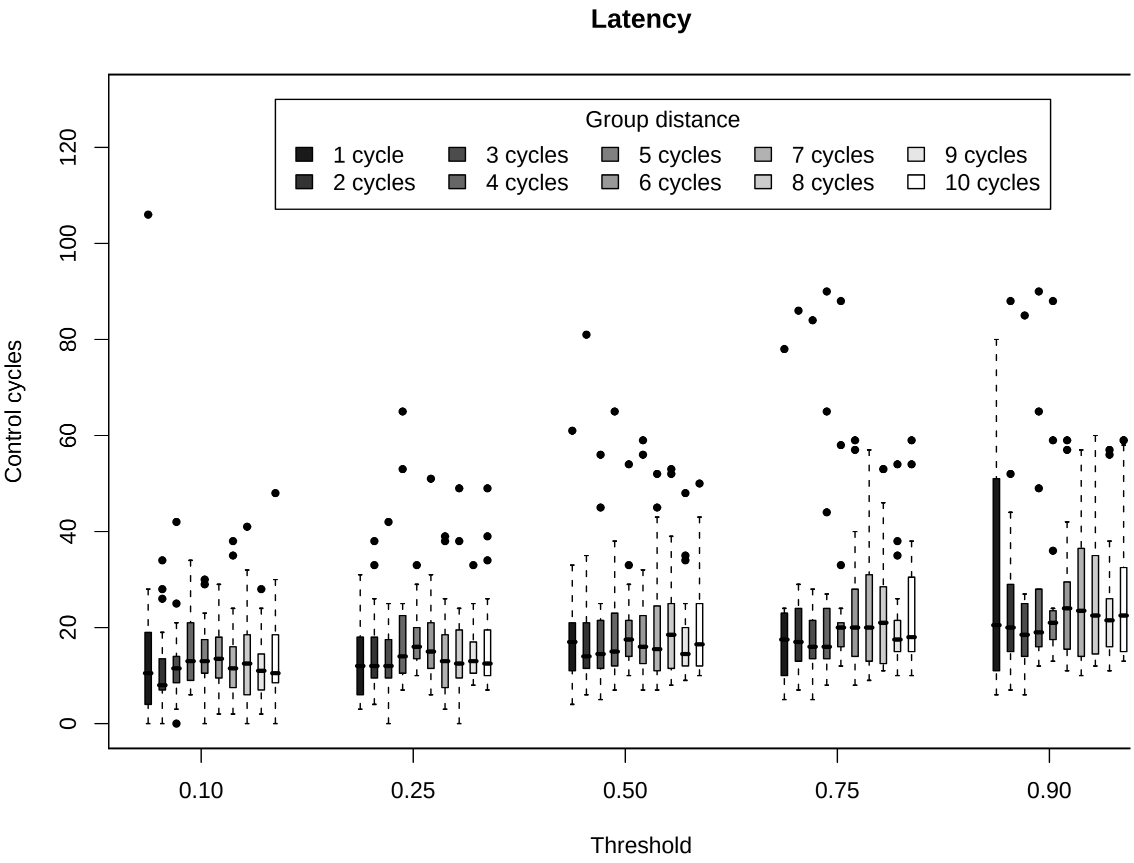


Figure 4.6: Box-plot of the latencies observed in 20 evaluation runs in the *find perimeter* setup using fault detectors with input group distances from 1 to 10. Results are shown for the thresholds 0.10, 0.25, 0.50, 0.75, and 0.90. Each box comprises observations ranging from the first to the third quartile. The median is indicated by a horizontal bar, dividing the box into the upper and lower part. The whiskers extend to the farthest data points that are within 1.5 times the interquartile range. Outliers are shown as dots. The results show that the input group distance does not have a major influence on the latency of a fault detector, while larger thresholds yield longer latencies.

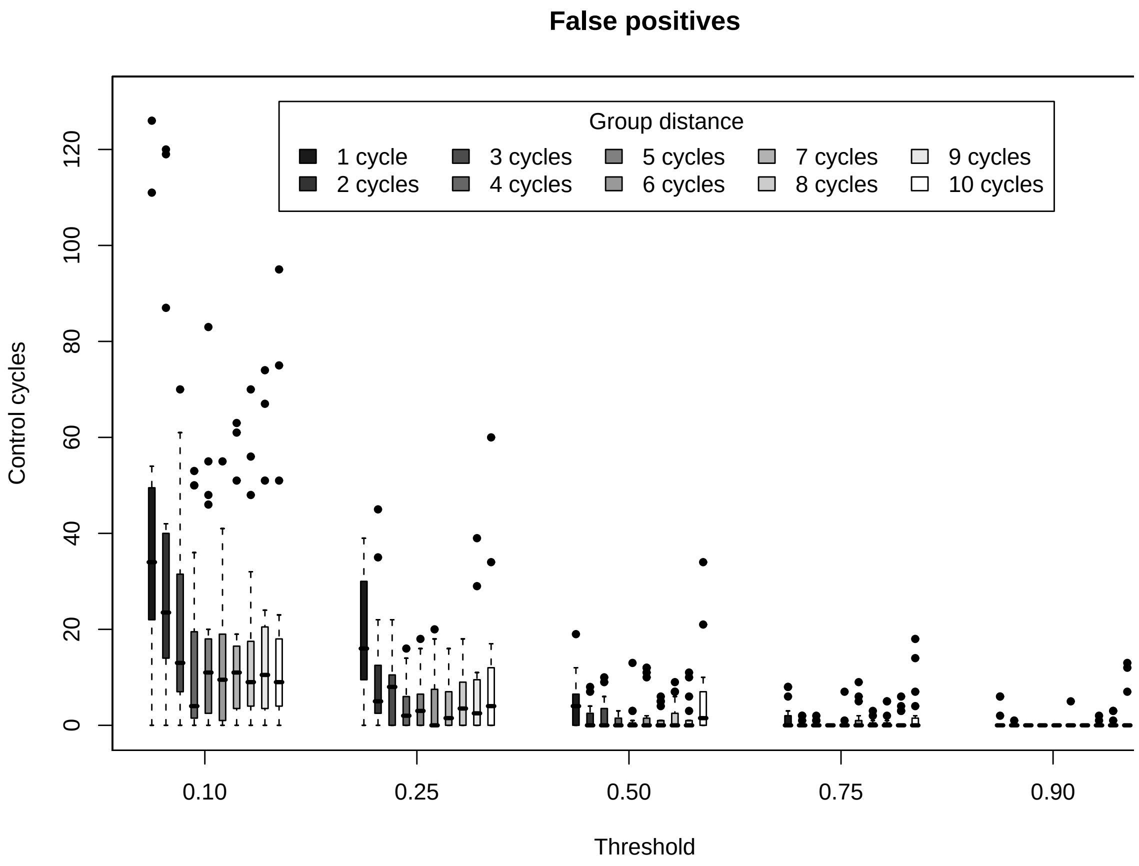


Figure 4.7: Box-plot of the number of false positives observed in 20 evaluation runs in the *find perimeter* setup using fault detectors with input group distances from 1 to 10. Results are shown for the thresholds 0.10, 0.25, 0.50, 0.75, and 0.90. For low input group distances, 1 and 2 in particular, the fault detector in general detects a large number of false positives, while no clear trend is observed for fault detectors with input group distances above 4. See the caption of Figure 4.6 for details on box-plots.

Table 4.1: Median latencies during 20 evaluation runs in the *find perimeter* setup with fault detectors using input groups distances from 1 to 10 and for the thresholds: 0.10, 0.25, 0.50, 0.75, and 0.90.

		Threshold				
		0.10	0.25	0.50	0.75	0.90
Input group distance	1	10.5	12.0	17.0	17.5	20.5
	2	8.0	12.0	14.0	17.0	20.0
	3	11.5	12.0	14.5	16.0	18.5
	4	13.0	14.0	15.0	16.0	19.0
	5	13.0	16.0	17.5	20.0	21.0
	6	13.5	15.0	16.0	20.0	24.0
	7	11.5	13.0	15.5	20.0	23.5
	8	12.5	12.5	18.5	21.0	22.5
	9	11.0	13.0	14.5	17.5	21.5
	10	10.5	12.5	16.5	18.0	22.5

in the *find perimeter* setup. Results are shown for the five thresholds: 0.10, 0.25, 0.50, 0.75, and 0.90. The median latencies and number of false positives for each configuration of input group distance and threshold are summarized in Table 4.1 and Table 4.2, respectively.

The latency results in Figure 4.6 show no clear correlation between latency and input group distance. The false positive results in Figure 4.7, however, show that for low input group distances, 1 and 2 in particular, the fault detector in general detects a large number of false positives. No clear trend is observed regarding the number of false positives for fault detectors with input group distances above 4.

An input group distance of 1 means that the TDNN is provided with data from the past 10 control cycles (because there are 10 input groups). 10 control cycles are equal to 1 second in the *find perimeter* setup. Similarly, an input group distance of 2 means that the TDNN is provided with data from the past 2 seconds, but only from every other control cycle. The false positive results indicate that data from a period longer than 2 seconds (i.e., an input group distance higher than 2) is needed for accurate classification.

The results in Figure 4.6 and Figure 4.7 show that the performance of the fault detectors, both in terms of latency and in terms of the number of fault positives, is clearly affected by the choice of the classification threshold: the lower the threshold, the lower the latency of the fault detector and the more false positives are observed. For the fault detector with an input group distance of 5, for instance, the median latency is 13 control cycles when a threshold of 0.10 is used, whereas the median latency is 21 when a threshold of 0.90 is used. For the same fault detector, the median number of fault positives is 11 if a threshold of 0.10 is used, while no false positives are observed when a threshold of 0.90 is used.

In a few cases, a fault is never detected. Undetected fault occur when a TDNNs output never exceeds the chosen threshold after a fault has been injected. The number of undetected faults for different thresholds and input group distances is shown in Table 4.3. All undetected faults were observed when low input group distances were used.

Table 4.2: Median number of false positives observed during 20 evaluation runs in the *find perimeter* setup with fault detectors using input groups distances from 1 to 10 and for the thresholds: 0.10, 0.25, 0.50, 0.75, and 0.90.

		Threshold				
		0.10	0.25	0.50	0.75	0.90
Input group distance	1	34.0	16.0	4.0	0.0	0.0
	2	23.5	5.0	0.0	0.0	0.0
	3	13.0	8.0	0.0	0.0	0.0
	4	4.0	2.0	0.0	0.0	0.0
	5	11.0	3.0	0.0	0.0	0.0
	6	9.5	0.0	0.0	0.0	0.0
	7	11.0	1.5	0.0	0.0	0.0
	8	9.0	3.5	0.0	0.0	0.0
	9	10.5	2.5	0.0	0.0	0.0
	10	9.0	4.0	1.5	0.0	0.0

Table 4.3: Number of undetected faults observed during 20 evaluation runs in the *find perimeter* setup, for five different thresholds, using input group distances from 1 to 10.

		Threshold				
		0.10	0.25	0.50	0.75	0.90
Input group distance	1	0	1	1	2	2
	2	1	1	1	3	3
	3	0	0	0	1	3
	4	0	0	1	1	2
	5	0	0	0	1	1
	6	0	0	0	0	0
	7	0	0	0	0	0
	8	0	0	0	0	0
	9	0	0	0	0	0
	10	0	0	0	0	0

In the other two setups, *follow the leader* and *connect to s-bot*, we did a similar study of the effect of the input group distance and the performance of the fault detectors. We found that an input group distance of 5 performed well in all setups. The experiments that we present in the following sections are all, therefore, conducted using an input group distance of 5. Since we use TDNNs with 10 input groups, an input group distance of 5 means that a TDNN can see $4.5\ \mathrm{s}$ into the past in the *find perimeter* setup, in which the control period is $0.10\ \mathrm{s}$. TDNNs in the *follow the leader* and *connect to s-bot* setups see $6.75\ \mathrm{s}$ into the past since the controllers in these setups run with a control period of $0.15\ \mathrm{s}$.

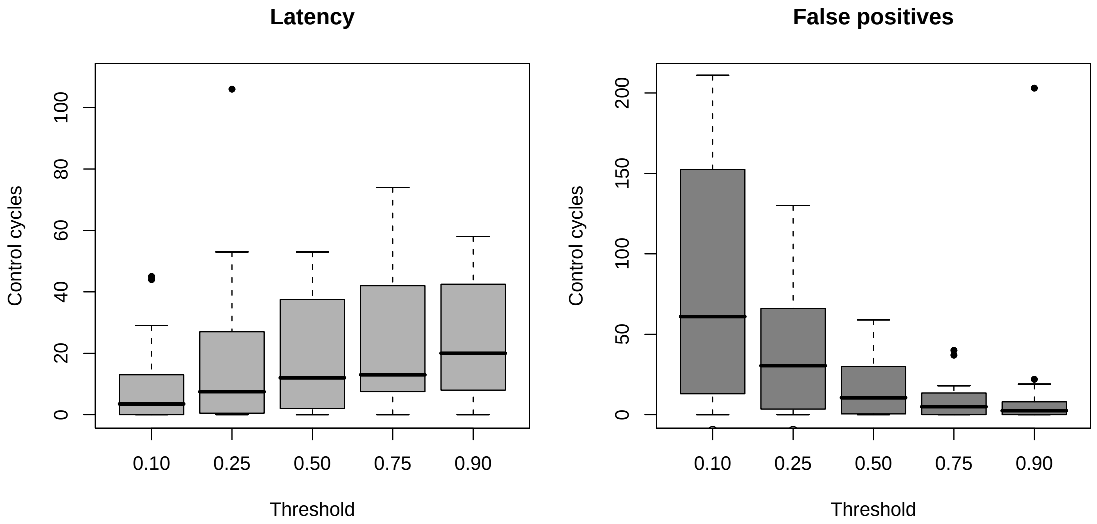


Figure 4.8: Box-plot of the latencies and number of false positives observed during 20 evaluation runs in the *follow the leader* setup for different thresholds and an input group distance of 5. See the caption of Figure 4.6 for details on box-plots.

4.4.2 Fault Detection Performance in the *follow the leader* and *connect to s-bot* setups

We trained a fault detector to detect faults in the *follow the leader* setup and another fault detector to detect faults in the *connect to s-bot* setup. Box-plots of the false positives and the latency results observed in 20 evaluation runs in the *follow the leader* and *connect to s-bot* setups are shown in Figure 4.8 and Figure 4.9, respectively. In both setups, the fault detector was configured to use an input group distance of 5. The number of undetected faults observed during the evaluation runs in the two setups is shown in Table 4.4. Two interesting tendencies can be seen: The number of false positives observed in the *follow the leader* setup is comparatively high, while in the *connect to s-bot* the observed latencies are high when compared with the results obtained in the two other setups. In the *follow the leader* setup, there are two robots moving around and the fault detector for the *follower*, in which faults were injected, has to infer the presence of faults based on its interactions with the *leader*. Misclassification of the *follower*'s state can occur in situations where, for instance, the *leader* and the *follower* are moving at constant speeds in a given direction. In these cases, the *follower* receives sensory data similar to those in situations where both its treels are *stuck-at-zero*: The *leader* waits for the *follower*, but due to the fault, the *follower* does not move. The fact that the control program (and therefore the fault detector) depends on a dynamic feature of the environment (the *leader*) seems to complicate the classification of the robot's state. However, the performance of the fault detector is still quite good, especially considering that the *leader* is often the *only* object perceivable by the *follower* (the proximity sensors will only sense the presence of walls at distances lower than approximately $10\ \mathrm{cm}$).

The comparatively high latencies observed in the *connect to s-bot* setup are similarly due to a task-dependent feature: After a successful connection has been made, the connecting robot

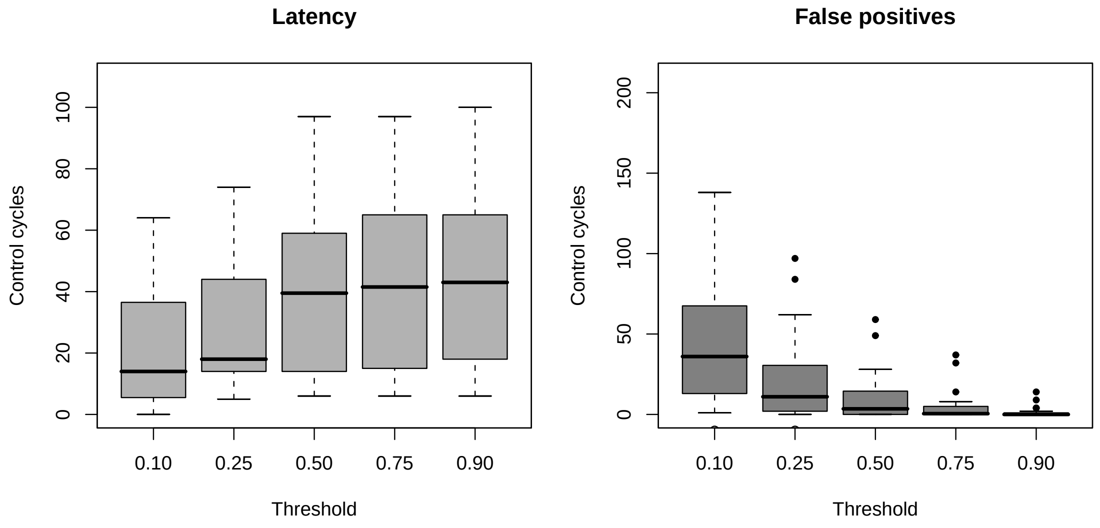


Figure 4.9: Box-plot of the latencies and number of false positives observed during 20 evaluation runs in the *connect to s-bot* setup, for different thresholds and an input group distance of 5. See the caption of Figure 4.6 for details on box-plots.

Table 4.4: Number of undetected faults observed during 20 evaluation runs in the *follow the leader* and *connect to s-bot* setups, for different thresholds, using an input group distance of 5.

	Threshold				
	0.10	0.25	0.50	0.75	0.90
Follow the leader	0	0	0	0	0
Connect to s-bot	1	2	2	2	3

waits for 10 seconds before disconnecting, moving back, and attempting to form the connection anew. During the waiting period it is not possible to detect if a fault has occurred in the treels or not. Even if a *stuck-at-constant* fault is injected, causing one or both treels to be assigned a random and non-changeable speed, the outcome is the same: The robot does not move because it is physically connected to the other robot. Thus, it can take longer to detect a fault due to these particular situations in which a fault does not have an effect on the behavior of the robot.

4.4.3 Reducing the Number of False Positives

The simplest way to interpret the output of a TDNN trained to detect faults is to compare the value of the output neuron against a threshold. Values above the threshold are interpreted as evidence of a fault whereas values below the threshold mean that no fault is detected. The fault detectors presented so far follow this simple classification scheme and results have been presented for five thresholds: 0.10, 0.25, 0.50, 0.75, and 0.90.

For many robotic tasks, a latency of a few seconds does not represent a risk: as long as a fault is eventually detected, the robot is able to communicate this to a human operator or to other robots, who can then take the necessary steps to ensure that the task is progressed.

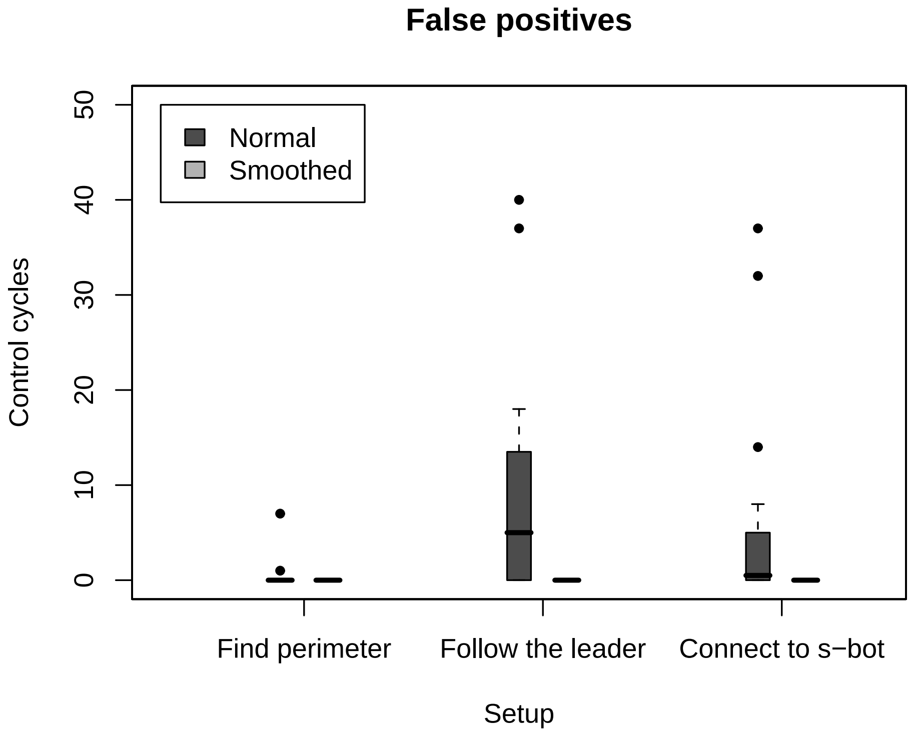


Figure 4.10: Box-plot of false positives results observed in 20 runs in each of the three setups using fault detectors in which the output of the TDNN is used directly and fault detectors in which the output is smoothed by computing the moving average over 25 control cycles. A threshold of 0.75 was used for all fault detectors. False positives were only observed during one run in the *follow the leader* setup when the TDNN's output was smoothed. The run is not shown in the figure since it is out of scale (164 false positives were detected during this run). See the caption of Figure 4.6 for details on box-plots.

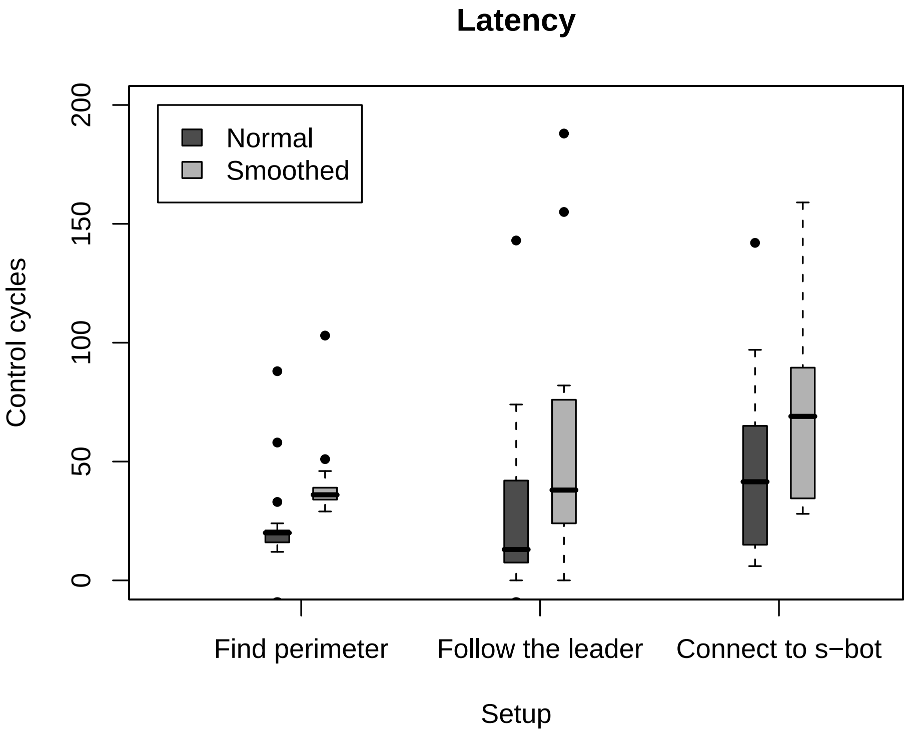


Figure 4.11: Box-plot of latency results observed in 20 runs in each of the three setups using fault detectors in which the output of the TDNN is used directly and fault detectors in which the output is smoothed by computing the moving average over 25 control cycles. A threshold of 0.75 was used for all fault detectors. See the caption of Figure 4.6 for details on box-plots.

Accommodating a fault, on the other hand, is usually expensive, as other robots need to take action or a human operator needs to evaluate and solve the situation. Frequent false positives, therefore, are likely to have a negative impact on the performance.

One way of reducing the number of false positives is to choose a high threshold, e.g. 0.90, which results in fewer false positives than lower thresholds (see for instance Figure 4.9). Many of the false positives observed occur for a single or few consecutive control cycles only (like in the example in Figure 4.5). This suggests an alternative way of reducing the number of false positives: to smooth the output of the trained neural networks in order to remove spikes of false positives. We do this by computing the moving average of a trained TDNN's output value and basing the classification on this moving average rather than on the TDNN's output directly. We configured the fault detectors to use a moving average over 25 control cycles of the TDNN's output and a threshold of 0.75. Figure 4.10 and 4.11 show respectively the number of false positives and the latencies observed in 20 evaluation runs for each task.

By computing the moving average and thereby smoothing the output of the TDNN, we almost completely eliminate false positives. As the results in Figure 4.11 show, however, this is at the cost of a higher latency. Since the moving average increases latency, it can result in more undetected faults as more runs finish before faults are detected. In the *find perimeter* setup, two faults were not detected when averaging the output over 25 control cycles, compared to only 1 when averaging was not used. Similarly, in the *connect to s-bot* setup, 5 faults were not detected when a moving average was used, compared to 2 when the output of the TDNN was used directly. In the *follow the leader* setup, all faults were detected in both cases.

4.4.4 Faults in Both Sensors and Actuators

Possible faults are not limited to the mechanical systems that propel robots; other hardware, such as sensors, can also fail. In this section, we demonstrate that our approach is equally applicable to faults in the sensors. We also show that a single appropriately trained fault detector is capable of detecting faults in both sensors and actuators. We first evaluate our approach when only faults in sensors are considered. We then go on to evaluate the approach when faults in both sensors and the treels are considered. All experiments are conducted in the *find perimeter* setup.

We conducted a set of runs in which we injected faults in the front infrared ground sensor, that is, the infrared ground sensor located closest to the gripper (see Figure 3.1). We conducted another set of runs in which we injected faults in the first and second light sensor counter-clockwise from the gripper when an *s-bot* is seen from above (see Figure 3.1).[4] We trained a TDNN with an input group distance of 5 on 40 runs: 20 runs during which a fault was injected in the front ground sensor and another 20 runs during which a fault was injected in the light sensors. The fault detector was evaluated on 20 runs: 10 runs in which a fault was injected in the ground sensor and another 10 runs in which a fault was injected in the light sensors.

We performed a set of experiments to determine if a single fault detector can be trained to detect faults in both the sensors and actuators. We trained a fault detector on a training set

---

[4] We initially tried to inject faults in the first light sensor only, but a fault in a single light sensor had no effect on the performance of the robot: with seven out of eight light sensors working the robot still completed the task. We therefore injected faults in both the first and the second light sensor.

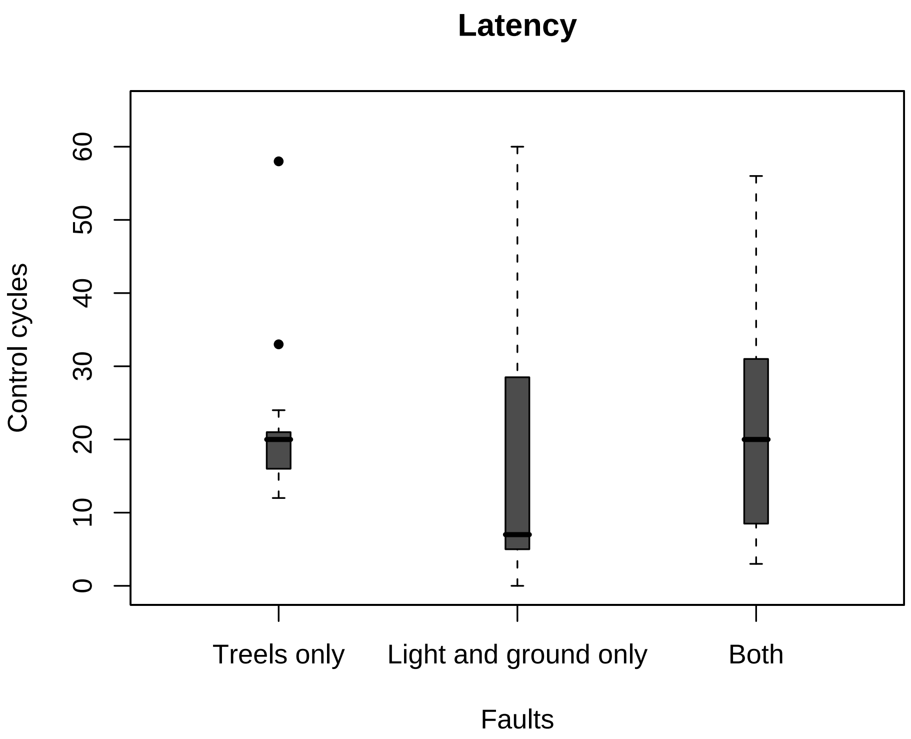


Figure 4.12: Box-plot of latency results for fault detectors trained to detect faults in the treels only (from Section 4.4.1), in the ground and light sensors only, and a fault detector trained to detect faults in both the ground and light sensors and the treels. In each case, the fault detector was evaluated on 20 runs in which faults corresponding to those the fault detector was trained to detect were injected. All three fault detectors were configured to use the output of the TDNN directly and to use a threshold of 0.75. See the caption of Figure 4.6 for details on box-plots.

consisting of 40 runs: 20 runs in which a fault was injected in either one or both treels and 20 runs in which a fault was injected in either the ground sensor or light sensors. The fault detector was evaluated on 20 runs: 10 runs in which a fault was injected in either the ground sensor or in the light sensors, and another 10 runs in which a fault was injected in either one of the treels or in both treels.

Figure 4.12 shows the latencies observed for the fault detector trained to detect faults in the sensors only and for the fault detector trained to detect faults in both the sensors and actuators. For each detector, we performed 20 evaluation runs. We have included the results for a fault detector trained to detect faults in the actuators (treels) only from Section 4.4.1 to allow for comparison.

The results show that a fault detector can be trained to detect faults in the ground sensor and faults in the light sensors. Furthermore, they show that we can train a single fault detector to detect faults in both the sensors and the treels. When a threshold of 0.75 (or higher) is used, no false positives were observed during any of the evaluation runs using the respective fault detectors. The median latency observed for the sensor-only fault detector was 7 control cycles (0.7 seconds). The median latency observed for the sensor and actuator fault detector was 20 control cycles (2.0 seconds), when a threshold of 0.75 was used, which is equivalent to the median latency observed for the treels-only fault detector (see Figure 4.12).

4.4.5 Robustness to Variations in the Task

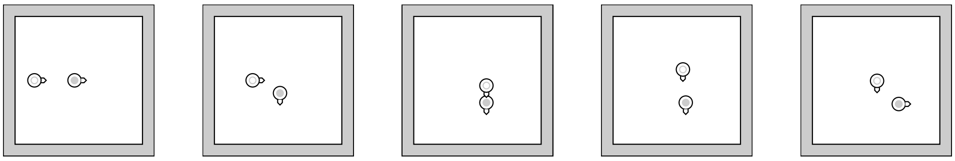
Autonomous mobile robots often navigate in environments in which the exact conditions and task parameters are unknown and sometimes even change over time. A fault detector has to be robust to such changes in order to be generally applicable. We trained a fault detector on data from three variations of the *connect to s-bot* setup. In addition to the original setup in which one *s-bot* connects to another stationary *s-bot* (see Figure 4.4), we collected data from runs in two additional setups, namely *connect to moving s-bot* and *connect to swarm-bot*, illustrated in Figure 4.13.

In the *connect to moving s-bot* setup, the *s-bot* to which a connection should be made (the seed) initially moves around instead of being passive. The seed only stops moving when the two robots get within a distance of $30\ \mathrm{cm}$ or less of one another. In the *connect to swarm-bot* setup, the connecting *s-bot* connects to a *swarm-bot*. In our experiment, the *swarm-bot* consists of three *s-bots* connected in a linear formation.

We trained a fault detector with an input group distance of 5 and 10 hidden nodes on a training set consisting of 60 runs: 30 runs in the original setup described in Figure 4.4, and 15 runs in each of the two setups illustrated in Figure 4.13. The fault detector was evaluated on data from 20 runs: 10 runs in the original setup and 5 in each of the new setups. In order to reduce the number of false positives, the moving average of the TDNN's output was computed (as explained in Section 4.4.3) and compared against a threshold of 0.90. The results observed in 20 evaluation runs using the output of the TDNN directly and using a moving average window length of 25 control cycles are shown in Figure 4.14.

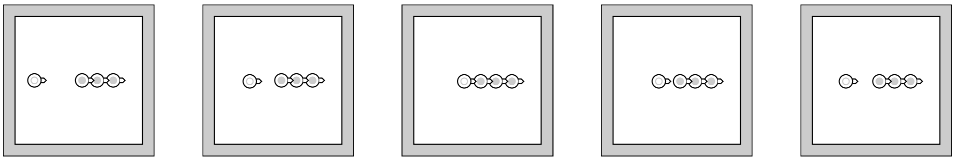
When the output of the TDNN was used directly one fault was not detected, whereas two faults were not detected when a moving average window length of 25 was used. When the output of

Connect to moving s-bot: One *s-bot* attempts to physically connect to another *s-bot* (the seed) using its gripper. As long as the connecting *s-bot* is farther than $30\ \mathrm{cm}$ from the seed, the seed moves around. When the two robots get within $30\ \mathrm{cm}$ of one another, the seed stops. When a successful connection has been made, the connecting *s-bot* waits for 10 seconds, disconnects, moves back, and tries to connect again. Faults are injected in the connecting *s-bot* only.



(1) (2) (3) (4) (5)

Connect to swarm-bot: One *s-bot* attempts to physically connect to a *swarm-bot* using its gripper. In our experiment, the *swarm-bot* consists of three *s-bots* physically connected in a linear formation. When a successful connection has been made, the connecting *s-bot* waits for 10 seconds, disconnects, moves back, and tries to connect again. Faults are injected in the connecting *s-bot* only.



(1) (2) (3) (4) (5)

Figure 4.13: Two additional setups for the *connect to s-bot* controller used to evaluate if a fault detector can generalize over variations of the task.

the TDNN is smoothed over 25 control cycles, false positives only occur in two out of the 20 evaluation runs. Hence, our results indicate that it is possible to train fault detectors that are robust to variations in the task.

4.4.6 Exogenous Fault Detection in a Cooperative Task

In robotics, exogenous fault detection is the activity in which one robot detects faults that occur in other, physically separate robots. The *s-bot* hardware platform used for the experiments in this thesis was originally designed and built in order to study multi-robot and swarm-robotic systems. Such systems have the potential to achieve a high degree of fault tolerance: if one robot fails while performing a task, another robot can take over and complete the task. Various approaches to fault detection and fault tolerance in multi-robot systems have been proposed, such as Parker's ALLIANCE [Parker, 1998], Lewis and Tan's *virtual structures* [Lewis and Tan, 1997], Gerkey and Matarić's MURDOCH [Gerkey and Matarić, 2002b,a], and Dias *et al.*'s TraderBots [Dias et al., 2004] among others (see Section 2.2).

In order to evaluate the applicability of fault injection and learning to exogenous fault detection,

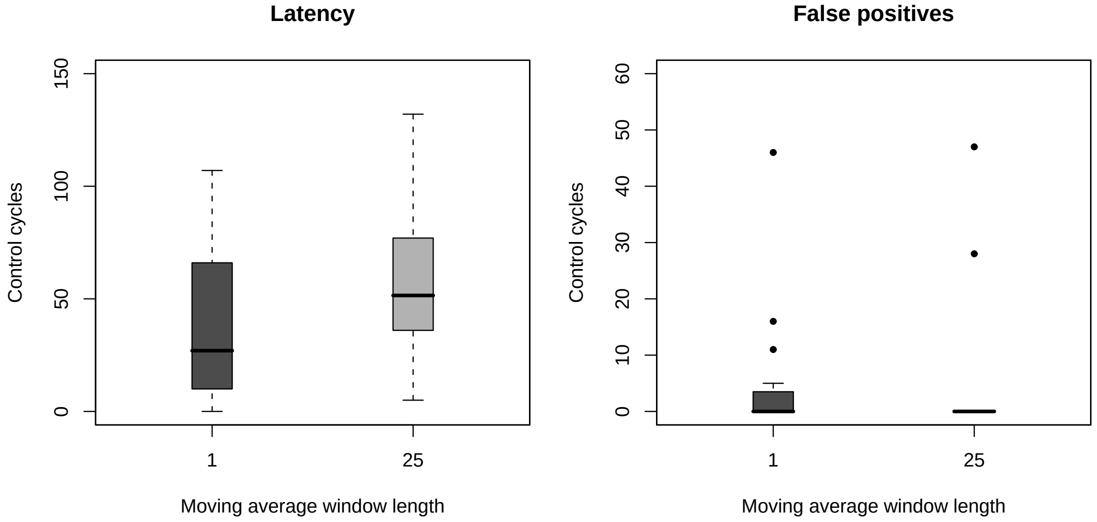


Figure 4.14: Box-plot of the latencies and the number of false positives observed during 20 evaluation runs using a fault detector trained on data from a total of 60 runs in all three variations of the *connect to...* setup. Results are shown for moving average window lengths of 1 (equivalent to using the output of the TDNN directly) and 25. A threshold of 0.90 was used. See the caption of Figure 4.6 for details on box-plots.

we attempted to get the *leader* to detect faults injected in the *follower* in the *follow the leader* setup. We recorded sensory data for the *leader* robot while faults were injected in the *follower*. An overview of the software architectures is shown in Fig. 4.15. The *Control Programs* are responsible for steering the robots. They read sensory inputs and send control signals to the robots' actuators. The *Fault Detectors* passively monitor the flow of sensory inputs and control signals that pass to and from the *Control Programs*. Faults are simulated by the *SWIFI Layer* in the *follower*. When the *follower*'s *Control Program* sends actuator control signals, these commands pass through the *SWIFI Layer*. If no fault is currently being simulated the *SWIFI Layer* forwards all actuator control signals to the robot hardware. If a fault has been injected, control signals to the hardware affected by the fault are discarded. In the case of endogenous fault detection, the fault detector is located in the *follower* in which faults are also injected (see Figure 4.15 top). However, for the experiments in this section concerning exogenous fault detection based on fault injection and learning, the fault detector is located in the *leader* robot (see Figure 4.15 bottom).

In order to train an exogenous fault detector, we collected sensory data (camera and infrared proximity sensors) and actuator control signals from the *leader* while faults were injected in the *follower* robot. The recorded readings from the *leader* were correlated with the fault state of the *follower* and a TDNN was trained on 40 runs to detect exogenous faults. Box-plots of the latencies and number of false positives observed during 20 evaluation runs are shown in Figure 4.16. In the figure, we have plotted results for the *leader* performing exogenous fault detection and the results for the *follower* performing endogenous fault detection during the same runs to allow for comparison. Both fault detectors were configured to use an input group distance of 5 and a classification threshold of 0.75. The median latency for the *leader*

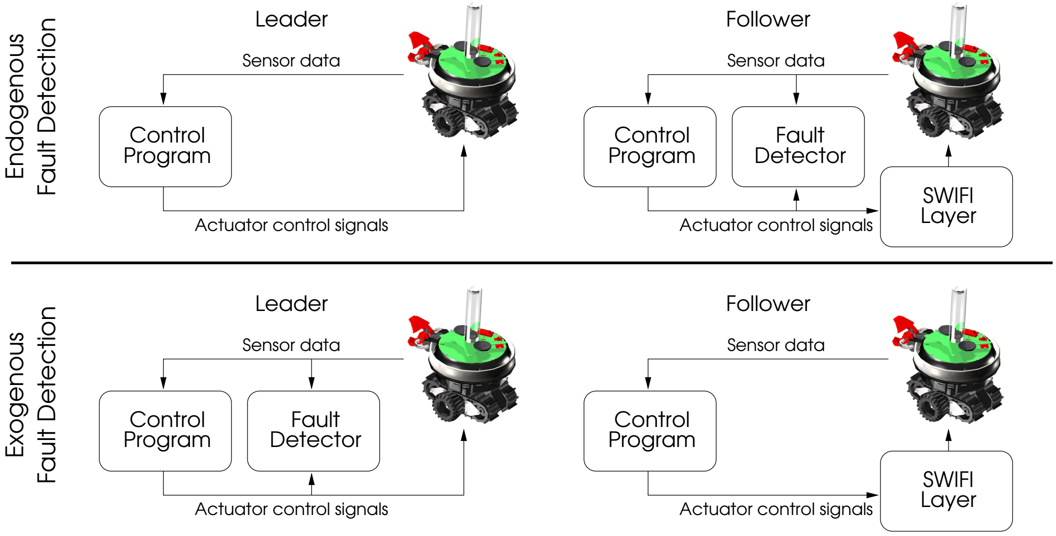

Endogenous Fault Detection
Leader
Sensor data
Control Program
Actuator control signals
Follower
Sensor data
Control Program
Fault Detector
SWIFI Layer
Actuator control signals
Exogenous Fault Detection
Leader
Sensor data
Control Program
Fault Detector
Actuator control signals
Follower
Sensor data
Control Program
SWIFI Layer
Actuator control signals

Figure 4.15: The software architecture for endogenous and exogenous fault detection based on fault injection and learning.

performing exogenous fault detection is 19 control cycles, while the median latency for the *follower* performing endogenous fault detection is 14 control cycles. This difference of 5 control cycles corresponds to 750 ms. The median number of false positives is 5 control cycles for both the exogenous and endogenous detector. Visual inspection of Figure 4.16 confirms that the performance of the two fault detectors is comparable. In every trial, the fault injected was detected by both the *leader* and the *follower*.

In order to reduce the number of false positives, we can compute the moving average of the trained TDNN's output as explained in Section 4.4.3. Figure 4.17 shows the false positive results for the different lengths of the moving average window. For moving average windows up to and including 10 control cycles, false positives occur in several trials. For longer windows, false positives are only observed in one or two trials. When a moving average window of length 50 is used, the exogenous fault detector produces no false positives.

For window lengths of 20-50, false positives for the *follower* performing endogenous fault detection occurred in one of the 20 trials. These results are not shown in Figure 4.17, since they are outside the scale of the figure. The endogenous fault detector produced 173 false positives with a window length of 20 control cycles and 128 false positives with a window length of 50 cycles for the trial in question. When the moving average is used, the latencies are increased by the length of the moving average window (results are not shown). Thus, introducing a threshold-based classification scheme based on the moving average of the TDNN output value can remove false positives in exogenous fault detection, but this comes at the cost of a higher latency.

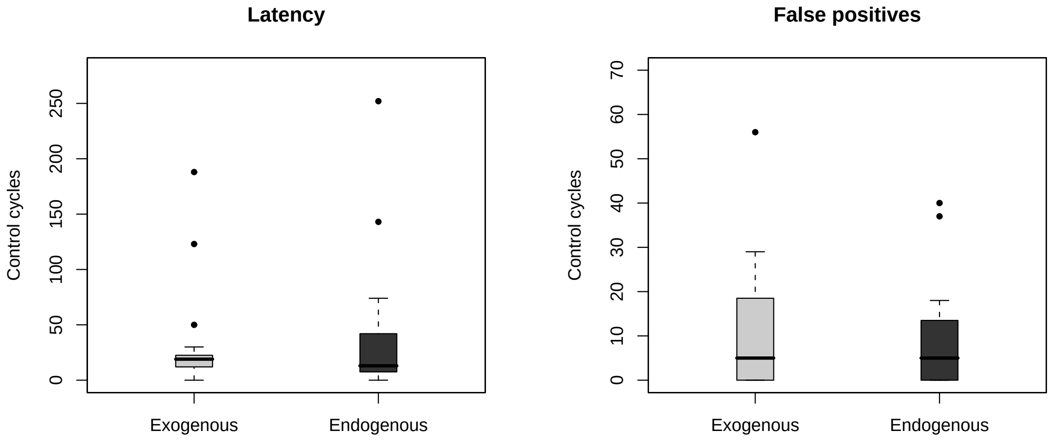


Figure 4.16: Box-plot of the performance results in terms of latency and number of false positives observed in 20 evaluation runs for the *follower* performing endogenous fault detection and for the *leader* performing exogenous fault detection during the same runs. For both sets of results, the output of the TDNN is used directly and compared against a threshold of 0.75. See the caption of Figure 4.6 for details on box-plots.

4.5 Extensions and Limitations

Below we list some possible extensions to the methodology that we have introduced in this chapter. We also discuss some of the limitations of the methodology and how these limitations could be addressed.

Fault Identification

In ongoing research, we are studying extensions to our approach that will allow for fault identification. Our aim is to obtain neural networks that can not only detect the presence of a fault but also the location of the fault. If the control program is made aware that a particular component is broken, it could direct the robot to perform tasks for which the component is not needed or only use the subset of behaviors that do not need the faulty component. For example, a control program might be designed to transport an object by grasping it with its gripper and then pulling the object. If this control program were informed that there was a fault in the gripper, it could change its behavior to push the object without grasping it - this might be less efficient than pulling, but would enable the robot to carry out its task despite the faulty gripper. One way of extending the proposed methodology to include fault identification would be to add more output neurons to the classifying neural network. Different output neurons would then correspond to different faults. Another approach could be to use multiple neural networks, one for each component in which faults should be identified.

The Neural Networks are Large

In TDNNs, the number of input neurons is the product of the number of taps and the number of sensor readings and actuator control signals used from every control step. In the experiments

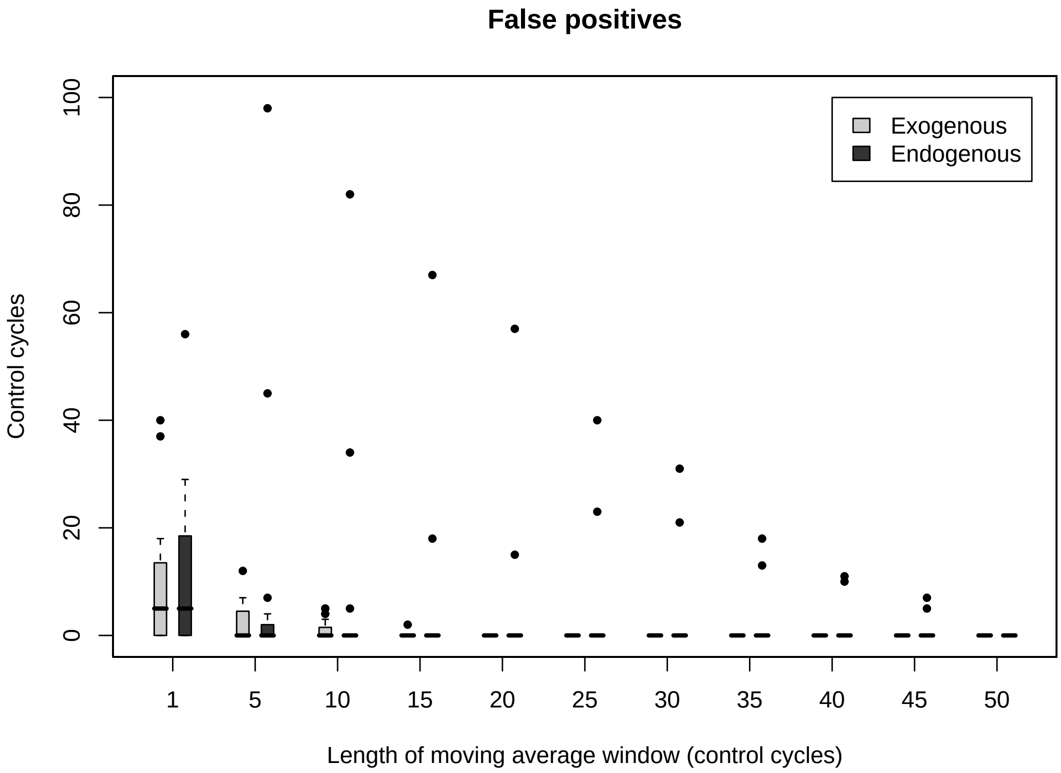


Figure 4.17: Box-plot of the performance results in terms of the number of false positives observed in 20 evaluation runs for the *follower* performing endogenous fault detection and for the *leader* performing exogenous fault detection during the same runs for different lengths of the moving window. For both sets of results, the moving average is compared against a threshold of 0.75. See the caption of Figure 4.6 for details on box-plots.

in this chapter, we used TDNNs with 10 taps. The resulting neural networks had a total of 126, 315 and 205 neurons in the *find perimeter* task (14 inputs), the *follow the leader* task (33 inputs) and the *connect to s-bot* task (22 inputs), respectively. All the input neurons from all taps are fully connected to the neurons in the hidden layer. We used 5 hidden nodes in the fault detectors. The resulting number of weights for the neural networks for the fault detectors in the *find perimeter* task, the *follow the leader* task and the *connect to s-bot* task was 711, 1679 and 1129, respectively. Neural networks of this size can be problematic when storage and computation resources are scarce (which is often the case for autonomous robots).

In our experiments, we used relatively few sensory inputs in the neural networks (14, 33, and 22 respectively). If we had used data from more sensors, such as data from rotary encoders, the number of weights would have increased by 50 per additional sensory input value.[5] Similarly, if we add more taps in order to refine the time-resolution of the inputs and/or expand the time frame based on which the TDNNs classify the state of a robot, for every additional tap the size of the TDNN would grow at a rate equal to the number of sensory inputs times the number of neurons in the hidden layer. Thus, the scalability properties of TDNNs may effectively limit the number of taps and/or sensor inputs on which a fault detector can base classification. This could be a limiting factor on the performance of the detector.

Other, more sophisticated flavors of neural networks, such as recurrent neural networks [Williams and Zipser, 1989, Werbos, 1990, Beer, 1995], are able to reason based on data distributed in time without requiring a tapped delay-line of past inputs. These neural networks retrain an internal state (memory) between subsequent propagations of input data. As a result, the size of these types of networks scales better with the number of inputs. Furthermore, we do not have to experimentally find a good value for the tap distance d, which determines how far into the past the TDNN sees. For other types of recurrent neural networks, these parameters are adjusted in the learning phase in the form of the weights on the recurrent connections (and possibly decay constants). In other words, different neural network structures could be used in order to decrease the space and time complexity of the fault detector.

Further fine-tuning or the use of other classifiers, such as the increasingly popular support-vector machines [Cristianini and Shawe-Taylor, 2000], could improve the classification quality, reduce the memory foot-print, and/or require fewer computational resources. However, since our aim was to show that the proposed methodology requires neither particular classifiers nor dedicated hardware, we chose the TDNN which is one of the simplest types of neural networks that are able to perform classification based on data distributed in time.

The Amount of Training Data Required is Significant

In order to train a fault detector, a significant amount of training data needs to be collected. It takes time to conduct the data collection experiments with real robots. When a control program is changed or if new faults should be considered, new data collection runs may be necessary. This can pose an obstacle to the practical application of the technique described in this chapter.

---

[5] An additional 50 weights are needed because each input has to be replicated once for each tap (= 10) and every copy of the neuron has to be connected to all the neurons in the hidden layer (= 5).

One way in which the data collection process could be sped up is by using simulation. Experiments could be setup and replicated multiple times in simulation with little effort. If a control program is changed or if additional faults are included, a new set of data collection experiments could easily be rerun on a workstation or on a cluster.

In evolutionary robotics, we face a number of issues when we transfer controllers synthesized in simulation to real robots. No simulation is completely accurate and the differences between simulation and the real world can mean that controllers behave differently in simulation and on real robots. The same may hold true if we use data from simulation to train a fault detector for real robots: there may be differences between the two environments that prevent accurate and timely detection of faults. As discussed in Section 1.3.3, there are various ways in which we can address this problem: either by building more accurate (and complex) simulators or by using noise as a means to mask discrepancies and hide features that we do not want fault detectors to rely on.

Another approach could be to use sensory data preprocessing and/or sensor fusion in order to let fault detection depend on high-level (processed) features instead low-level sensory data. We could for instance preprocess camera data (position of nearby LEDs) so that the relative locations of nearby teammates were extracted and used by a fault detector. Since we can choose what information to extract and choose how the information should be preprocessed, we could chose the type of high-level features for which there is a good correspondence between simulation and reality. In this way, sensory data preprocessing and/or sensor fusion may help alleviate some of the transfer issues.

One Neural Network May Not Be Able To Generalize in Complex Tasks

In this chapter, we presented results for three different tasks. We also showed how a fault detector was able to generalize over variations of the *connect to s-bot* task. However, when a task becomes more complex and consists of multiple sub-tasks it is questionable if a single neural network would be able to generalize and accurately detect faults in all sub-tasks. If we combined the three tasks discussed in this chapter, a robot could for instance first try to find the perimeter of a dark geometric shape on the area floor (*find perimeter*) until it encountered another robot. One of the two robots would assume the role of the leader, while the other robot would assume the role of the follower, and the robots would subsequently move around together (*follow the leader*). In case a certain obstacle was encountered, the two robots could self-assemble (*connect to s-bot*) in order to overcome the obstacle. In this scenario — and for complex tasks in general — we could train not one, but several neural networks: one for each sub-task. Fault detection could then be based on the output of the neural network specially trained for the sub-task which the robot is currently performing.

The Approach Does Not Scale to Exogenous Fault Detection in Groups or Swarms of Robots

In Section 4.4.6, we demonstrated how the leader robot was able to detect faults in the follower robot. Exogenous fault detection based on fault injection and learning was applicable in this case because it was straightforward to associate sensory readings from the leader robot with

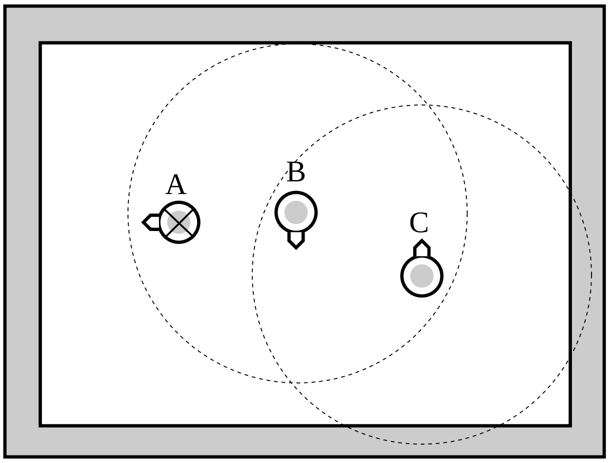


Figure 4.18: An example of three robots, A, B and C, in which robot A has experienced a fault while within the perceptual range of B but outside of the perceptual range of C. The perceptual ranges of B and C, respectively, are indicated by circles.

the state of the follower. The only robot that the leader was able to perceive at any time was the follower. If the task had been different and involved more robots, it would be less clear how to associate a certain flow of sensory information from one robot with the state of another robot.

Imagine, for instance, three robots[6] A, B and C that move around randomly in an arena. If robot A experiences a fault while robot B is within perception range, but while robot C cannot perceive A, we might conclude that robot B should be able to detect the fault in A, while robot C should not be able to detect the fault (at least not directly). The situation is illustrated in Figure 4.18. In this case, we could use readings from robot B in order to train an exogenous fault detector, and we could discard the readings from C. When the robots B and C move, however, the situation changes. Robots B and C might be able to perceive robot A from time to time. In that case, it is no longer obvious how to associate sets of sensor readings and actuator control signals to the state of A. It is therefore difficult to obtain meaningful training data and to synthesize a fault detector.

Exogenous fault detection based fault injection and learning is hard to apply in multi-robot systems because there seems to be no general method of associating a certain set of sensory data with the state of a particular robot that would allow for subsequent training of a fault detector. In Chapter 5, we propose a different approach that allows for exogenous fault detection in large groups of robots. The method relies on firefly-inspired visual synchronization.

4.6 Summary

Dependable fault detection and fault response mechanisms are likely to be a central requirement for autonomous robots before we can realistically expect them to be widely adopted in domestic and industrial environments. Due to concerns over safety and potential costs incurred by malfunctioning robots, the scope of the tasks with which robots can be entrusted will remain fairly narrow until a high level of dependability can be attained. In this chapter, we have suggested a new method for synthesizing fault detectors for autonomous mobile robots. We first presented the approach. Our method is based on learning from examples in which robots operate normally and in which faults are present, respectively. We then went on to test the approach on three different tasks with real autonomous robots. The tasks differed significantly in terms of both the actions performed by the robot(s) and the sensors and actuators used.

---

[6] We assume that the robots are homogeneous, but the issue exists regardless of whether the robots are homogeneous or heterogeneous.

The results suggest that fault detection through fault injection and learning is a viable method to generate fault detectors for autonomous mobile robots. The robots need not be equipped with dedicated or redundant sensors for the method to be applicable. For all the results presented, the only data used was the information flowing between the control program and the robots' sensors and actuators necessary for navigation. Although the performance of fault detectors could probably be improved if data from more (possibly dedicated) sensors were used, our results show that a fairly small amount of key information is sufficient to obtain good fault detectors.

We explored various aspects of the method: We showed that it is possible to train fault detectors to detect faults in both actuators and sensors. We demonstrated that a single fault detector is capable of detecting faults of different types and locations. We showed that we can train a fault detector to be robust to variations in the task performed by a robot. Finally, we showed how the proposed method can be extended to exogenous fault detection: in the *follow the leader task*, we trained a fault detector for the leader robot to detect faults that occurred in the follower robot.

Finally, we discussed various ways in which the approach could be improved and we discussed some of its limitations. We discussed how these limitations possibly could be overcome. However, one of the inherent limitations of the method is that it does not generalize to exogenous faults detection for groups or swarms of robots. In the next chapter, we address this issue and propose a method for exogenous fault detection based on visual firefly-like synchronization.

Fault Detection in Swarms of Robots

Chapter 5

Nature has produced a multitude of remarkably robust and adaptive systems. These qualities are often derived from underlying self-organization mechanisms and from massive built-in redundancy. Examples can be found at scales from the nano to the macro: nanostructures [Pohl et al., 1999], the architecture of a cell [Misteli, 2001], ensembles of cells forming organs such as a human heart [Peskin, 1975], and societies of insects [Bonabeau et al., 1997]. As engineers, we can learn from and try to imitate such natural systems in which complex behavior results from the basic rules of interaction between essentially simple components.

In this chapter, we exploit some of the high-level principles behind synchronizing systems found in Nature in order to obtain a robust, simple, distributed approach to exogenous fault detection in groups or swarms of autonomous robots. By detecting faults, a multi-robot system can leverage its multiplicity and ensure continued operation by reassigning functional robots to the failed robots' task or by taking steps to have the failed robots repaired. Carlson et al. [2004] tracked the reliability of 15 mobile robots from three different manufacturers over a period of three years and found the average mean time between failures to be 24 hours. The result suggests that faults in mobile robots are quite frequent. As the number of constituent robots increases, we would expect the rate of failure to grow correspondingly. Faults are therefore likely to be common events in multi-robot systems.

The method presented in the previous chapter gives a robot the capacity to detect faults in itself. As discussed in Chapter 2, many other studies have been devoted to endogenous fault detection. Some faults are, however, hard to detect in the robot in which they occur. These faults include software bugs that cause a control program to hang, sensor failures that prevent a robot from detecting that something is wrong, and mechanical faults such as an unstable connection to a power source. Alternatively, a robot might be able to detect a fault, but the fault itself might still render the robot unable to alert other robots or a human operator. The robustness of a multi-robot system can therefore be improved by giving robots the ability to detect faults in one another, that is, implementing an exogenous fault detection scheme.

Exogenous fault detection and fault tolerance in multi-robot systems have been studied in [Parker, 1998, Gerkey and Matarić, 2002b,a, Dias et al., 2004] among others. The types of systems studied in the literature differ from the types of systems in which we are interested. Whereas previous studies have mostly been concerned with multi-robot systems that rely on high-level coordination and sophisticated reasoning, we are interested in systems composed of simple robots that self-organize through local interactions. In the majority of the proposed approaches, the robots are required to be tightly coupled. Radio communication is used to facilitate fault detection and/or fault tolerance. As the number of robots grows, tightly coupled systems become harder to realize due to scalability issues.

When we look at Nature, we seldom find centralized approaches. Instead, we find numerous examples of decentralized and self-organized systems: schools of fish changing direction at the same time and never colliding, trail formation in ants, and termite mound construction without a pre-defined blueprint or a central coordination mechanism [Camazine et al., 2001]. The philosophy behind Nature-inspired swarm robotics, such as the method that we present in this chapter, is to rely on self-organization through local interactions between robots. The potential advantages of designs adhering to this philosophy include scalability, inherent parallelism, and robustness to individual failures [Cao et al., 1997, Bonabeau et al., 1999, Shen et al., 2004]. The approach that we advocate in this chapter is completely distributed. Through local interactions, a group of robots are able to synchronize and reach a state in which they flash periodically in unison. When a robot breaks down it also ceases to flash. By detecting the absence of flashes, operational robots can effectively detect failed robots.

This chapter is organized as follows: In Section 5.1, we motivate our approach. In Section 5.2, we discuss previous studies related to synchronization among pulse-coupled oscillators in natural and artificial systems. In Section 5.3, we show how simulated and real robots can synchronize their flashing and we explore various parameters such as robot density, group size and coupling strength. In Section 5.4, we show how non-operational robots can be detected and we provide results for a system where the robots have (simulated) self-repair capabilities. A summary of the results and directions for future research are provided in Section 5.5.

5.1 Motivation

If one robot in a multi-robot system needs to detect a fault in one of its team members, it has to either observe the team member directly, observe the results of its actions, or attempt some form of communication with the team member. In the previous chapter, we demonstrated how we could obtain fault detectors capable of detecting endogenous faults by passively monitoring the flow of sensory data and actuator control signals. However, as we concluded, there appears to be no general and meaningful way of associating sensory data and actuator control signals from one robot to the state of another robot when the robots have limited sensing capabilities and when they are in motion. This is not to claim that fault detection through observation does not work for *any* multi-robot system: if the robots are sufficiently sophisticated to observe one another's actions and if they have sufficient knowledge of one another's task, they could detect faults by observation only.

Instead of detecting faults through passive observation, robots can take action in order to facilitate fault detection. One way of detecting faults is to force each unit to perform a certain action periodically. In the real world, this type of approach is for instance used in trains: a driver of a locomotive has to press a switch (the *dead man's switch*) from time to time in order for the train to continue its course. The purpose of the switch is to provide a fail-safe in case the driver becomes incapacitated. If the driver does not press the switch periodically, the locomotive stops. Similarly, in some distributed computer systems, periodic messages are broadcast by nodes to indicate that they are still operational.

In multi-robot systems, we can adopt a similar strategy: every robot periodically takes a certain deliberate action which can be perceived by other robots. By detecting the absence of the action over a certain time period, team members can detect non-operational robots. On the *s-bots* we have a number of options concerning the type of deliberate action: Wi-Fi communication, sound emission and change of the LED color configuration. Over a Wi-Fi network, robots could periodically send out broadcast messages to indicate that they are still operational. If sound is used, the robots could periodically emit a tone with a certain frequency. If visual communication is used, *s-bots* could periodically change the color of their LEDs. Below we first discuss the three different options (Wi-Fi, sound and LEDs) and go on to discuss different protocols for periodic deliberate actions that allow for exogenous fault detection.

Situated and Abstract Communication

Støy [2001] makes the distinction between *situated* and *abstract* communication. Wi-Fi communication and sound (although to a less degree) suffer from the fact that they provide abstract as opposed to situated communication: when one robot communicates it is not obvious for other robots to determine the source relative to their own frame of reference. Thus, the messages are separated from the environment localization information [Støy, 2001]. To illustrate how this is an issue, imagine a team of three robots: one of the robots perceives only one "I am alive" message over the Wi-Fi network (or hears only one beep sound if auditory communication is used). Hence, one of its two team members are no longer operational, but which one? Unless the robots have unique features (for instance bar-codes that can be read by nearby robots, different colors, or similar) to distinguish them and somehow encode this feature in the message (for instance an ID), other robots will not know which robot is sending the message. Thus, the robots will know that one of their teammates has become non-operational, but they have no way of knowing which one and can therefore do little to accommodate the fault.

In some cases radio communication and auditory communication can be situated to a certain degree: technologies like ZigBee and Bluetooth (currently none of these technologies have been implemented on the *s-bots*) allow robots to propagate signals locally. In some cases, a crude approximation of the direction and distance to the emitting source can be obtained. In dense swarms of robots it is, however, still problematic to determine exactly what robot emitted which signal. A similar problem manifests itself with auditory communication: if multiple microphones are used (*s-bots* have four microphones), a robot can with reasonable accuracy detect the direction of a sound source using time delay estimates. However, estimating the distance with a sufficiently high accuracy is still difficult.

Visual communication is situated. Whenever a robot changes the colors of its LEDs, other robots nearby perceive this from their own perspective. The source and relative location of the emitting robot is therefore immediately obvious. The *s-bots* can control their LEDs and set each of the eight sets of RGB colored LEDs to one of three colors – red, green and blue. Illuminated LEDs can be perceived and distinguished by *s-bots* up to 50 cm away depending on light conditions. In the next section, we discuss different protocols for exogenous fault detection based on visual communication.

Protocols for Exogenous Fault Detection based on Periodic Action

We discuss three different protocols for exogenous fault detection based on periodic action and the *s-bots*' ability to control the color configuration of their on-board LEDs:

Unsynchronized flashing: *S-bots* change the color of their LEDs periodically. Every two seconds, for instance, an *s-bot* changes from either red to green or from green to red. A robot that does not change the color of its LEDs may have a fault.

Ping-pong: With a low probability, a robot sends out a message (a ping) to which surrounding robots should respond (a pong). If a robot does not reply, it may have a fault. In the following, we assume that a robot sends out a ping by illuminating its red LEDs while other robots respond with a pong by illuminating their green LEDs.

Synchronized flashing: Robots change color in a synchronized fashion, any robot that is not flashing when the rest of the swarm flashes may have a fault.

In the unsynchronized flashing protocol, every robot has to constantly monitor all surrounding robots. Keeping track of individual robots is hard. The image processing capability of the *s-bots* is limited: the *s-bots* can only see illuminated LEDs. Features of such as the gripper, the treels, and chassis cannot be identified by the image processor. Each LED is detected as one or more distinct blobs (see Figure 3.4 on page 24 for an example). One robot can partly or completely occlude another robot. When a robot moves around, the perspex tube holding the camera and the hemispherical mirror (see Figure 3.1 on page 22) shakes and as a result even static objects "jump around" from frame to frame. Hence, it can be difficult for a robot to determine what LEDs belong to which robot, especially when two or more neighboring robots are close to one another. Furthermore, due to the robots' limited visual range, they enter and exit each other's view repeatedly. These issues can make it hard identify and track moving robots and thus to realize a working implementation of the unsynchronized flashing protocol.

In the ping-pong protocol, the pinging robot (a robot that illuminates its red LEDs for a brief period of time) has to ensure that all the surrounding robots correctly respond with a pong (by illuminating their green LEDs for a brief period of time). In the ping-pong protocol, a robot therefore also needs keep track of surrounding robots, but only when it decides to send out a ping. When a robot sends out a ping it can stop its motion so that the camera and perspex tube do not move. In this way, the *s-bot* can see more accurately if the surrounding robots respond correctly. The issue of keeping track of surrounding robots is therefore not as pronounced in the ping-pong protocol as it is in the unsynchronized flashing protocol. There are, however, two issues with the ping-pong protocol:

- *A robot fails to see a ping.* Example: One robot (N) cannot see a pinging robot (P) when robot P changes color in order to ping. Robot P, on the other hand, sees robot N before the response time limit is reached. Robot P could (wrongly) conclude that N is non-operational because N does not respond to the ping that it did not see.
- *Constant pong.* Example: One robot receives ping messages from different robots in succession and it therefore replies by constantly having its green LEDs illuminated for an extended period of time. A robot that is merely responding to one ping immediately after another is hard to distinguish from a robot that has become non-operational while it was replying.

The issues related to the ping-pong protocol and to the unsynchronized flashing protocol could probably be solved by making the protocols more complex or by introducing better sensory equipment. However, since our aim is to demonstrate exogenous fault detection on real *s-bots*, we have chosen the synchronized flashing protocol: when robots flash in unison, there is no need for a single robot to identify all its neighbors. It is sufficient for a robot to detect LEDs with the wrong color (which is evidence that one of the team members in not flashing) while the robot itself is flashing. In this way, more complex reasoning such as constantly keeping track of all the surrounding robots is not necessary. Furthermore, the two issues listed above for the ping-pong protocol do not exist for the synchronized flashing protocol: since the robots flash in synchrony, every robot is in effect "pinging" and "ponging" at the same time.

The challenge in implementing a synchronized flashing protocol is to make all the robots change color periodically at the same time. However, Nature has already solved this problem in an elegant manner. In the following sections, we discuss synchronization in natural and artificial systems and we show how robots can synchronize based on a firefly-inspired approach. We then demonstrate an algorithm to reliably detect non-operational robots. Finally, we show how 10 real *s-bots* are capable of detecting faults in each other and how they can survive a high rate of failure when they have the capacity to repair one another.

5.2 Synchronization in Natural and Artificial Systems

In Nature, we find many examples of coupled oscillating systems that lead to various types of synchronous behavior. The canonical example is large groups of tropical fireflies, found on river banks in Southeast Asia, which spontaneously synchronize their rhythmic flashes [Buck, 1988, Smith, 1935]. Other examples include cardiac cells [Glass, 2001], choruses of grasshoppers [Snedden et al., 1998], female menstrual cycles [McClintock, 1971], and clapping in theaters [Néda et al., 2000].

Systems of coupled oscillators can be divided into two classes: oscillators that continuously influence one another (see for instance [Strogatz, 2000]) and so called *integrate-and-fire* or *pulse-coupled* oscillators, where one oscillator only influences other oscillators during short, periodic pulses. In this study, we focus on the latter type. The internal state or *activation* of each oscillator increases over time until it reaches a certain threshold. When the threshold is reached, the oscillator discharges (*fires*) and the activation instantly jumps back to zero – the cycle then repeats. When a nearby oscillator observes a flash it immediately increases its activation by a (small) amount. If this increase causes the oscillator's activation to exceed the firing threshold, the oscillator fires, resets its activation to zero, and commences a new cycle. Analytically, many pulse-coupled networks can be written in the following form [Izhikevich, 1999]:

$$\dot{x}_i = f(x_i) + \epsilon \sum_{j \in \mathsf{N}} h(x_i)\delta(t - t_j^*), \tag{5.1}$$

where $x_i \in [0, 1]$ denotes the activation of oscillator i. The function f describes its dynamics. The pulse-coupling constant ϵ defines the strength of the coupling between oscillators. N is

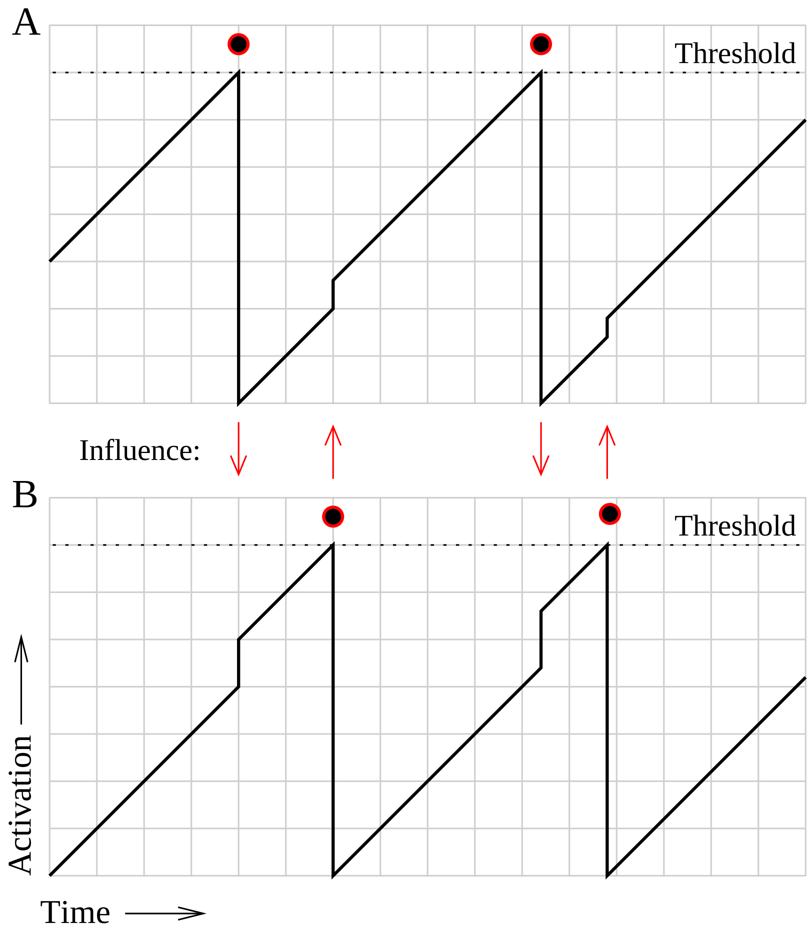


Figure 5.1: An example of two pulse-coupled oscillators. Both oscillators increase at a constant rate until the threshold is reached or until one oscillator observes that the other one fires. When an oscillator's activation reaches the threshold, the oscillator fires. If one oscillator observes the other's firing, it increases its own state by ϵx, where ϵ is the pulse-coupling constant and x the activation of the oscillator.

the set of oscillator i's neighbors. The pulse-coupling function h describes the effect of the firing of another oscillator j on i. The time t_j^* marks the moment when j last fired. The delta distribution function $\delta(t)$ satisfies that $\delta = 0$ for all $t \neq 0$, $\delta(0) = \infty$ and $\int \delta = 1$. An example with two oscillators for which f is constant and h is linear is shown in Figure 5.1.

Peskin [1975] was the first to propose a model for self-synchronization of pulse-coupled oscillators after observing cardiac pacemaker cells. Mirollo and Strogatz later showed that a population of fully connected pulse-coupled oscillators almost always evolves to a state in which all oscillators are firing synchronously [Mirollo and Strogatz, 1990]. Recently, Lucarelli and Wang [2004] showed that a group of pulse-coupled oscillators will eventually synchronize even when each oscillator interacts with only a subset of the population. This holds true for systems with changing topologies as long as the interaction graphs are connected.[1] Ramíre Ávila [2004] has experimentally shown that light emitting pulse-coupled oscillators synchronize.

---

[1] We obtain the interaction graph for a population of oscillators by letting every oscillator correspond to a node

Understanding synchronization is not only important for describing natural phenomena – synchronization is a central issue in distributed computing and distributed sensing. The problem of establishing a consistent global time base across nodes in a distributed system subject to message delays, network congestion, node failures, and clock skews has received a great deal of attention (see for instance [Tanenbaum and van Steen, 2002, Elson and Estrin, 2001]). The behavior of fireflies has inspired algorithms for heartbeat synchronization in overlay networks [Babaoglu et al., 2007], imposing reference timing in wireless networks [Tyrrell and Auer, 2007], and in sensor networks for coordinating sensing and communication [Werner-Allen et al., 2005].

In this study, we rely on local visual communication. We are therefore not faced with issues such as variable propagation delays and congestion that several studies on synchronization across data networks have had to deal with.

5.3 Synchronization in Robots

We propose an approach for synchronization based on local visual communication. The approach resembles behavior observed in fireflies: we let each robot act as an integrate-and-fire oscillator and when the activation of the oscillator reaches a certain threshold, the robot lights up its red LEDs as in the example shown in Figure 3.3 and resets its oscillator. When neighboring robots (within $50\,\mathrm{cm}$) detect the flash, they increment their own activation.

5.3.1 Discrete Oscillators

Due to the inherent discreteness of the sense-think-act control paradigm, we transform the continuous model in Eqn. 5.1 into a discrete model with piece-wise linear dynamics:

$$x_i(n+1) = x_i(n) + \frac{1}{T} + \epsilon\,\alpha_i(n)\,h\big(x_i(n)\big), \tag{5.2}$$

where $x_i(n)$ is the activation of robot i at control cycle n. T is the period between flashes of an isolated robot. In this study, we have chosen T to be 100 control cycles (which corresponds to 15 s). We experiment with different values for the pulse-coupling constant ϵ in Section 5.3.2. $\alpha_i(n)$ is the number of flashing robots seen by i at control step n. We use the linear pulse-coupling function:

$$h(x) = x \tag{5.3}$$

When $x_i(n)$ exceeds 1, robot i flashes and its activation is reset to 0. There is a (small) latency between the moment that the control program sends a signal to the flash LEDs and until the moment they respond. Before a neighboring robot can perceive a flash, it must have already recorded and processed a frame from its on-board camera in which the flash is visible.

---

in the graph with an edge to each member of its neighbor set.

This step entails an additional latency. Furthermore, images from the camera are retrieved and processed asynchronously by the on-board software in a separate execution thread. We have experimentally found that we can compensate for these delays by keeping the flash LEDs on for 5 control cycles (0.75 s). Flash spans of this length allow the robots to reach and remain in a synchronized state. When robots in a synchronized system keep their flash LEDs on for 5 consecutive control cycles, they perceive the flashes from other robots, while they are flashing themselves. This creates a stable fix-point for the system. We compute $\alpha_i(n)$ based on the most recent frame recorded by the on-board camera. In order to prevent the robots from perceiving the same flashes multiple times, we compute $\alpha_i(n)$ in the following way: we dissect a robot's omni-directional field of view into 16 equally sized slices and count only flashes from slices from which no flashes were perceived in the previous control cycle. In this way, we obtain a reasonable estimate of the number of flashing robots without the need to identify and track individual robots (as discussed in Section 5.1, it is difficult for one robot to keep track of each neighboring robot).

5.3.2 Synchronization Experiments in Simulation

We are interested in the time it takes for all robots to synchronize. We define the system to be synchronized when it is in a state where the value of every activation $x_i(n)$ is no further than $1/T$ from all other activations. An example of the development of activation values sampled every T during a run with 25 robots in simulation is shown in Figure 5.2. The activation for each robot at every T–th simulation step is plotted as a cross. In the example, the robots synchronize after 435 s.

The time it takes for a swarm of robots to synchronize depends, apart from the parameters ϵ and T, also on the density of robots, on how the robots move, and on the total number of robots. In a given environment, the density of robots together with the total number of robots define the average degree and the diameter of the interaction graph, while the pattern of movement for the robots defines its dynamic properties. As the pattern of movement is strongly task-dependent, we limit our experiments to two extreme cases: one in which all robots perform a random walk and one in which all robots are static. In both types of experiments, the robots start at random positions and with random orientations. In the experiments with static robots, a check is performed before the start of each experiment to ensure that the interaction graph is connected. In case it is not, all the robots are repositioned until a configuration is found that produces a connected interaction graph.

In simulation, we have explored the rate of synchronization for swarms of 10, 25, 50, and 100 robots (see Figure 5.3). Moreover, we tested the rate of synchronization at densities of 2, 4, 6, 8, and 10 robots/m^2 in square-shaped arenas with 50 robots (see Figure 5.4). Finally, we tested the rate of synchronization for pulse-coupling strengths ϵ ranging from 0.01 to 0.50 (see Figure 5.5). In all figures, one bar represents the mean synchronization rate observed in 100 replications of the experiment and the error-bars denote the standard deviation.

For all experimental configurations, moving robots tend to synchronize faster than static robots. Visual inspection of the experiments confirmed that a system of static robots in many cases reaches a state of near-synchrony, where flashes propagate in waves through the swarm before

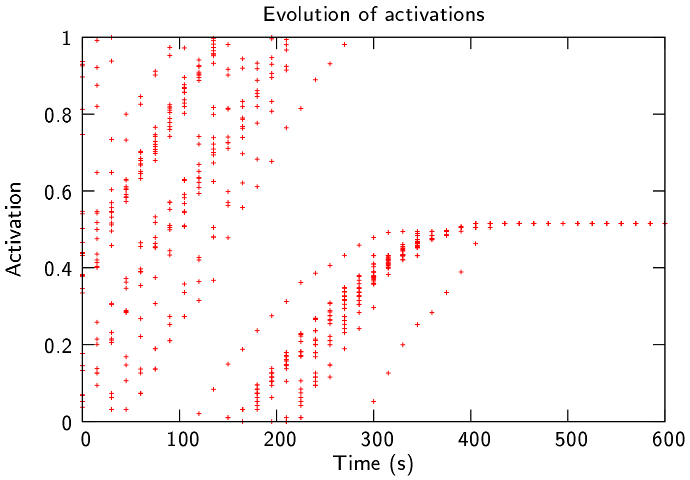


Figure 5.2: An example of the evolution of activations sampled every T in 25 mobile robots over the course of 10 minutes. One cross represents the activation for a single robot at the corresponding time.

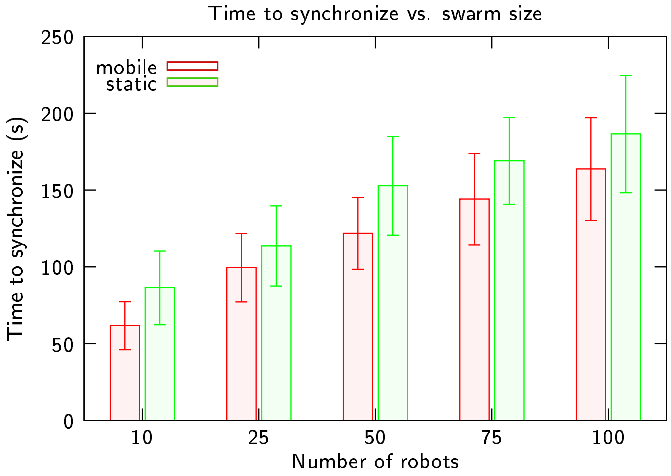


Figure 5.3: Synchronization rate in groups of 10 to 100 simulated robots. Each bar summarizes 100 runs and error-bars denote the standard deviation. The density was 8 robots/m^2 and a coupling constant of $\epsilon = 0.1$ was used in all runs.

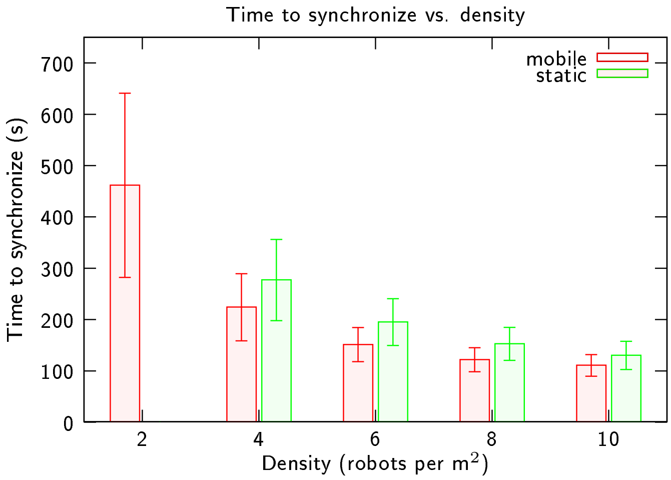


Figure 5.4: Synchronization rate in a group of 50 simulated robots at different densities. Each bar summarizes 100 runs and error-bars denote the standard deviation. A coupling constant of $\epsilon = 0.1$ was used in all runs. Due to the limited sensory range of the *s-bots* (up to 50 cm) experiments with static robots at a density of 2 robots$/\mathrm{m}^2$ were not conducted. At this density, the interaction graph is almost never connected when the robots are distributed randomly.

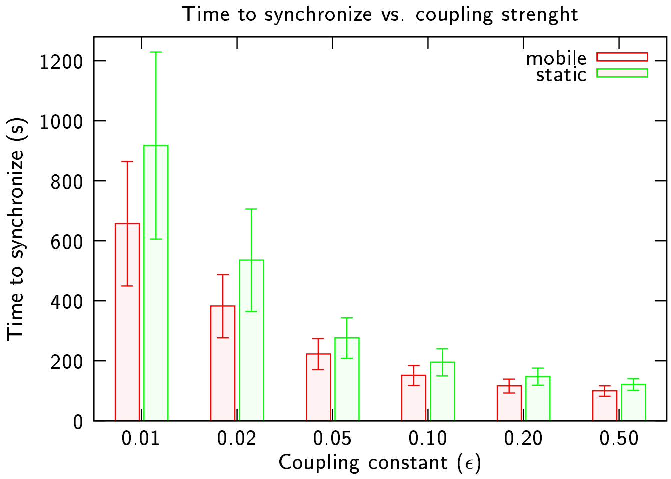


Figure 5.5: Synchronization rate a group of 50 simulated robots for coupling constants $0.01, 0.02, 0.05, 0.10, 0.20, 0.50$. Each bar summarizes 100 runs and error-bars denote the standard deviation. The density was 6 robots/m^2.

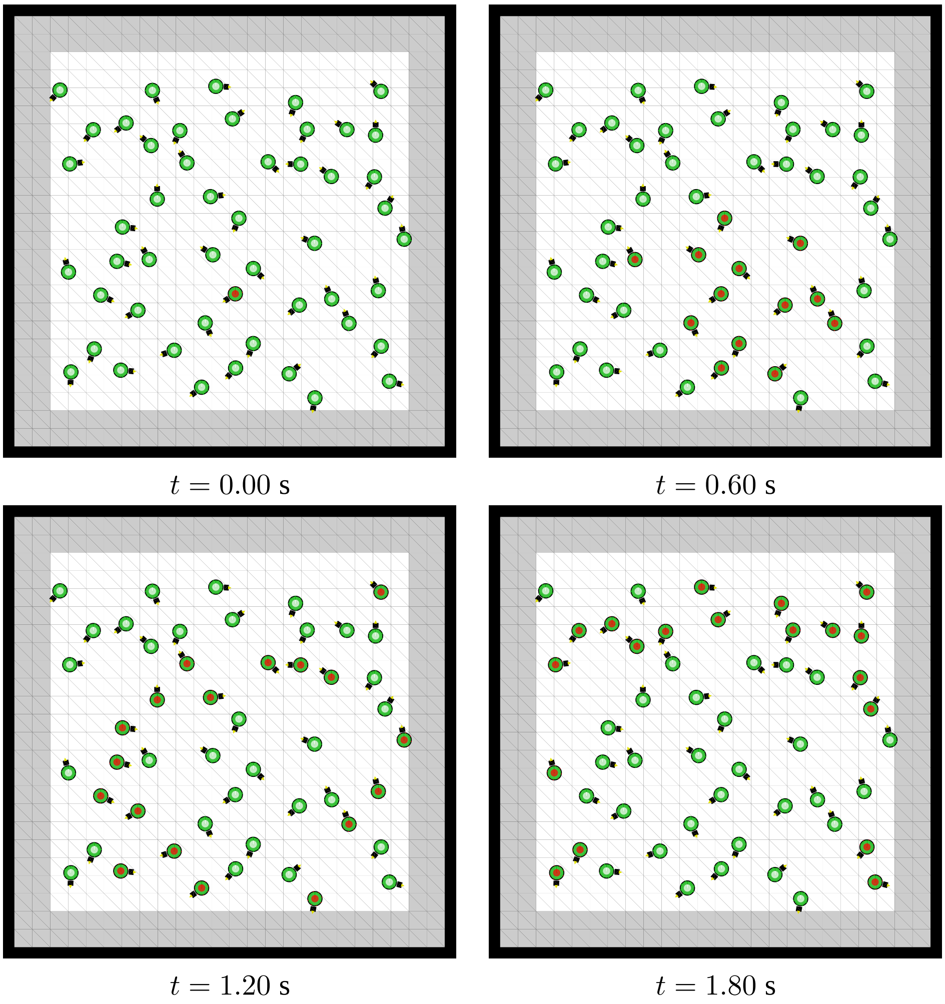


Figure 5.6: An example of a flash wave in a group of static robots.

the robots synchronize. Snapshots of a flash wave propagating through a static swarm of robots is shown in Figure 5.6. When the flash wave starts, all the robots have activations close to the firing threshold and they therefore flash as soon as a nearby robot flashes. Flashes would not propagate in waves if robots could perceive and respond to flashes instantly, because as soon as one robot flashes, all robots would flash (and they would be synchronized). Wave propagation of flashes can thus occur due to the latencies associated with turning on the flash LEDs in the flashing robot and the image capture and image processing.

In moving robots, the wave propagation phenomenon is not as pronounced. Robots that are close to each other have similar activation values when flashes propagate in waves. However, when the robots move, the interaction graph changes. This means that the individual robots do not remain at the same distance from the origin of the flash wave as the system evolves. When individuals that are close to the robot that triggers the flash waves move away, they cause other robots (more distant from the wave origin) to flash sooner. Similarly, as individuals further away from the robot that triggers the flash wave move closer to the wave origin, they are driven to flash sooner, thus speeding up the global synchronization process.

Table 5.1: Synchronization rate for 10 real robot.

	Mean	St.dev.	Shortest	Longest
Static	94 s	72 s	55 s	174 s
Moving	77 s	28 s	30 s	118 s

The synchronization rate scales linearly (with a gentle slope) with swarm sizes up to 100 *s-bots* (see Figure 5.3). The mean synchronization rate for a group of 10 *s-bots* is 62 s, while for 100 *s-bots* the rate is 164 s – less than 3 times as long.

The synchronization rate at different densities plotted in Figure 5.4 shows that denser swarms tend to synchronize faster. When a swarm is dense, more members are within each others' sensory range. The results indicate that the larger the subset of robots each individual interacts with, the faster the overall group synchronizes.

The strength of each interaction is controlled by the coupling constant ϵ. The results in Figure 5.5 show that if ϵ is large a swarm tends to synchronize faster. Setting ϵ too high is, however, problematic when we want to detect faults because one robot – including a failed one – has a significant effect on its neighbors. Furthermore, for large swarm sizes, high values of ϵ can make the system unstable and prevent it from synchronizing.

5.3.3 Synchronization Experiments with Real Robots

We conducted two sets of experiments with 10 real robots: one set of experiments with static robots and one set of experiments with moving robots (random walk and obstacle avoidance). The experiments were performed in a walled arena with dimensions $1.6\ \mathrm{m}$ x $1.6\ \mathrm{m}$ (yielding a density of $4\ \mathrm{robots/m^2}$). The coupling constant ϵ was set to 0.1 and the flash period T was 100 control cycles. The robots were assigned different initial random activations. The initial positions for the robots were obtained in the same way as in simulation (see Section 5.3.2). Based on video recordings, the synchronization rate was measured as the time from the frame in which the robots were started until the first frame in which all the robots had their flash LEDs illuminated. Figure 5.7 shows an example of synchronized robots flashing. The experimental setups with static robots and with moving robots, respectively, were replicated 10 times with different initial conditions. A summary of the results is shown in Table 5.1. Videos of the experiments can be downloaded from:

`http://iridia.ulb.ac.be/supp/IridiaSupp2008-012`

In real robots, we observe the same trend as in simulation: moving robots tend to synchronize faster than static robots. The mean synchronization rate of static robots was 94 s while the mean synchronization rate for mobile robots was 77 s. In all 10 experiments with static robots and in all 10 experiments with moving robots, the robots synchronized. The results indicate that real robots operating as pulse-coupled oscillators are able to synchronize despite the discrete nature of the control sense-think-act paradigm and despite the inherent latencies associated with the sensory and actuator systems.

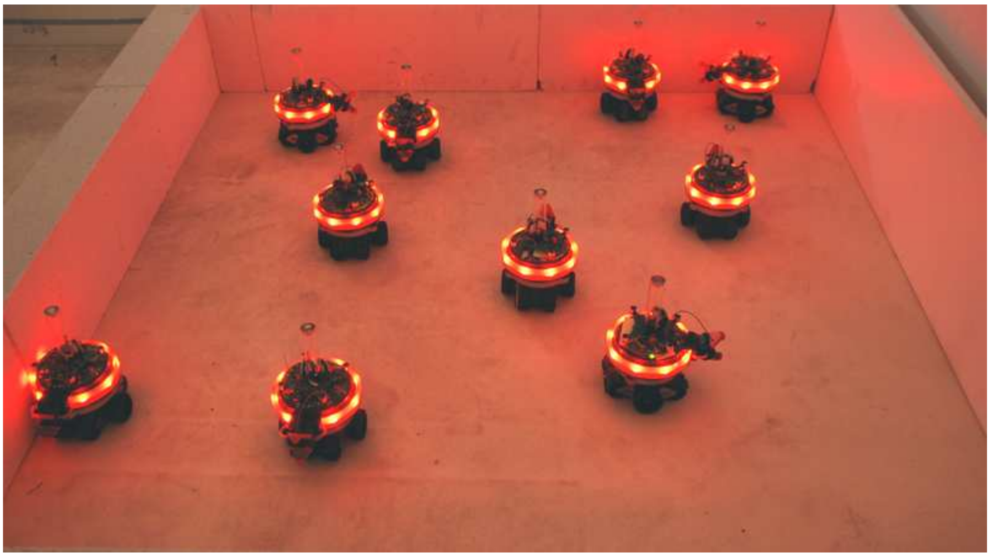


Figure 5.7: A photo of synchronized robots flashing at the same time.

5.4 Fault Detection in Swarms of Robots

Synchronization can be used as an exogenous fault detection tool if the robots assume that a robot that is not flashing has a fault. A robot can decide to stop flashing if it detects a fault in itself. In this way, it can implicitly signal that it requires assistance. A robot also stops flashing when it experiences a catastrophic fault (software bug, physical damage, and so on...) which causes the control program and thus the periodic flashing to stop. When operational robots discover a non-flashing teammate they know that a fault has occurred and they can take steps to rectify the situation. Conceptually, the scheme is straightforward. However, two issues need to be addressed in order for the scheme to be implemented on real robots: it cannot be assumed that the robots are always synchronized and the sensory range of the robots is limited.

5.4.1 Detecting Faults in Non-Synchronized Robots

In a normal situation, the robots would be operational and synchronized (see Figure 5.8a). However, when robots commence a task or when they encounter each other after having been separated for a period of time, their activations are likely to differ. In other words, they are not synchronized. This means that one robot cannot assume that another robot has become non-operational just because the two robots do not flash in unison. To address this issue, a flashing robot does not immediately consider another robot non-operational if the two robots do not flash at the same time. Instead, the flashing robot (F) treats the robot (N) that did not flash when F flashed as a *candidate* robot. We say that F becomes *suspicious* of N. If N flashes before F flashes again, both robots are operational but they are just not (yet) synchronized (see Figure 5.8b). However, if F flashes *again before* N flashes, F assumes that N is non-operational (see Figure 5.8c). Hence, a robot detects a fault if it flashes twice while observing that another robot does not flash at all.

There is however a problem with this scheme. In fact, there is a rare situation in which one

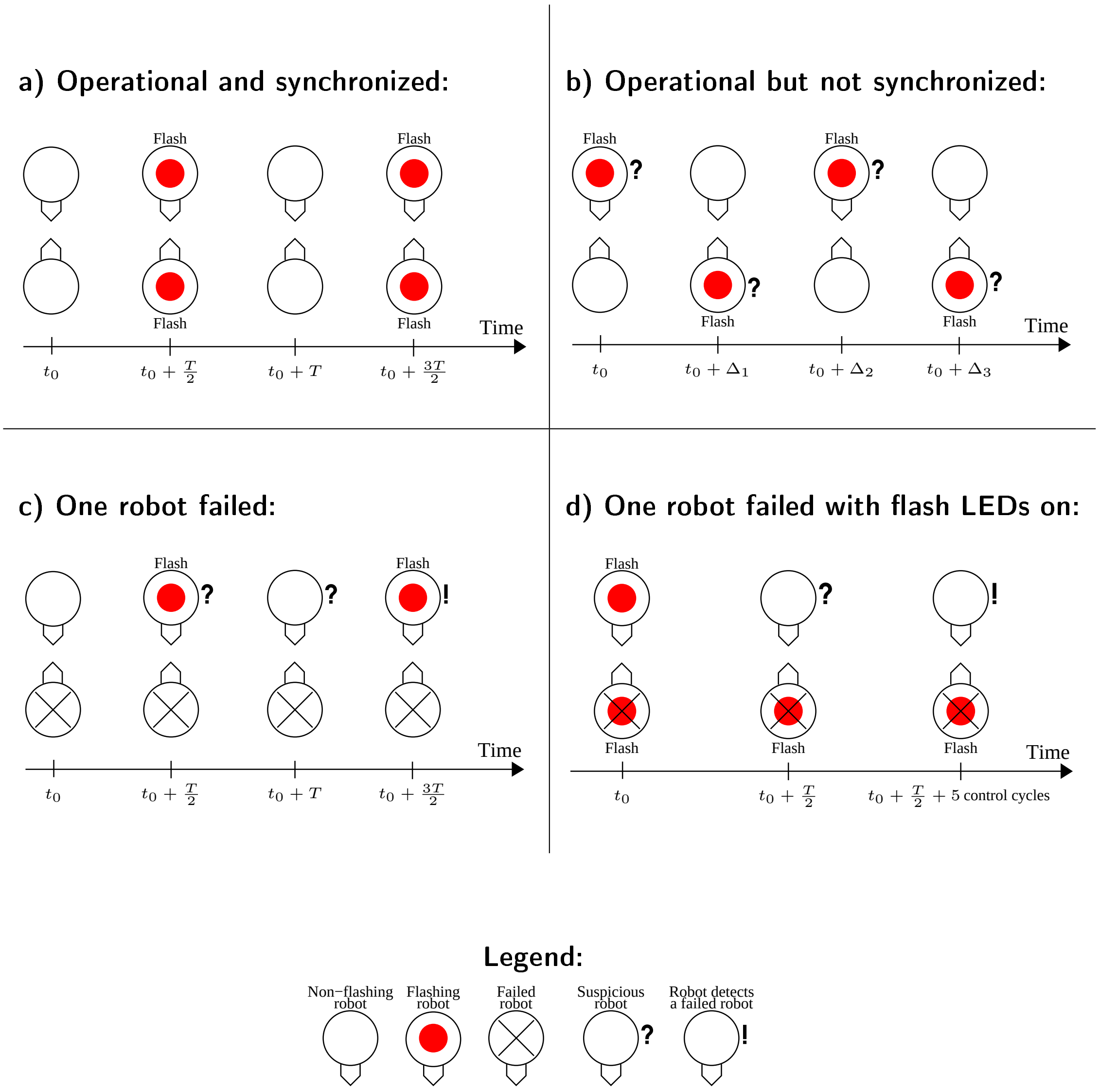

a) Operational and synchronized:
Flash
Flash
Flash
Flash
Time
t_0
$t_0 + \frac{T}{2}$
$t_0 + T$
$t_0 + \frac{3T}{2}$
b) Operational but not synchronized:
Flash
?
Flash
?
?
Flash
?
Flash
Time
t_0
$t_0 + \Delta_1$
$t_0 + \Delta_2$
$t_0 + \Delta_3$
c) One robot failed:
Flash
?
?
Flash
!
Time
t_0
$t_0 + \frac{T}{2}$
$t_0 + T$
$t_0 + \frac{3T}{2}$
d) One robot failed with flash LEDs on:
Flash
?
!
Flash
Flash
Flash
Time
t_0
$t_0 + \frac{T}{2}$
$t_0 + \frac{T}{2}$ + 5 control cycles
Legend:
Non−flashing robot
Flashing robot
Failed robot
Suspicious robot
?
Robot detects a failed robot
!

Figure 5.8: Four possible scenarios. See text.

operational robot (*R2*) can flash twice while another *operational* robot (*R1*) does not flash a single time. This can happen when *R1* flashes right before *R2* and when *R2* subsequently perceives sufficient flashes to increase its activation so much that it flashes again before *R1* flashes a second time. However, R2's second flash will often provoke R1 to flash. R2 can, in fact, calculate the sufficient conditions under which its second flash will provoke R1 to flash. When these conditions are met and R2's second flash does not provoke R1 into flashing, R2 can safely assume that R1 has a fault. We let Δ denote the amount by which *R2*'s activation has been increased due to flashes from other robots. In the worst case, *R1*'s activation has not been advanced by any flashes. When *R2* reaches the firing threshold $(= 1)$ *R1*'s activation is therefore at most Δ away from the firing threshold, i.e. *R1*'s activation is at least $1 - \Delta$. Assuming that the two robots perceive each others' flashes, *R2*'s second flash will increase *R1*'s activation by at least $\epsilon h(1 - \Delta)$. Thus, *R2*'s second flash will drive *R1* to flash if:

$$\epsilon h(1 - \Delta) \geq \Delta. \tag{5.4}$$

Thus, if R2 flashes twice while R1 does not flash at all (including not being provoked to flash by R2's second flash) and if Eqn. 5.4 is true, R2 can conclude that R1 has a fault. Otherwise R2 must wait until its next flash to determine whether or not R1 is operational. If R1 has still not flashed in the meantime it must have a fault.

The case in which a robot breaks down *while* it is flashing is not caught by the scheme presented above. In other words, no robot would ever become suspicious of a robot that becomes non-operational while its flash LEDs are illuminated. Consequently, the non-operational robot would never be detected. Faults that occur while the robot is flashing, leaving the flash LEDs illuminated, however, can easily be detected: when a robot's activation passes its midpoint (0.5), it becomes suspicious of any robot that has its flash LEDs illuminated. If the candidate robot still has it LEDs on after the normal flash span (5 control cycles), the suspicious robot can conclude that the candidate robot with the flash LEDs on is not operating correctly. This situation is illustrated in Figure 5.8d.

5.4.2 Time Overhead

When a group of robots start a new task, they are not always synchronized. This means that the robots do not flash at the same time. While a group of robots is in the process of synchronizing, they frequently become suspicious. While they are suspicious, they stop performing the task and wait while they determine if the candidate robot is non-operational or if it is just not synchronized. This has a negative impact on the performance of the group as time that could have been used for carrying out a task is spent on being suspicious. In Figure 5.9, we have plotted the average percentage of control cycles that the robots were suspicious in the beginning of an example run.

The time initially spent by the robots on being unnecessarily suspicious while a group of robots is synchronizing can be reduced or eliminated entirely by introducing a *warm-up period*. During the warm-up period a robot ignores any indications of faults and does not become suspicious. If we had introduced a warm-up period of 120 s or longer in the experiment summarized in Figure 5.9,

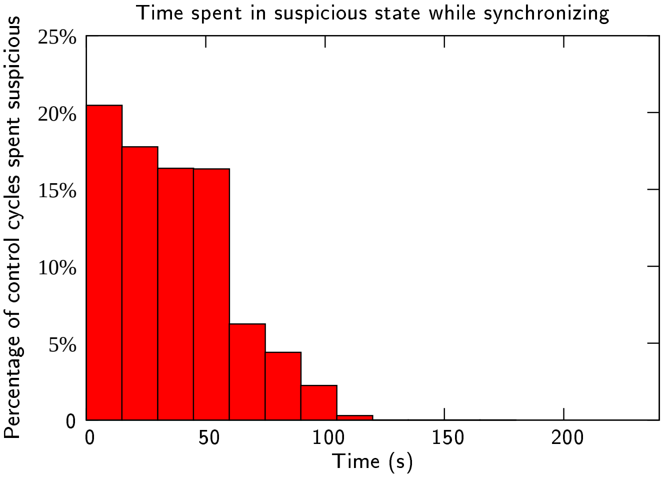


Figure 5.9: Average percentage of the control cycles spent in the suspicious state over intervals of 15 s during a run with 50 simulated robots in a 2.5 m x 2.5 m arena. The robots were not initially synchronized.

none of the robots would have become suspicious during the initial synchronization period, and the initial overhead of the stop-while-suspicious strategy would have been eliminated. However, there is a trade-off between the length of the warm-up period and the latency of fault detection since faults cannot be detected during the warm-up period.

5.4.3 Implementation

An overview of the control and fault detection logic executed every control cycle is shown in Algorithm 1. The activation x is incremented by the sum of the constant increase $1/T$ and the product of the coupling strength ϵ, the number of flashes seen α, and the pulse-coupling function $h(x)$. When x exceeds 1, the robot flashes and checks for non-flashing robots, while if x has just passed 0.5, a check is made to determine if any neighboring robot has become non-operational with its flash LEDs illuminated. In case a candidate robot is found, the robot stops and waits until it can be decided whether the candidate is operational or not. If no candidate was found, the robot performs a random walk while avoiding obstacles.

The logic for checking for non-flashing candidates and for candidates with their flash LEDs illuminated is shown in Algorithm 2 and Algorithm 3, respectively.

Algorithm 1: ControlCycle()

```
ReadSensors();
if HasFlashed($candidate$) then
    $candidate = none$;
end
$x \leftarrow x + \frac{1}{T} + \epsilon\alpha h(x)$;
$\Delta \leftarrow \Delta + \epsilon\alpha h(x)$ ;
if $x > 1$ then
    FlashAndCheckForFailedRobots();
    $x \leftarrow 0$;
end
if $x$ has passed $0.5$ then
    CheckForFailedRobotsWithFlashOn();
end
if $candidate \neq none$ then
    StopMoving();
else
    RandomWalkAndAvoidObstacles();
end
```

Algorithm 2: FlashAndCheckForFailedRobots()

```
TurnOnFlashLeds(5 cycles) ;
if $candidate = none$ then
    $candidate$ = CheckForNonFlashingRobots();
    $\Delta = 0$;
else
    if $\epsilon h(1 - \Delta) \geq \Delta$ then
        $failedrobot \leftarrow candidate$;
        ...A fault has been detected. Take
        ...actions to deal with the fault.
    else
        ...Wait until next flash before concluding
        ...if the $candidate$ robot has failed or not.
    end
end
```

Algorithm 3: CheckForFailedRobotsWithFlashOn()

```
$candidate$ = CheckForFlashingRobots();
if $candidate \neq none$ then
    StopMoving();
    if CandidateRobotStillFlashing() after 5 control cycles then
        $failedrobot \leftarrow candidate$;
        ...A fault has been detected. Take
        ...actions to deal with the fault.
    end
end
```

5.4.4 Fault Detection Experiments with Real Robots

In 10 experiments, we measured the time it took for one or more robots to detect and react to a failed robot. We took the first steps towards a scenario in which one or more robots

Table 5.2: Fault reaction time results on real robots

Mean reaction time	53.2 s
Standard deviation	31.3 s
Shortest reaction time	30.2 s
Longest reaction time	135.5 s

can facilitate the repair of a failed robot – either directly or by physically connecting and transporting the failed robot[2] to a special zone where the robot is then repaired or replaced. The experiments were performed in the same arena and with the same parameter settings as the synchronization experiments for mobile robots described in Section 5.3.3 (an arena of 1.6 m x 1.6 m, $\epsilon = 0.1$, and $T = 100$ control cycles). In each experiment, we let a group of 10 robots synchronize and then we simulated a catastrophic fault in one of the robots. We measured the time from the moment a fault was injected until one of the operational robots reacted to the fault by detecting the fault and physically connecting to the failed robot. The results are shown in Table 5.2.

The fault was correctly detected in all 10 experiments. The mean reaction time was 53.2 s. This result includes the times required for the following activities: an operational robot detects the absence of a flash, the operational robot is suspicious for up to T (15 s), the operational robot navigates to and grasps the failed robot.

The shortest reaction time to a fault was 30.2 s. In the corresponding experiment, the fault was injected just before the other robots in the swarm flashed and a nearby robot therefore became suspicious less than a second after the fault had been injected. Furthermore, the robot that detected the fault was close to the failed robot and had an orientation that allowed it to quickly connect to the failed robot. In the experiment in which the longest reaction time (135.5 s) was observed, at first only a single operational robot detected the fault. It unsuccessfully attempted to grasp the failed robot twice. Eventually another operational robot detected the fault and connected to the failed robot.

In one of the experiments, a real fault occurred. After an operational robot had detected and connected to the robot in which we had injected a fault, a robot experienced a real hardware I/O error. The error rendered the robot unable to control any of its actuators, including its treels and its LEDs. This real (non-simulated) fault was also detected and an operational robot connected to the failed robot.

5.4.5 Fault Tolerance Experiments with Real Robots

In order to test our approach in a scenario where more than one robot can become non-operational, we conducted an experiment with a group of 10 robots, in which a fault was injected in an operational robot with a probability of $p = 0.0005$ every control cycle. We simulated a repair mechanism that allowed one robot to "repair" another robot by physically connecting to it and by illuminating its blue LEDs for 15 s. When a failed robot detected that it had been "repaired", it set its activation to a random value and restarted its controller. We let the experiment run for 12 minutes. All robots were operational from the start of the experiment and the first fault occurred after 20 s. During the experiment a total of 13 simulated faults occurred. At one point a total of four robots were non-operational, while only one robot was non-operational when the experiment was stopped.

---

[2] *S-bots* have been shown capable of collectively transporting objects that are larger and heavier than an *s-bot*, see for instance [Gross et al., 2006c].

A robot experienced a real hardware I/O fault similar to the one described above in Section 5.4.4. Two neighboring robots detected the fault and connected to the robot with the real fault. After the two robots had connected to the failed robot and performed the repair action, we removed the failed robot from the arena. We let the other nine robots continue while we restarted and reintroduced the failed robot $3\ \mathrm{min}$ later. Furthermore, we manually put a robot upright after it had toppled over due to a collision with two other robots. A video of the experiment can be found on:

`http://iridia.ulb.ac.be/supp/IridiaSupp2008-012`

The results suggest that our approach is robust in situations where multiple faults can be present at the same time. Furthermore, when the robots can repair one another, a swarm of robots can survive a relatively high rate of failure.

5.4.6 Limitations of the Approach

For our approach to work, we assume that when a robot experiences a fault, it also ceases to flash periodically. For catastrophic faults such as the real hardware I/O fault that occurred in two experiments described above, the *s-bots* do, in fact, automatically stop their periodic flashing as a direct consequence of the fault. However, a fault such as a broken wheel or a toppled robot does not automatically cause a robot to stop flashing. For those types of faults, we assume that the robot can detect the fault itself using for instance the fault injection and learning technique described in Chapter 4. A robot that detects a fault in itself by means of endogenous fault detection can *decide* to stop its periodic flashing in order to signal other robots for assistance. However, if the endogenous fault detector fails to detect a certain fault and if the robot's software and hardware still render it capable of flashing, other robots will be unable to determine that the robot has a fault.

5.4.7 Limitations of the Current Implementation of the Approach

The current implementation of the *s-bot* vision software only allows the *s-bots* to see objects that display illuminated LEDs. In our experiments, the *s-bots* therefore always illuminate their on-board LEDs in some color – in red to indicate that they are flashing and in green in order to be visible to other robots while they are not flashing. Robots with simulated faults stopped flashing periodically, but their LEDs remained illuminated in the color configuration they had when the fault occurred. Some faults, such as a dead battery, would cause an *s-bot* to go dark, that is, since the robot would no longer display any LEDs it would not be visible to the other *s-bots*. If we assume that operational robots have their LEDs illuminated in some color and if dark *s-bots* could be detected in some way, faults causing an *s-bot* to turn dark could be easily detected: the absence of any illuminated LEDs would immediately indicate that the dark *s-bot* had become non-operational. In the current implementation, however, we have not implemented any way of detecting dark *s-bots*.

In our experiments, we assumed that the presence of a fault causes a robot to stop moving. This means that it was sufficient for a robot to also stop when it became suspicious of a non-flashing robot in order to determine if the robot eventually would flash or if it had a fault. Some real catastrophic faults, such as hardware I/O faults, may not always cause a robot to stop its movement. In order to ensure that faults are correctly detected even when the faulty robot is moving, a suspicious robot would have to track the candidate robot and stay within visual range until it could determine whether or not the candidate robot has a fault.

5.5 Summary and Directions for Future Work

In this chapter, we have presented a distributed approach to determine the presence of faults in members in swarms of robots. Our algorithm is inspired by the synchronous flashing behavior observed in some species of fireflies. Robots flash periodically by lighting up their on-board LEDs. Whenever a robot perceives a flash from a nearby robot, it increases its own activation and flashes slightly sooner than if it had not seen a flash. We showed that swarms of simulated and real robots following this scheme are driven to flash in synchrony. The rate of synchronization was found to depend on the size of the swarm, the number of robots that each member interacts with, the coupling strength between the robots (the effect of one robot's flash on another nearby robot), and whether the robots move or are stationary.

In our exogenous fault detection scheme, the periodic flashes function as a heartbeat mechanism. A failed robot need not explicitly signal other nearby robots that it requires assistance – it only needs to stop flashing. We do not, therefore, need to distinguish between robots that have decided to stop flashing after they have detected a fault in themselves and robots that, for instance, have experienced a catastrophic fault rendering them unable to take any action – including flashing. We showed that real robots are able to detect and respond to faults by detecting non-flashing robots. We also showed that the scheme is robust to multiple faults and that a team of robots with self-repair capabilities is able to survive a relatively high rate of failure.

In many previous studies fault detection was facilitated by global negotiation and radio communication. In contrast, our firefly-inspired fault detection approach is completely distributed and relies on local information only. A potential advantage of a distributed approach is scalability, which becomes an increasingly important factor as larger swarms of, for instance, hundreds or thousands of robots are considered. In our experiments, we found that the rate of synchronization depends on the size of a swarm. This means that as the size of a swarm grows it takes longer for the robots to synchronize. However, swarms need not be *globally* synchronized for our fault detection scheme to work efficiently – it suffices that robots are synchronized *locally* with nearby robots. It would thus be interesting to determine the performance of our approach when a swarm is in a global state of near-synchrony, for instance, when waves of flashes are propagating through the swarm (see Section 5.3.2).

A potential direction for future research is implementing and evaluating the performance of our approach in a real task-execution scenario. While carrying out a task, operational robots could detect and transport failed robots to a pre-designated zone and alert a human operator, who could then repair or replace the failed robots. Another interesting question is how to extend the approach to take advantage of possible heterogeneities in a swarm, e.g. robots with different sensory, manipulation, and/or repair capabilities. This type of heterogeneity could possibly be leveraged to facilitate faster synchronization, faster fault detection and true self-repair, while still allowing for a completely distributed, swarm intelligent approach.

Summary and Future Work

CHAPTER 6

In this chapter, we summarize our contributions and we give directions for future studies.

6.1 Summary of Contributions

In Chapter 1, we motivated why fault detection and fault tolerance represent important challenges that need to be addressed before widespread adoption of autonomous robots for domestic and for industrial purposes can occur. Since robots operate in the physical world, a malfunctioning robot can directly cause human injury and/or material damage. In Chapter 1, we also listed the scientific publications on which this thesis is based. We listed and summarized our scientific contributions related to morphology control and to evolutionary robotics. We presented TwoDee, a fast and flexible *s-bot* simulator. TwoDee has been used by several researchers and parts of it have been reused in two other simulators, namely TwoDeePuck and ARGoS.

In Chapter 2, we reviewed the literature on endogenous fault detection for autonomous robots, exogenous fault detection and fault tolerance in multi-robot systems.

In Chapter 3, we presented the *swarm-bot* robotic platform and the capabilities of constituent *s-bots*. The main novelty of the *swarm-bot* platform is that its components, the *s-bots*, can self-assemble in order to overcome their individual physical limitations. At the same time, a single *s-bot* is capable of carrying out meaningful tasks on its own. We discussed some of the studies conducted with the *s-bots* such as self-assembly, coordinated motion, cooperative transport, and morphology control. We described other multi-robot systems and modular self-reconfigurable robotic systems, and we argued why fault detection and fault tolerance are important in such systems.

In Chapter 4, we proposed a new method for detecting endogenous faults in autonomous robots. We demonstrated how fault injection and learning can be applied in order to synthesize fault detectors. For three tasks, *find perimeter*, *follow the leader*, and *connect to s-bot*, we collected training data in 40 experiment of 1000 control cycles for each task (corresponding to 100, 150 and 150 seconds, respectively). We injected faults during the data collection runs. We subsequently trained time delay neural networks to classify the state of a robot, that is, whether it is operating normally or if it has a fault, based on the sensory data used for navigation by the control program and the control signals sent to the robot's actuators. We demonstrated how faults could be accurately detected with a relatively low latency. We also demonstrated that fault detectors can be trained to be robust to variations in a task. Finally, we demonstrated how a fault detector could be obtained that gave the *leader* the capacity to detect faults that occurred in the *follower*. Our results show that a fairly small amount of information from non-dedicated sensors and knowledge about the control program's actions (actuator control signals) was sufficient to achieve accurate and timely fault detection.

Two of the tasks, namely the *follow the leader* task and the *connect to s-bot* task, involved two or more robots. For previous studies in endogenous fault detection it has been uncommon to consider complex multi-robot setups. Furthermore, many studies in model-based fault detection are concerned with deriving an analytical model of how an operational robot moves - but the derived model is never tested on real robots (for an example see [Dixon et al., 2001]). We therefore consider it a significant contribution that we have demonstrated the applicability of the proposed method – not only on real robotic hardware, but that we have demonstrated good performance in complex setups involving multiple robots.

In Chapter 5, we proposed a method for exogenous fault detection in large groups or swarms of robots. We showed how robots acting as pulse-coupled oscillators are able to synchronize the periodic flashing of their LEDs. We explored the synchronization rate in simulation for various values of parameters such as pulse-coupling strength, density and group size. We demonstrated that a system composed of 10 real robots synchronized in 10 replications of an experiment in which they remained static and in 10 replications of an experiment in which they performed random walk and obstacle avoidance. In our exogenous fault detection approach, the periodic flashes function as a heartbeat mechanism: robots that do not flash are assumed to have a fault, while robots that flash periodically are assumed to be operational. We showed that real robots have the capacity to detect and respond to faults by detecting non-flashing robots. With a group of 10 real robots, we demonstrated how exogenous faults were correctly detected in 10 replications of an experiment in which one fault was injected in each replication. We also showed that the approach is applicable when multiple robots experience faults and that a team of robots with self-repair capabilities is able to survive a relatively high rate of failure. In two experiments, a real fault occurred. In both experiments, the fault was correctly detected and operational robots connected to the failed robot to simulate its repair. To the best of our knowledge, the experiments presented in Chapter 5 are the first in which real autonomous robots detect exogenous faults and take action in order to repair failed robots (in this case by physically connecting to them and by performing a simulated repair action).

Many of the previously proposed methods for exogenous fault detection rely on radio communication and/or global knowledge of the system. Our approach is completely distributed and uses only locally accessible information. We therefore believe it scales to large groups or swarms of robots.

6.2 Challenges for the Future

Below we discuss directions for future work and some of the open challenges concerning the proposed methods for endogenous and exogenous fault detection, respectively.

Endogenous Fault Detection

In Chapter 4, we described how endogenous fault detectors were trained to distinguish incorrect behavior from normal behavior based on observations from experiments in which faults were simulated. An obvious extension would be to include *fault identification*, that is, not only to detect the presence of a fault, but also its location. This could be useful when, for instance, a gripper breaks during transport of a heavy object. If the control program is made aware that the robot has a fault *and* that the failed component is its gripper, the control program could, for instance, steer the robot to push the object instead of unsuccessfully trying to connect to the object and pull it. One way of extending our methodology to include fault identification is to add more output neurons to the neural network in the fault detector. Different output neurons would then correspond to different types of faults. Another approach would be to use multiple neural networks, one for each fault type/location.

In our experiments concerning endogenous fault detection, we trained time delay neural networks. We used data from several sensors and a total of 10 taps. As a result, the neural networks became large in terms of the number of neurons in the network and the weights that define the connections between the neurons. We furthermore had to experimentally determine a good value for the input group distance (the parameter that defines the amount of time the network 'sees' into the past). We used time delay neural networks due to their ability to classify based on data distributed in time and their simple feed-forward structure. As discussed in Section 4.5, one of the ways to improve the scalability of the approach is to use more sophisticated neural network structures such as recurrent neural networks. We have conducted some initial experiments concerning the evolution of continuous real-time recurrent neural networks (see [Beer, 1995]) in simulation and the results are promising. It is, however, still an open question if fault detectors trained (in this case *evolved*) in simulation have a good performance on real robots.

In general, it would be beneficial to study ways to obtain training data (at least partially) for fault detectors in simulation. One of the issues with the synthesizing fault detectors based on fault injection and learning is the effort needed to collect training data. If some or all training data could be obtained in simulation instead, the approach would be far easier to apply in practice. Imagine, for instance, the addition of a fault injection module and a data collection module in the Autonomous Robots Go Swarming (ARGoS) simulator currently under development for the *swarmanoid* project (see page 13): researchers could synthesize fault detection modules by specifying the sensors, actuators, possible faults, and by selecting a special data collection mode in the simulator. The simulator would then run the controllers a certain number of times, inject faults and save sensory data and actuator control signals. The data collected could subsequently be used for training a fault detector. The whole process could largely be automated when data collection experiments are conducted in simulation.

For the above scheme to work, however, it is important that the fault detectors can transfer from simulation to reality. Since ARGoS allows the user to specify different physics engines, researchers could choose the more accurate and complex engine when collecting fault detection training data. For simulation-only based training to work on real robots, we may face issues similar to those that we find in evolutionary robotics, namely crossing the reality gap [Jakobi et al., 1995]. No simulation is completely accurate and the differences between simulation and the real world may cause differences in behavior when controllers have been evolved in simulation. Similarly, fault detectors trained on data from simulation may rely on features that may not exist in the reality or that may differ to a degree that prevents a fault detector from achieving a good performance. One way of overcoming this issue is by adding noise [Jakobi, 1998] or by using sensor fusion as discussed in Section 4.5. With sensor fusion, fault detectors could rely on more high-level information for which it may be easier to assure good correspondence between simulation and reality. The use of sensor fusion to bridge the reality gap is an approach that needs to be explored to determine if it is applicable in fault detection based on fault injection and learning.

Exogenous Fault Detection

We have demonstrated that a group of robots acting as pulse-coupled oscillators synchronize both when all the robots are static and when they perform random walk. We demonstrated how synchronization can be used for exogenous fault detection. It would be interesting to test the approach in a real task-execution scenario in order to determine its performance. Whenever a fault is detected, some meaningful accommodation action should be performed by one or more operational robots. A first step could be for another robot to take over the failed robot's task. In a more elaborate scheme, we could utilize the *s-bots*' ability to connect to and transport one another. A failed robot could be moved out of the way or transported to a special repair zone where a human operator could repair and/or replace the failed robot. In a real task-execution scenario, it would also be interesting to combine endogenous fault detection with exogenous fault detection. In case a robot detects an endogenous fault, it could stop flashing in order to (implicitly) signal for assistance.

Multi-robot systems do not always consist of homogeneous robots. New initiatives, such as the *swarmanoid* project, are focused on the study of heterogeneous robotic systems, that is, multi-robot systems consisting of units with different capabilities. Firefly-inspired synchronization and exogenous fault detection could be applied in such systems as well. However, given that the constituent robots differ in sensory, navigation and manipulation hardware, a number of options for leveraging the heterogeneity in the context of synchronization and fault detection exist. Consider, for instance, the system shown in Figure 6.1 composed of *s-bots* navigating on the ground plane using a differential drive system and *eye-bots* [Roberts et al., 2007] navigating in the air using a quad-rotor propulsion system. The *eye-bots* are designed to carry high-resolution camera equipment in order to provide long-range sensing from elevated positions. The *eye-bots* have few manipulation capabilities. In contrast, the *s-bots* have only short range sensing capabilities (around 50 cm depending on light conditions) but they can grasp one another and cooperatively transport items such as a broken robot. This type of heterogeneity could possibly be leveraged to facilitate faster synchronization, faster fault detection and a higher degree of fault tolerance. In Chapter 5, we concluded that the more members each robot interacts with, the faster the swarm synchronizes. Since *eye-bots* interact with (or at least perceive) more robots due to their long range sensing capabilities and due to their elevated position, they could serve as synchronization facilitators: for robots on the ground the pulse-coupling constant ϵ could be higher for flashes perceived from above than from other *s-bots*. Alternatively, *s-bots* could adjust their activation *only* when flashes from above are perceived and ignore flashes from other *s-bots* altogether.

The *eye-bots* are in a good position to detect failed (non-flashing) robots (see Figure 6.1). Similarly they can effectively guide operational *s-bots* to rescue or repair failed robots. Hence, our method for exogenous fault detection based on firefly-inspired synchronization could be

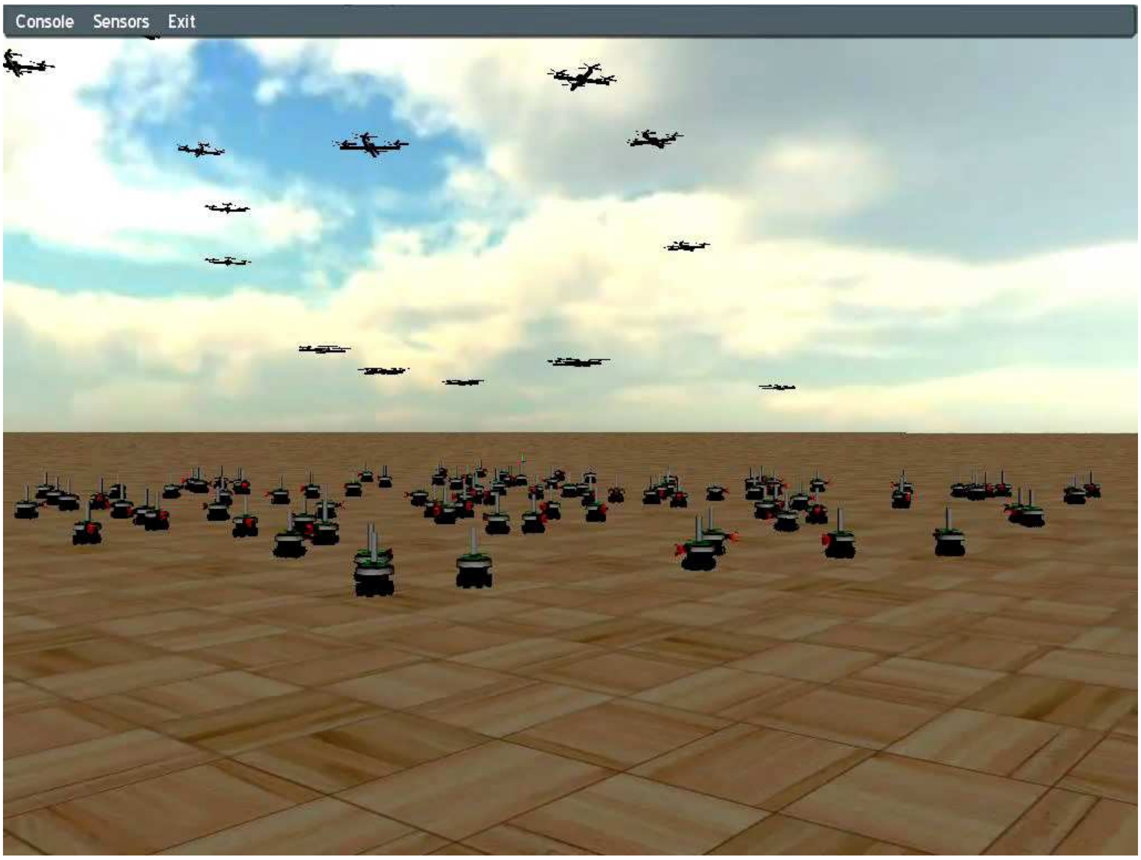


Figure 6.1: An example of a heterogeneous swarm of robots: *s-bots* navigate on the ground using a differential drive systems, while *eye-bots* are quad-rotor flying robots with long-range sensing capabbilities.

specialized to take advantage of certain heterogeneities in robot swarms in order to provide faster synchronization, faster fault detection and faster response to faults.

Conclusions

Chapter 7

Detecting faults in autonomous robots is challenging. Robots have limited sensing capabilities and imperfect actuators, and they are often situated in unstructured environments. Due to these factors it can be hard to determine if a robot is operating normally or if a fault has occurred. One method for implementing fault detection capabilities in autonomous robots is to add redundant sensors and/or proprioceptive sensors. If redundant sensors are used, discrepancies between the readings from sensors measuring the same phenomenon can be symptoms of a fault. Proprioceptive sensors, on the other hand, can verify that actuators are functioning correctly. In both cases, faults can be detected in a relatively straightforward manner, namely by comparing readings and/or by checking that readings are within acceptable bounds. The addition of hardware, however, drives up the cost, complexity and power consumption and it is therefore not an option in many cases. Furthermore, with the added complexity there is more hardware that can break. An alternative (or complementary) approach is to try to distinguish between normal and abnormal operation through more complex reasoning based on readings from existing sensors. There are two main categories of such techniques, namely model-based approaches and model-free approaches.

In model-based approaches, some model of how a system is supposed to behave is constructed and the actual behavior of the robot is compared to the predictions of the model. If the predicted behavior and the actual behavior differ sufficiently, the robot is not operating as it is supposed to, which can be due to the presence of a fault. Two central issues in applying model-based fault detection in autonomous robots are: i) it is a non-trivial effort requiring expert knowledge to construct the analytical model of the system, and ii) it can be difficult to determine if any discrepancies between the predicted and actual behavior is due to noisy sensors and imperfect actuators or due to a fault.

In model-free approaches, fault detectors are usually obtained through the application of data-driven techniques. We discussed how for instance artificial immune systems and novelty filters, can be employed in order to provide abnormality (fault) detection. The method we proposed for endogenous fault detection is model-free. We trained fault detectors to detect faults based on data collected through experiments with real robots. Over a number of trials, the robot for which a fault detector was desired performed its task. During each trial, a fault was injected. We recorded what the robot sensed and the signals that the control program sent to the robot's actuators. We trained a neural network to classify the state of the robot, that is, whether the robot has a fault or not, based on the data collected. Since the faults are purposefully injected in our approach, we can correlate sensory data and actuator control signals with the state of the robot. In this way, supervised learning can be used to train a neural network to distinguish between normal operation and operation influenced by the presence of a fault.

Catastrophic faults cannot be detected by the robots in which they occur. In some cases, a robot might be able to detect that it has a fault, but still be unable to take any action as a consequence of the fault. Catastrophic faults have to be detected externally. In multi-robot and modular robotic systems, robots often cooperate closely and fault detection and fault accommodation is necessary to ensure that a fault in one unit does not comprise the operation of the whole system. On the other hand, a multi-robot system has the potential to achieve a high degree of tolerance to faults, namely by leveraging their multiplicity. When a robot fails, another robot can take steps to repair the failed robot or take over the failed robot's task.

We showed that fault injection and learning could give one robot the capacity to detect faults in another robot. However, we also discussed why fault injection and learning is difficult to use for exogenous fault detection in groups or swarms of robots: when a fault is injected in one of the robots, it is not clear how and which sensory data and actuator control signals should used for training a fault detector.

We proposed a novel method based on firefly-inspired synchronization for exogenous fault detection in swarms of robots. As opposed to fault injection and learning, our firefly-inspired method for exogenous fault detection requires the robots to take deliberate action in order to facilitate fault detection. In our approach, the robots flash periodically by changing the color of their on-board LEDs to notify surrounding team members that they are still operational. If one robot sees that another robot no longer emits flashes periodically, it can conclude that the non-flashing robot has become non-operational. Due to the limited sensing capabilities of the robots used and in order to simplify the approach, we let the system of robots operate as pulse-coupled oscillators. This means that whenever one robot flashes, the team members within visual range of the flashing robot each flash slightly sooner than they would have if they had not seen the flash. As a result, the robots eventually synchronize and flash periodically in unison. When the robots are synchronized, it is relatively simple to detect non-flashing robots: each robot can detect non-flashing team members by looking for LEDs that do not have the right color when the rest of the swarm of robots flashes.

On a real world multi-robot system comprised of 10 real robots, we demonstrated that robots do synchronize when they act as pulse-coupled oscillators. In a set of experiments, we simulated faults in one of the robots. In all the experiments, the faulty robot was correctly detected. The results indicate that our approach gives robots in a swarm the capacity to detect non-operational members of the swarm. In fact, in two of our experiments, a real fault occurred in one of the robots. In both cases, some of the other robots correctly detected this fault.

In the near future, we expect to see an increased effort going into the research of new approaches to fault detection and fault tolerance for autonomous robots. Robots need to be both safe and dependable before they can enter our homes and before they can be entrusted with mission and business-critical tasks. With the research covered in this thesis, we have taken a step in this direction: we have proposed and demonstrated new methods for endogenous and exogenous fault detection, respectively, and our research and conclusions have been supported by experiments conducted on real robots.

Appendix

APPENDIX A

A.1 Software Architecture for Fault Detection based on Fault Injection and Learning

In this appendix, we present the software architecture that allows us to collect training data and to inject faults without changing a control program for which we want to synthesize a fault detector (see Chapter 4 for more on fault injection and learning). The architecture is enabled by the Common Interface. As described in Section 1.3.3, the Common Interface is an encapsulation of the *s-bot* API that allows a control program to run in simulation and on real robots without change. In the following, we illustrate the concept of the Common Interface and why it is particularly useful when we want to collect data, inject faults and detect faults.

If we do not use the Common Interface, sensors are read and control signals are sent to the robot's actuators through the *s-bot* API functions on a real *s-bots*. The prototypes for two of the API functions are shown below:

```
// Defined in ‘‘sbot.h’’
...
/** Set the speed of the tracks.
    left:  Left motor speed. [-127;127].
    right: Right motor speed. [-127;127].
           One unit corresponds to 2.8 mm/s.
*/
void setSpeed(int left, int right);
...
...
/** Read the value of all proximity sensors. \\
    sensors: Array of distances to objects. For each value, 20
    is 15 cm, 50 is 5 cm, 1000 is 1 cm, 3700 is 0.5 cm. [0;8191]
*/
void getAllProximitySensors(unsigned int sensors[15]);
...
```

The functions are declared in C. The corresponding methods in the Common Interface implemented in C++ are shown below:

```
// Defined in ‘‘ci_sbot.h’’

class CCISbot
{
public:
  ...
  ...
  virtual void SetSpeed(int left, int right);
  ...
  ...
  virtual void GetAllProximitySensors(unsigned int sensors[15]);
}
```

The parameters `left`, `right`, and `sensors[15]` have the same meaning in the Common Interface as in the *s-bot* API. As it can be seen, the Common Interface and the real *s-bot* API are nearly identical, except for the fact that the Common Interface lives in a class (*`CCISbot`*) whereas the *s-bot* API is a set of C functions. When a control program written for the Common Interface is started it is given an instance of a specialization of *`CCISbot`*. On real *s-bot* this instance is a specialization in which all methods of the Common Interface call their *s-bot* API counterpart, for instance:

```
void CCIRealSbot::SetSpeed(int left, int right)
{
  setSpeed(left,right);  // Call the real s-bot API
}
```

`CCIRealSbot` is a specialization of the abstract common interface class *`CCISbot`*. The specialization of *`CCISbot`* for our simulator, TwoDee, implements the method `SetSpeed` in a different way:

```
void CCITwoDeeSbot::SetSpeed(int left, int right)
{
  CWheelsActuator* pcWheelsActuator =  m_pcSbot->GetActuator(ACTUATOR_WHEELS);
  if (pcWheelsActuator == NULL)
  {
      // If no wheels actuator is present, we add a noisy one by default:
      pcWheelsActuator = new CWheelsActuator("WheelsActuator",
                                              m_pcTwoDeeSbot);
      pcWheelsActuator = new CNoisyActuator("NoisyWheelsActuator",
                                            m_pcTwoDeeSbot,
                                            pcWheelsActuator, 0.05);
      m_pcTwoDeeSbot->AddActuator(pcWheelsActuator);
  }
  pcWheelsActuator->SetOutput(0, (float) (left + CI_MAX_SBOT_SPEED) /
                                         (CI_MAX_SBOT_SPEED * 2));
  pcWheelsActuator->SetOutput(1, (float) (right + CI_MAX_SBOT_SPEED) /
                                         (CI_MAX_SBOT_SPEED * 2));
}
```

In the implementation of the Common Interface for TwoDee, the method translates any arguments and return values between the representation used by a control program and TwoDee and vice versa as in the above example.

A control program written for the Common Interface reads sensors and controls the robot through the methods specified in the interface *`CCISbot`* and does thus not depend on whether the underlying implementation is an instance of `CCIRealSbot` or `CTwoDeeCommonInterfaceSbot` (or some other specialization). In the next section, we demonstrate how this allows for a *layered* architecture, in which the data recorder and the fault injection layers can be implemented in an elegant manner.

A.1.1 The Common Interface and Layers

We have implemented a specialization of the *`CCISbot`* called `CCISbotLayer`. A layer forwards the calls it gets from above (with *above* meaning closer to the control program) to some specialization of the *`CCISbot`* below (with *below* meaning closer to the hardware). As illustrated in the following the example, this concept is really simple:

```
...
void CCISbotLayer::SetSpeed(int left, int right)
{
  m_pcCISbotBelow->SetSpeed(left, right);
}
..
..
unsigned int CCISbotLayer::GetProximitySensor(unsigned int sensor)
{
    return m_pcCISbotBelow->GetProximitySensor(sensor);
}
...
```

In the `CCISbotLayer`, all methods are implemented in this way: all calls are forwarded to another layer below and the values returned from below are forwarded to the layer above. The layer itself does not add, remove or alter any data. However, specializations of the `CCISbotLayer` class can add functionality such as record the speeds to the left and right treel during data collection. Alternatively, the fault injection module can intercept and propagate a fault-dependent speed to one or both of the treels (in case a simulated fault is present) whenever a control program tries to set the speeds of the treels. Figure A.1 illustrates the relationship among the *`CCISbot`* and its specializations in UML notation.

The control signal recorder (`CCIReflectionSbot`) and the fault injector (`CCISWIFISbot`) both inherit from *`CCISbot`*. The `CCIReflectionSbot` records the control signals sent by the control program to the robot's actuators. The signals are recorded so that training data can be collected and saved and so that a fault detector can read them in order to classify the state of the robot. The control program is unaware that it is sending control signals and reading sensory data through an instance of the `CCIReflectionSbot` class. Thus, in order to collect training data, we do not need to modify a control program.

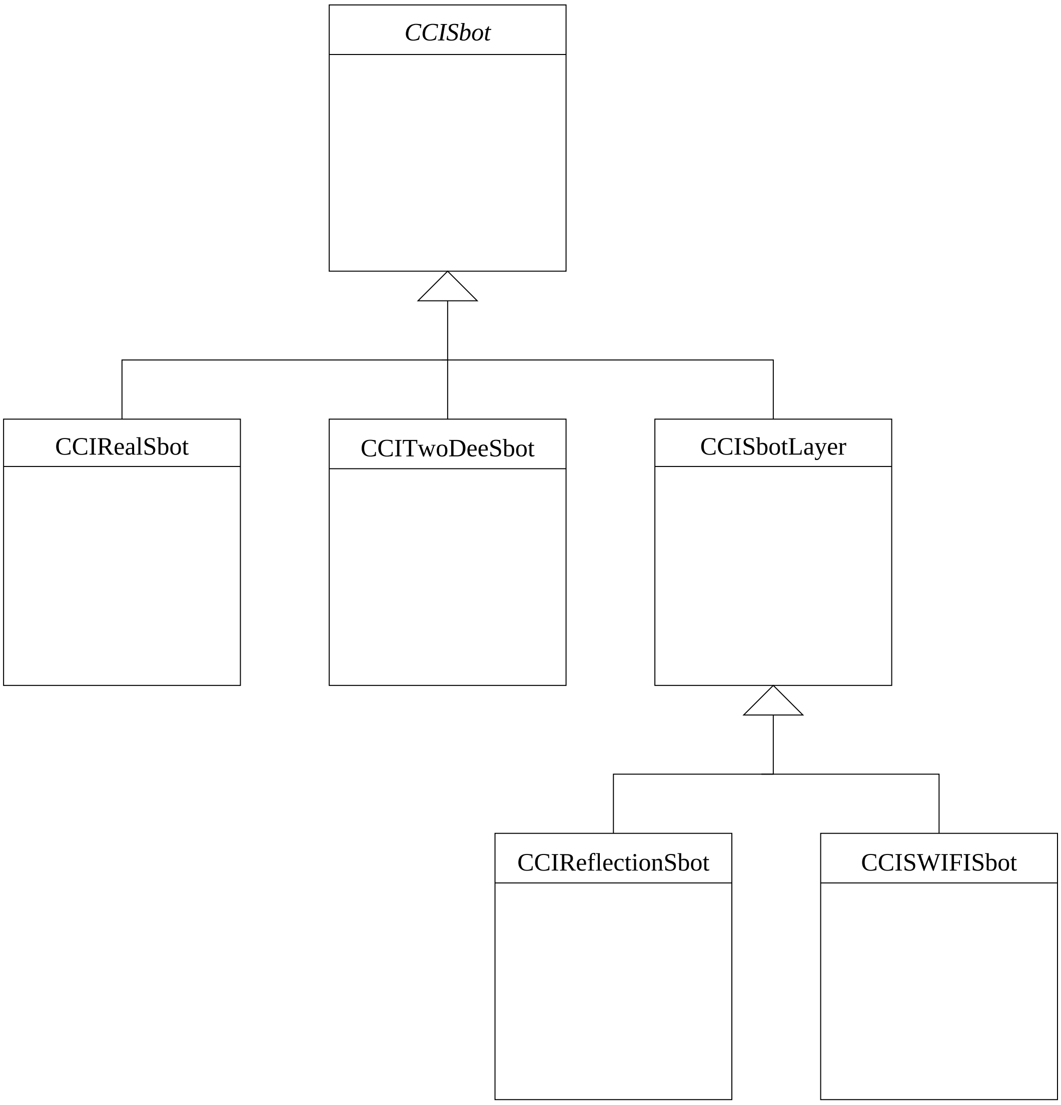

CCISbot
CCIRealSbot
CCITwoDeeSbot
CCISbotLayer
CCIReflectionSbot
CCISWIFISbot

Figure A.1: The Common Interface *s-bot* class `CCISbot` and its specializations

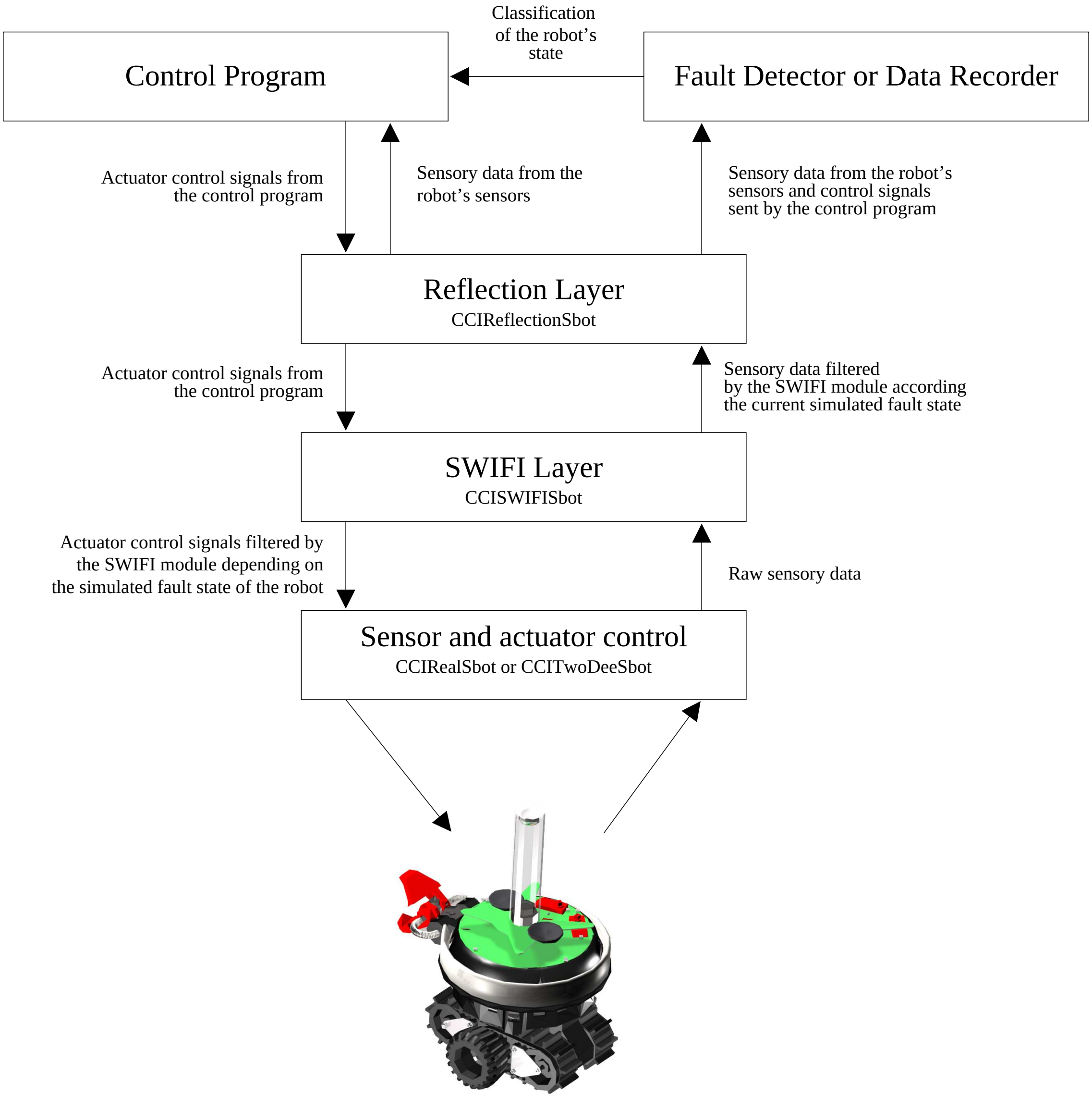

Classification of the robot's state
Control Program
Fault Detector or Data Recorder
Actuator control signals from the control program
Sensory data from the robot's sensors
Sensory data from the robot's sensors and control signals sent by the control program
Reflection Layer
CCIReflectionSbot
Actuator control signals from the control program
Sensory data filtered by the SWIFI module according the current simulated fault state
SWIFI Layer
CCISWIFISbot
Actuator control signals filtered by the SWIFI module depending on the simulated fault state of the robot
Raw sensory data
Sensor and actuator control
CCIRealSbot or CCITwoDeeSbot

Figure A.2: The relationships between the different layers and modules for fault detection based on fault injection and learning.

Figure A.2 illustrates the relationships among the various layers and modules. The control program is situated at the very top together with either a fault detector or a data collection module. During the training data collection phase, the data collection module is responsible for acquiring sensory data and actuator control signals. When a fault detector has been trained, the data collection module is no longer needed and it is replaced by the fault detector. In each control cycle, the control program acquires and processes readings from some of the robot's sensors and subsequently sends signals to the robot's actuators such as the treels. Control signals pass down through the Reflection layer that records them. The Reflection layer in turn forwards the signals to the SWIFI layer which is responsible for injecting and simulating faults. If a fault is being simulated, the SWIFI layer modifies that control signals according to the type of fault before the control signals are passed on to either the real robot Common Interface implementation or to the TwoDee Common Interface implementation. Sensory data are requested by the control program pass from the hardware (or a simulator) and up to the control program. First through the SWIFI layer which, in case a fault has been injected in the sensor, modifies the data according to the fault currently being simulated. If no fault is simulated, the sensory data passes through the SWIFI layer unmodified. The Reflection layer forwards the sensory data from the SWIFI layer to the control program (or to the fault detector). Below we provide illustrative examples of how the SWIFI layer and the Reflection layer are implemented.

SWIFI Layer

The SWIFI layer (`CCISWIFISbot`) is responsible for injecting and simulating faults. The SWIFI layer is located right on top of the layer that communicates with either the real hardware or with TwoDee (see Figure A.2).

Sensory data from the hardware/simulator and control signals to a robot's actuators all pass through the SWIFI layer. The SWIFI layer thus can modify the information that passes though it whenever a fault is simulated. The implementations in the SWIFI layer of the two methods for setting the speed of the treels and for getting a proximity sensor reading, respectively, are shown below:

```
// An example of how faults in an actuator is implemented:
void CCIFaultySbot::SetSpeed(int n_left, int n_right)
{
  if (m_eTotalCrash != FAULT_TYPE_NONE)
  {
      m_pcCISbotBelow->SetSpeed(0, 0);
  } else {
      int nLeftSpeed = SetValueDependingOnFaultStatus(LEFT_TREEL);
      int nRightSpeed = SetValueDependingOnFaultStatus(RIGHT_TREEL);
      m_pcCISbotBelow->SetSpeed(nLeftSpeed, nRightSpeed);
  }
}
...
// An example of how faults in a sensor is implemented:
unsigned int CCISWIFISbot::GetProximitySensor(unsigned sensor)
{
    if (m_eProximitySensors && ((1 << sensor) & m_unProximitySensorMask))
    {
        return SetValueDependingOnFaultStatus(PROXIMITY_SENSOR);
    }
    return m_pcCISbotBelow->GetProximitySensor(sensor);
}
```

Faults are injected based on *fault events*. Fault events can either be generated probabilistically on the fly, but in the experiments described in Chapter 4 they were loaded from an external file. In this way, the training data collected can be correlated with the state of a robot off-line. A fault event is composed of three pieces of information: the time (from the start of an experiment) at which the fault state of the robot should change, the affected component (for instance the left treel) and the type of fault (for instance *stuck-at-one*).

Reflections Layer

The Reflection layer is responsible for storing the control signals sent by the control program so that they can be saved for training purposes and for a fault detector once it has been trained. For instance, the method `SetSpeed` is part of *`CCISbot`* (Common Interface) and it is the method called by control programs to set the speeds of the left and right treel, respectively. The Reflection layer records speeds set by the control program so that they can be retrieved later by the data collection module or by a fault detector:

```
...
void CCIReflectionSbot::SetSpeed(int left, int right)
{
  m_nLeftSpeed  = left;
  m_nRightSpeed = right;
  m_pcCISbotBelow->SetSpeed(m_nLeftSpeed, m_nRightSpeed);
}
...
void CCIReflectionSbot::ReflectionGetSetSpeed(int* pn_left, int* pn_right)
{
  (*pn_left)  = m_nLeftSpeed;
  (*pn_right) = m_nRightSpeed;
}
```

The method `ReflectionGetSetSpeed` is used exclusively by the data collection module and by fault detectors to monitor the actuator control signals sent by the control program to a robot's treels. Because of the Reflection layer and the control program does not need to be changed in order to keep the data collection module or a fault detector informed about its actions. The Reflection layer provides this functionality transparently.

A.2 Summary

In this appendix, we presented the software architecture for our fault detection through fault injection and learning approach. We showed how the Common Interface enables us to rely on a layered architecture. We have implemented two layers: the SWIFI layer, which is responsible for injecting and simulating faults, and the Reflection layer, which records the actuator control signals sent by the control program.

One of the main benefits of using the Reflection layer and the SWIFI layer is that fault detection capabilities can be added to an existing controller without changing the controller (expect for the addition of logic for handling detected faults): the recording of actuator control signals and fault injection is handled outside of the main control program. Furthermore, other layers can be added in the future: one could for instance envision a logging layer that would enable a robot's actions and sensory data to be sent in real-time to a workstation. Another layer that could ease the deployment of the same control on different robots is an *Alignment* layer, that is, a layer which transforms actuator control signals and sensory data in a robot-dependent manner to minimize individual differences between robots from the point of view of control programs.

List of Figures

1.1 Two examples of robotic entities self-assembled into morphologies appropriate for the task. Left: A connected robotic entity crosses a trough. A line formation is well-suited to this task, since it allows the entity to stretch further and requires only a minimum number of robots to be suspended over the trough at any one time. Right: A more dense structure provides greater stability for rough terrain navigation. 6

3.1 The *s-bot*: An autonomous, mobile robot capable of self-assembly. Processor: 400 MHz XScale CPU, operating system: Linux, weight: ∼700 g, battery allows for ∼2 h of operation between recharges. 22

3.2 Illustration of the gripper-based connection mechanism: One *s-bot* grasping the transparent LED-right of another *s-bot*. 23

3.3 A: an *s-bot* with all its LEDs off. B: an *s-bot* with its red LEDs illuminated. . . 23

3.4 An image captured by a robot's omni-directional camera and the processing steps to obtain information about the LEDs of nearby robots. A: The captured image. B: After color segmentation with indications of the distance estimates from the robot that captured the image to some of the LEDs detected. 24

4.1 The four steps of our methodology for obtaining and evaluating fault detectors based on fault injection and learning. 30

4.2 The fault detection module monitors the sensory data read by the control program and the consequent control signals sent to the actuators. The fault detection module is passive and does not interact with the robot hardware or the control program. The SWIFI Module facilitates fault injection (see text). 31

4.3 An illustration of a fault detection module based on a TDNN. The current control program input and output (CPIO) is stored in the tapped delay-line and the activations of the neurons in the logical input groups are set according to the current and past CPIOs. In the example illustrated, there are 3 input groups and the input group distance d is 4. 34

4.4 Description of the three setups: *find perimeter*, *follow the leader*, and *connect to s-bot*. For each setup a list of sensors used and the control cycle period for the controllers are shown. The number in brackets after each sensor listed corresponds to the number of input values the sensor provides to the fault detector at each control cycle. 37

4.5 An example of the output of a trained TDNN during a run. The dotted line shows the optimal output. At control cycle 529 a fault is injected. Five different thresholds are indicated, 0.10, 0.25, 0.50, 0.75, and 0.90, and a false positive for threshold 0.50 is shown at control cycle 304 (the output has a value greater than 0.50 *before* the fault was injected at control cycle 529). The latency for a threshold is the number of control cycles from the moment the fault is injected till the moment the output value of the TDNN becomes greater than the threshold. In the example above, the latency for threshold 0.75 is 43 control cycles because the output of the TDNN reaches 0.75 only at control cycle 562, that is, 43 control cycles after the fault was injected. 39

4.6 Box-plot of the latencies observed in 20 evaluation runs in the *find perimeter* setup using fault detectors with input group distances from 1 to 10. Results are shown for the thresholds 0.10, 0.25, 0.50, 0.75, and 0.90. Each box comprises observations ranging from the first to the third quartile. The median is indicated by a horizontal bar, dividing the box into the upper and lower part. The whiskers extend to the farthest data points that are within 1.5 times the interquartile range. Outliers are shown as dots. The results show that the input group distance does not have a major influence on the latency of a fault detector, while larger thresholds yield longer latencies. 41

4.7 Box-plot of the number of false positives observed in 20 evaluation runs in the *find perimeter* setup using fault detectors with input group distances from 1 to 10. Results are shown for the thresholds 0.10, 0.25, 0.50, 0.75, and 0.90. For low input group distances, 1 and 2 in particular, the fault detector in general detects a large number of false positives, while no clear trend is observed for fault detectors with input group distances above 4. See the caption of Figure 4.6 for details on box-plots. 42

4.8 Box-plot of the latencies and number of false positives observed during 20 evaluation runs in the *follow the leader* setup for different thresholds and an input group distance of 5. See the caption of Figure 4.6 for details on box-plots. . . . 45

4.9 Box-plot of the latencies and number of false positives observed during 20 evaluation runs in the *connect to s-bot* setup, for different thresholds and an input group distance of 5. See the caption of Figure 4.6 for details on box-plots. . . . 46

4.10 Box-plot of false positives results observed in 20 runs in each of the three setups using fault detectors in which the output of the TDNN is used directly and fault detectors in which the output is smoothed by computing the moving average over 25 control cycles. A threshold of 0.75 was used for all fault detectors. False positives were only observed during one run in the *follow the leader* setup when the TDNN's output was smoothed. The run is not shown in the figure since it is out of scale (164 false positives were detected during this run). See the caption of Figure 4.6 for details on box-plots. 47

4.11 Box-plot of latency results observed in 20 runs in each of the three setups using fault detectors in which the output of the TDNN is used directly and fault detectors in which the output is smoothed by computing the moving average over 25 control cycles. A threshold of 0.75 was used for all fault detectors. See the caption of Figure 4.6 for details on box-plots. 48

4.12 Box-plot of latency results for fault detectors trained to detect faults in the treels only (from Section 4.4.1), in the ground and light sensors only, and a fault detector trained to detect faults in both the ground and light sensors and the treels. In each case, the fault detector was evaluated on 20 runs in which faults corresponding to those the fault detector was trained to detect were injected. All three fault detectors were configured to use the output of the TDNN directly and to use a threshold of 0.75. See the caption of Figure 4.6 for details on box-plots. 50

4.13 Two additional setups for the *connect to s-bot* controller used to evaluate if a fault detector can generalize over variations of the task. 52

4.14 Box-plot of the latencies and the number of false positives observed during 20 evaluation runs using a fault detector trained on data from a total of 60 runs in all three variations of the *connect to...* setup. Results are shown for moving average window lengths of 1 (equivalent to using the output of the TDNN directly) and 25. A threshold of 0.90 was used. See the caption of Figure 4.6 for details on box-plots. 53

4.15 The software architecture for endogenous and exogenous fault detection based on fault injection and learning. 54

4.16 Box-plot of the performance results in terms of latency and number of false positives observed in 20 evaluation runs for the *follower* performing endogenous fault detection and for the *leader* performing exogenous fault detection during the same runs. For both sets of results, the output of the TDNN is used directly and compared against a threshold of 0.75. See the caption of Figure 4.6 for details on box-plots. 55

4.17 Box-plot of the performance results in terms of the number of false positives observed in 20 evaluation runs for the *follower* performing endogenous fault detection and for the *leader* performing exogenous fault detection during the same runs for different lengths of the moving window. For both sets of results, the moving average is compared against a threshold of 0.75. See the caption of Figure 4.6 for details on box-plots. 56

4.18 An example of three robots, A, B and C, in which robot A has experienced a fault while within the perceptual range of B but outside of the perceptual range of C. The perceptual ranges of B and C, respectively, are indicated by circles. . 59

5.1 An example of two pulse-coupled oscillators. Both oscillators increase at a constant rate until the threshold is reached or until one oscillator observes that the other one fires. When an oscillator's activation reaches the threshold, the oscillator fires. If one oscillator observes the other's firing, it increases its own state by ϵx, where ϵ is the pulse-coupling constant and x the activation of the oscillator. 66

5.2 An example of the evolution of activations sampled every T in 25 mobile robots over the course of 10 minutes. One cross represents the activation for a single robot at the corresponding time. 69

5.3 Synchronization rate in groups of 10 to 100 simulated robots. Each bar summarizes 100 runs and error-bars denote the standard deviation. The density was 8 robots/m^2 and a coupling constant of $\epsilon = 0.1$ was used in all runs. 69

5.4 Synchronization rate in a group of 50 simulated robots at different densities. Each bar summarizes 100 runs and error-bars denote the standard deviation. A coupling constant of $\epsilon = 0.1$ was used in all runs. Due to the limited sensory range of the *s-bots* (up to 50 cm) experiments with static robots at a density of 2 robots/m^2 were not conducted. At this density, the interaction graph is almost never connected when the robots are distributed randomly. 70

5.5 Synchronization rate a group of 50 simulated robots for coupling constants $0.01, 0.02, 0.05, 0.10, 0.20, 0.50$. Each bar summarizes 100 runs and error-bars denote the standard deviation. The density was 6 robots/m^2. 70

5.6 An example of a flash wave in a group of static robots. 71

5.7 A photo of synchronized robots flashing at the same time. 73

5.8 Four possible scenarios. See text. 74

5.9 Average percentage of the control cycles spent in the suspicious state over intervals of 15 s during a run with 50 simulated robots in a 2.5 m x 2.5 m arena. The robots were not initially synchronized. 76

6.1 An example of a heterogeneous swarm of robots: *s-bots* navigate on the ground using a differential drive systems, while *eye-bots* are quad-rotor flying robots with long-range sensing capabbilities. 87

A.1 The Common Interface *s-bot* class `CCISbot` and its specializations 94

A.2 The relationships between the different layers and modules for fault detection based on fault injection and learning. 95

List of Tables

4.1 Median latencies during 20 evaluation runs in the *find perimeter* setup with fault detectors using input groups distances from 1 to 10 and for the thresholds: 0.10, 0.25, 0.50, 0.75, and 0.90. 43

4.2 Median number of false positives observed during 20 evaluation runs in the *find perimeter* setup with fault detectors using input groups distances from 1 to 10 and for the thresholds: 0.10, 0.25, 0.50, 0.75, and 0.90. 44

4.3 Number of undetected faults observed during 20 evaluation runs in the *find perimeter* setup, for five different thresholds, using input group distances from 1 to 10. 44

4.4 Number of undetected faults observed during 20 evaluation runs in the *follow the leader* and *connect to s-bot* setups, for different thresholds, using an input group distance of 5. 46

5.1 Synchronization rate for 10 real robot. 72

5.2 Fault reaction time results on real robots 78

Bibliography

C. Ampatzis, E. Tuci, V. Trianni, A. L. Christensen, and M. Dorigo. Evolving autonomous self-assembly in homogeneous robots. Technical Report TR/IRIDIA/2008-04, IRIDIA, Université Libre de Bruxelles, Belgium, 2008. Submitted to Artificial Life.

J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. Fabre, J. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, 1990.

I. Ashokaraj, A. Tsourdos, P. Silson, and B. A. White. Sensor based robot localisation and navigation: using interval analysis and unscented Kalman filter. In *Proceedings of the 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS'04*, pages 64–70. IEEE Press, Las Vegas, NV, 2004.

O. Babaoglu, T. Binci, M. Jelasity, and A. Montresor. Firefly-inspired heartbeat synchronization in overlay networks. In *Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems, SASO'07*, pages 77–86. IEEE Computer Society Press, Los Alamitos, CA, 2007.

R. D. Beer. On the dynamics of small continuous-time recurrent neural networks. *Adaptive Behavior*, 3(4):469, 1995.

E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, New York, NY, 1999.

E. Bonabeau, G. Theraulaz, J. L. Deneubourg, S. Aron, and S. Camazine. Self-organization in social insects. *Trends in Ecology & Evolution*, 12(5):188–193, 1997.

H. B. Brown, J. M. V. Weghe, C. A. Bererton, and P. K. Khasla. Millibot trains for enhanced mobility. *IEEE/ASME Transactions on Mechatronics*, 7(4):452–461, 2002.

J. Buck. Synchronous rhythmic flashing of fireflies. II. *Quarterly Review of Biology*, 63:265–289, 1988.

W. Burgard, M. Moors, D. Fox, R. Simmons, and S. Thrun. Collaborative multi-robot exploration. In *Proceedings of the 2000 IEEE International Conference on Robotics and Automation, ICRA'00*, volume 1, pages 476–481. IEEE Computer Society Press, Los Alamitos, CA, 2000.

L. Bury. Conception et implémentation en c++ d'un simulateur pour les robots e-puck et réalisation de tests de validation pour la cinématique de base. Technical Report - Master Thesis, IRIDIA, Université Libre de Bruxelles, Belgium, 2007.

E. Butterfield. The future of robots - tomorrow's domestic help at your service. *PC World*, October 2, 2006.

S. Camazine, N. R. Franks, J. Sneyd, E. Bonabeau, J.-L. Deneubourg, and G. Theraula. *Self-Organization in Biological Systems*. Princeton University Press, NJ, 2001.

R. Canham, A. Jackson, and A. Tyrrell. Robot error detection using an artificial immune system. In *Proceedings of NASA/DoD Conference on Evolvable Hardware, 2003*, pages 199–207. IEEE Computer Society, Washington, DC, 2003.

Y. U. Cao, A. S. Fukunaga, and A. Kahng. Cooperative mobile robotics: Antecedents and directions. *Autonomous Robots*, 4(1):7–27, 1997.

J. Carlson, R. R. Murphy, and A. Nelson. Follow-up analysis of mobile robot failures. In *Proceedings of the 2004 IEEE International Conference on Robotics and Automation, ICRA'04*, pages 4987–4994. IEEE Computer Society Press, Los Alamitos, CA, 2004.

A. Castano, W.-M. Shen, and P. Will. CONRO: Towards deployable robots with inter-robots metamorphic capabilities. *Autonomous Robots*, 8(3):309–324, 2000.

A. L. Christensen. Efficient neuro-evolution of hole-avoidance and phototaxis for a swarm-bot. Technical Report TR/IRIDIA/2005-14, IRIDIA, Université Libre de Bruxelles, Belgium, 2005. DEA Thesis.

A. L. Christensen and M. Dorigo. Evolving an integrated phototaxis and hole-avoidance behavior for a swarm-bot. In L. M. Rocha, L. S. Yaeger, M. A. Bedau, D. Floreano, R. L. Goldstone, and A. Vespignani, editors, *Artificial Life X: Proceedings of the Tenth International Conference on the Simulation and Synthesis of Living Systems*, pages 248–254. MIT Press, Cambridge, MA, 2006a.

A. L. Christensen and M. Dorigo. Incremental evolution of robot controllers for a highly integrated task. In S. Nolfi, G. Baldassarre, R. Calabretta, J. Hallam, D. Marocco, J.-A. Meyer, O. Miglino, and D. Parisi, editors, *From Animals to Animats 9: 9th International Conference on Simulation of Adaptive Behavior, SAB 2006*, volume 4095 of *Lecture Notes in Artificial Intelligence*, pages 473–484. Springer Verlag, Berlin, Germany, 2006b.

A. L. Christensen, R. O'Grady, and M. Dorigo. A mechanism to self-assemble patterns with autonomous robots. In *Proceedings of the 9th European Conference on Artificial Life (ECAL2007)*, pages 716–725. Springer Verlag, Berlin, Germany, 2007a.

A. L. Christensen, R. O'Grady, and M. Dorigo. Morphogenesis: Shaping swarms of intelligent robots. In *AAAI-07 Video Proceedings*. AAAI Press, 2007b.

A. L. Christensen, R. O'Grady, and M. Dorigo. Morphology control in a self-assembling multi-robot system. *IEEE Robotics & Automation Magazine*, 14(4):18–25, 2007c.

A. L. Christensen, R. O'Grady, and M. Dorigo. SWARMORPH-script: A language for arbitrary morphology generation in self-assembling robots. *Swarm Intelligence*, 2008. In press.

D. Clouse, C. Giles, B. Horne, and G. Cottrell. Time-delay neural networks: Representation and induction of finite-state machines. *IEEE Transactions on Neural Networks*, 8:1065–1070, 1997.

N. Cristianini and J. Shawe-Taylor. *An introduction to Support Vector Machines*. Cambridge University Press, Cambridge, UK, 2000.

R. Damoto, A. Kawakami, and S. Hirose. Study of super-mechano colony: concept and basic experimental set-up. *Advanced Robotics*, 15(4):391–408, 2001.

R. Dearden, F. Hutter, R. Simmons, S. Thrun, V. Verma, and T. Willeke. Real-time fault detection and situational awareness for rovers: Report on the Mars technology program task. In *Proceedings of IEEE Aerospace Conference*, volume 2, pages 826–840. IEEE Computer Society Press, Los Alamitos, CA, 2004.

L. Devroye, L. Györfi, and G. Lugosi. *A Probabilistic Theory of Pattern Recognition*. Springer Verlag, New York, NY, 1996.

M. B. Dias, M. B. Zinck, R. M. Zlot, and A. Stentz. Robust multirobot coordination in dynamic environments. In *Proceedings of IEEE Conference on Robotics and Automation, ICRA'04*, volume 4, pages 3435 – 3442. IEEE Press, Piscataway, NJ, 2004.

W. Dixon, I. Walker, and D. Dawson. Fault detection for wheeled mobile robots with parametric uncertainty. In *Proceedings of the 2001 IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, volume 2, pages 1245–1250. IEEE Press, Piscataway, NJ, 2001.

R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. Wiley, New York, NY, 2nd edition, 2000.

J. Elson and D. Estrin. Time synchronization for wireless sensor networks. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, pages 1965–1970. IEEE Computer Society, Washington, DC, 2001.

S. Forrest, A. Perelson, L. Allen, and R. Cherukuri. Self-nonself discrimination in a computer. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, volume 212, pages 202–212. IEEE Computer Society Press, Los Alamitos, CA, 1994.

M. Fujita, M. Veloso, W. Uther, M. Asada, H. Kitano, V. Hugel, P. Bonnin, J.-C. Bouramoué, and P. Blazevic. Vision, strategy, and localization using the Sony legged robots at RoboCup-98. *AI Magazine*, 21(1):47–56, 2000.

T. Fukuda, M. Buss, H. Hosokai, and Y. Kawauchi. Cell structured robotic system CEBOT: control, planning and communication methods. *Robotics and Autonomous Systems*, 7(2-3): 239–248, 1991.

B. Gates. A robot in every home. *Scientific American*, December 2006.

B. P. Gerkey and M. J. Matarić. Pusher-watcher: An approach to fault-tolerant tightly-coupled robot coordination. In *Proceedings of IEEE International Conference on Robotics and Automation, ICRA'02*, pages 464 – 469. IEEE Press, Piscataway, NJ, 2002a.

B. P. Gerkey and M. J. Matarić. Sold!: auction methods for multirobot coordination. *IEEE Transactions on Robotics and Automation*, 18(5):758–768, 2002b.

J. Gertler. *Fault Detection and Diagnosis in Engineering Systems*. CRC Press, Boca Raton, FL, 1998.

J. J. Gertler. Survey of model-based failure detection and isolation in complex plants. *IEEE Control Systems Magazine*, 8:3–11, 1988.

L. Glass. Synchronization and rhythmic processes in physiology. *Nature*, 410:277–284, 2001.

P. Goel, G. Dedeoglu, S. Roumeliotis, and G. Sukhatme. Fault detection and identification in a mobile robot using multiple model estimation and neural network. In *Proceedings of IEEE International Conference on Robotics and Automation, ICRA'00*, volume 3, pages 2302–2309. IEEE Computer Society Press, Los Alamitos, CA, 2000.

R. Gross, M. Bonani, F. Mondada, and M. Dorigo. Autonomous self-assembly in a swarm-bot. In K. Murase, K. Sekiyama, N. Kubota, T. Naniwa, and J. Sitte, editors, *Proceedings of the 3rd International Symposium on Autonomous Minirobots for Research and Edutainment (AMiRE 2005)*, pages 314–322. Springer Verlag, Berlin, Germany, 2005.

R. Gross, M. Bonani, F. Mondada, and M. Dorigo. Autonomous self-assembly in swarm-bots. *IEEE Transactions on Robotics*, 22(6):1115–1130, 2006a.

R. Gross and M. Dorigo. Evolution of solitary and group transport behaviors for autonomous robots capable of self-assembling. *Adaptive Behavior*, 2008a. In press.

R. Gross and M. Dorigo. Self-assembly at the macroscopic scale. *Proceedings of the IEEE*, 2008b. In press.

R. Gross, M. Dorigo, and M. Yamakita. Self-assembly of mobile robots—from swarm-bot to super-mechano colony. In *Proceedings of the 9th International Conference on Intelligent Autonomous Systems*, pages 487–496. IOS Press, Amsterdam, The Netherlands, 2006b.

R. Gross, E. Tuci, M. Dorigo, M. Bonani, and F. Mondada. Object transport by modular robots that self-assemble. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation*, pages 2558–2564. IEEE Computer Society Press, Los Alamitos, CA, 2006c.

I. Harvey, E. A. Di Paolo, R. Wood, M. Quinn, and E. Tuci. Evolutionary robotics: A new scientific tool for studying cognition. *Artificial Life*, 11(1-2):79–98, 2005.

I. Harvey, P. Husbands, and D. Cliff. Seeing the light: Artificial evolution, real vision. In *Proceedings of the Third International Conference on Simulation of Adaptive Behavior: From animals to animats 3*, pages 392–401. MIT Press, Cambridge, MA, 1994.

M. Hinchey, J. Rash, C. Rouff, and W. Truszkowski. NASA's swarm missions: the challenge of building autonomous software. *IT Professional*, 6:47–52, 2004.

S. Hirose, T. Shirasu, and E. F. Fukushima. Proposal for cooperative robot "Gunryu" composed of autonomous segments. *Robots and Autonomous Systems*, 17:107–118, 1996.

M. Hsueh, T. Tsai, and R. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

R. Isermann. Supervision, fault-detection and fault-diagnosis methods – An introduction. *Control Engineering Practice*, 5(5):639–652, 1997.

R. Isermann and P. Ballé. Trends in the application of model-based fault detection and diagnosis of technical processes. *Control Engineering Practice*, 5(5):709–719, 1997.

E. M. Izhikevich. Weakly pulse-coupled oscillators, FM interactions, synchronization, and oscillatory associative memory. *IEEE Transactions on Neural Networks*, 10(3):508–526, 1999.

N. Jakobi. Evolutionary robotics and the radical envelope-of-noise hypothesis. *Adaptive Behavior*, 6(2):325, 1997a.

N. Jakobi. Half-baked, ad-hoc and noisy: Minimal simulations for evolutionary robotics. In P. Husbands and I. Harvey, editors, *Proceedings of the Fourth European Conference on Artificial Life: ECAL97*, pages 348–357. MIT Press, Cambridge, MA, 1997b.

N. Jakobi. Evolutionary robotics and the radical envelope-of-noise hypothesis. *Adaptive Behavior*, 6(2):325–368, 1998.

N. Jakobi, P. Husbands, and I. Harvey. Noise and the reality gap: The use of simulation in evolutionary robotics. pages 704–720. Springer Verlag, Berlin, Germany, 1995.

F. V. Jensen. *Introduction to Bayesian Networks*. Springer Verlag, New York, NY, 1996.

S. J. Julier and J. K. Uhlmann. A new extension of the Kalman filter to nonlinear systems. In *Proceedings of the 11th International Symposium on Aerospace/Defense Sensing, Simululation and Controls*, volume 3, pages 182–193. SPIE, Bellingham, WA, 1997.

R. Kalman. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, 82(1):35–45, 1960.

Y. Kawauchi, M. Inaba, and T. Fukuda. A principle of distributed decision making of celluar robotic system (CEBOT). In *Proceedings of the 1993 IEEE International Conference on Robotics and Automation, ICRA'93*, pages 833–838. IEEE Press, Piscataway, NJ, 1993.

A. Kochan. A bumper year for robots. *Industrial Robot: An International Journal*, 32:201–204, 2005.

R. Kurzweil. *The Singularity is Near*. Viking Adult, New York, NY, 2005.

J. J. Leonard and H. F. Durrant-Whyte. Mobile robot localization by tracking geometric beacons. *IEEE Transactions on Robotics and Automation*, 7(3):376–382, 1991.

U. Lerner, R. Parr, D. Koller, and G. Biswas. Bayesian fault detection and diagnosis in dynamic systems. In *Proceedings of the 7th National Conference on Artificial Intelligence*, pages 531–537. AAAI Press/The MIT Press, Cambridge, MA, 2000.

M. A. Lewis and K. H. Tan. High precision formation control of mobile robots using virtual structures. *Autonomous Robots*, 4(4):387–403, 1997.

P. Li and V. Kadirkamanathan. Particle filtering based likelihood ratio approach to fault diagnosis in nonlinear stochastic systems. *IEEE Transactions on Systems, Man and Cybernetics, Part C*, 31(3):337–343, 2001.

X. Li and L. E. Parker. Sensor analysis for fault detection in tightly-coupled multi-robot team tasks. In *Proceedings of the 2007 IEEE International Conference on Robotics and Automation, ICRA'07*, pages 3269–3276. IEEE Computer Society Press, Los Alamitos, CA, 2007.

D. Lucarelli and I. J. Wang. Decentralized synchronization protocols with nearest neighbor communication. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, pages 62–68. ACM Press, New York, NY, 2004.

S. Marsland, U. Nehmzow, and J. Shapiro. On-line novelty detection for autonomous mobile robots. *Robotics and Autonomous Systems*, 51(2-3):191–206, 2005.

M. J. Matarić, M. Nilsson, and K. Simsarian. Cooperative multi-robot box-pushing. In *Proceedings of the 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 556–561. IEEE Computer Society Press, Los Alamitos, CA, 1995.

M. K. McClintock. Menstrual synchrony and suppression. *Nature*, 229(5282):244–245, 1971.

J. McLurkin. Stupid robot tricks: A behavior-based distributed algorithm library for programming swarms of robots, 2004. Master Thesis, MIT.

J. McLurkin and J. Smith. Distributed algorithms for dispersion in indoor environments using a swarm of autonomous mobile robots. In *Proceedings of the 6th International Symposium on Distributed Autonomous Robotic Systems (DARS)*, pages 399–408. Springer, Tokoyo, Japan, 2004.

O. Miglino, H. H. Lund, and S. Nolfi. Evolving mobile robots in simulated and real environments. *Artificial Life*, 2(4):417–434, 1995.

R. E. Mirollo and S. H. Strogatz. Synchronization of pulse-coupled biological oscillators. *SIAM Journal on Applied Mathematics*, 50(6):1645–1662, 1990.

T. Misteli. The concept of self-organization in cellular architecture. *The Journal of Cell Biology*, 155(2):181–186, 2001.

F. Mondada, E. Franzi, and P. Ienne. Mobile robot miniaturization: A tool for investigation in control algorithms. In *Proceedings of the Third International Symposium on Experimental Robotics*, pages 501–513. Springer Verlag, Berlin, Germany, 1994.

F. Mondada, L. M. Gambardella, D. Floreano, S. Nolfi, J.-L. Deneubourg, and M. Dorigo. The cooperation of swarm-bots: Physical interactions in collective robotics. *IEEE Robotics & Automation Magazine*, 12(2):21–28, 2005.

K. Motomura, A. Kawakami, and S. Hirose. Development of arm equipped single wheel rover: Effective arm-posture-based steering method. *Autonomous Robots*, 18(2):215–229, 2005.

S. Murata, E. Yoshida, A. Kamimura, H. Kurokawa, K. Tomita, and S. Kokaji. M-TRAN: Self-reconfigurable modular robotic system. *IEEE-ASME Transactions on Mechatronics*, 7 (4):431–441, 2002.

Z. Néda, E. Ravasz, Y. Brechet, T. Vicsek, and A. L. Barabási. Self-organizing processes: The sound of many hands clapping. *Nature*, 403(6772):849–850, 2000.

S. Nolfi and D. Floreano. *Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines*. MIT Press/Bradford Books, Cambridge, MA, 2000.

S. Nouyan, R. Gross, M. Bonani, F. Mondada, and M. Dorigo. Group transport along a robot chain in a self-organised robot colony. In *Proceedings of the 9^{th} Int. Conf. on Intelligent Autonomous Systems*, pages 433–442. IOS Press, Amsterdam, The Netherlands, 2006.

S. Nouyan, R. Gross, M. Bonani, F. Mondada, and M. Dorigo. Teamwork in self-organised robot colonies. *IEEE Transactions on Evolutionary Computation*, 2008. In press.

R. O'Grady, A. L. Christensen, and M. Dorigo. Self-assembly and morphology control in a swarm-bot. In *Video Proceedings of the 2007 International Conference on Intelligent Robots and Systems*, pages 2551–2552. IEEE Computer Society, Los Alamitos, CA, 2007a.

R. O'Grady, A. L. Christensen, and M. Dorigo. SWARMORPH: Morphology control with a swarm of self-assembling robots. In *Workshop on Self-Reconfigurable Robots/Systems and Applications, IEEE/RSJ International Conference on Intelligent Robots and Systems, San Diego, CA*. 2007b. Unpublished manuscript.

R. O'Grady, A. L. Christensen, and M. Dorigo. SWARMORPH: Multi-robot morphogenesis using directional self-assembly. *IEEE Transactions on Robotics*, 2008a. Submitted.

R. O'Grady, R. Gross, A. L. Christensen, and M. Dorigo. Self-assembly strategies in a group of autonomous mobile robots. *Autonomous Robots*, 2008b. Submitted.

R. O'Grady, R. Gross, A. L. Christensen, F. Mondada, M. Bonani, and M. Dorigo. Performance benefits of self-assembly in a swarm-bot. In *Proceedings of the 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS'07)*, pages 716–725. IEEE Press, Las Vegas, NV, 2007c.

R. O'Grady, R. Gross, F. Mondada, M. Bonani, and M. Dorigo. Self-assembly on demand in a group of physical autonomous mobile robots navigating rough terrain. In M. S. Capcarrere, A. A. Freitas, P. J. Bentley, C. G. Johnson, and J. Timmis, editors, *Advances in Artificial Life: 8th European Conference, ECAL 2005*, volume 3630 of *Lecture Notes in Artificial Intelligence*, pages 272–281. Springer Verlag, Berlin, Germany, 2005.

L. E. Parker. ALLIANCE: an architecture for fault tolerant multirobot cooperation. *IEEE Transactions on Robotics and Automation*, 14(2):220–240, 1998.

R. Patton, F. Uppal, and C. Lopez-Toribio. Soft computing approaches to fault diagnosis for dynamic systems: A survey. In A. Edelmayer and C. Banyasz, editors, *Proceedings of 4th IFAC Symposium on Fault Detection supervision and Safety for Technical Processes*, volume 1, pages 298–311. Elsevier, Oxford, UK, 2000.

D. W. Payton, M. Daily, R. Estkowski, M. Howard, and C. Lee. Pheromone robotic. *Autonomous Robots*, 11(3):319–324, 2001.

C. S. Peskin. *Mathematical aspects of heart physiology*. Courant Institute of Mathematical Sciences, New York University, New York, 1975.

K. Pohl, M. C. Bartelt, J. de La Figuera, N. C. Bartelt, J. Hrbek, and R. Q. Hwang. Identifying the forces responsible for self-organization of nanostructures at crystal surfaces. *Nature*, 397 (6716):238, 1999.

A. Pollack. Technology; robot's future as a servant. *The New York Times*, April 9 1981.

L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

G. M. Ramíre Ávila. *Synchronization Phenomena in Light Controllered Oscillators*. PhD thesis, Université Libre de Bruxelles, Brussels, Belgium, Feburary 2004.

J. Roberts, T. Stirling, J. Zufferey, and D. Floreano. Quadrotor using minimal sensing for autonomous indoor flight. In *European Micro Air Vehicle Conference and Flight Competition (EMAV2007)*. DVD-ROM Proceedings, Toulouse, France, 2007.

I. Roman-Ballesteros and C. Pfeiffer. A framework for cooperative multi-robot surveillance tasks. In *Electronics, Robotics and Automotive Mechanics Conference, CERMA2006*, volume 2, pages 163–170. IEEE Computer Society, Los Alamitos, CA, 2006.

S. Roumeliotis, G. Sukhatme, and G. Bekey. Sensor fault detection and identification in a mobile robot. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, pages 1383–1388. IEEE Computer Society Press, Los Alamitos, CA, 1998.

M. Rubenstein, K. Payne, and P. Will. Docking among independent and autonomous CONRO self-reconfigurable robots. In *Proceedings of the 2004 IEEE International Conference on Robotics and Automation, ICRA'04*, volume 3, pages 2877–2882. IEEE Press, Piscataway, NJ, 2004.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by backpropagating errors. *Nature*, 323:533–536, 1986.

B. Salemi, M. Moll, and W.-M. Shen. SUPERBOT: A deployable, multi-functional, and modular self-reconfigurable robotic system. In *Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3636–3641. IEEE Press, Piscataway, NJ, 2006.

J. Seyfried, M. Szymanski, N. Bender, R. Estana, M. Thiel, and H. Worn. The I-SWARM Project: Intelligent small world autonomous robots for micro-manipulation. Springer Verlag, Berlin, Germany, 2005.

W.-M. Shen, M. Krivokon, H. Chiu, J. Everist, M. Rubenstein, and J. Venkatesh. Multimode locomotion for reconfigurable robots. *Autonomous Robots*, 20(2):165–177, 2006.

W.-M. Shen, P. Will, A. Galstyan, and C. M. Chuong. Hormone-inspired self-organization and distributed control of robotic swarms. *Autonomous Robots*, 17(1):93–105, 2004.

E. N. Skoundrianos and S. G. Tzafestas. Finding fault - fault diagnosis on the wheels of a mobile robot using local model neural networks. *IEEE Robotics & Automation Magazine*, 11 (3):83–90, 2004.

H. M. Smith. Synchronous flashing of fireflies. *Science*, 82(2120):151–152, 1935.

R. Smith and P. Cheeseman. On the representation and estimation of spatial uncertainty. *The International Journal of Robotics Research*, 5(4):56, 1986.

W. A. Snedden, M. D. Greenfield, and Y. Jang. Mechanisms of selective attention in grasshopper choruses: who listens to whom? *Behavioral Ecology & Sociobiology*, 43(1):59–66, 1998.

K. Støy. Using situated communication in distributed autonomous mobile robots. In *Proceedings of the 7th Scandinavian Conference on Artificial Intelligence*, pages 44–52. IOS Press, Amsterdam, The Netherlands, 2001.

S. H. Strogatz. From Kuramoto to Crawford: exploring the onset of synchronization in populations of coupled oscillators. *Physica D*, 143(1-4):1–20, 2000.

A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, Upper Saddle River, NJ, 2002.

M. Terra and R. Tinos. Fault detection and isolation in robotic manipulators via neural networks: A comparison among three architectures for residual analysis. *Journal of Robotic Systems*, 18(7):357–374, 2001.

V. Trianni and M. Dorigo. Self-organisation and communication in groups of simulated and physical robots. *Biological Cybernetics*, 95:213–231, 2006.

V. Trianni, T. H. Labella, and M. Dorigo. Evolution of direct communication for a swarm-bot performing hole avoidance. In M. Dorigo, M. Birattari, C. Blum, L. M. Gambardella, F. Mondada, and T. Stützle, editors, *Ant Colony Optimization and Swarm Intelligence – Proceedings of ANTS 2004 – Fourth International Workshop*, volume 3172 of *Lecture Notes in Computer Science*, pages 131–142. Springer Verlag, Berlin, Germany, 2004.

E. Tuci, C. Ampatzis, V. Trianni, A. L. Christensen, and M. Dorigo. Self-assembly in physical autonomous robots: the evolutionary robotics approach. In *Artificial Life XI, 11th Conference on the Simulation and Synthesis of Living Systems*, 2008. In press.

E. Tuci, R. Gross, V. Trianni, F. Mondada, M. Bonani, and M. Dorigo. Cooperation through self-assembly in multi-robot systems. *ACM Transactions on Autonomous and Adaptive Systems*, 1(2):115–150, 2006.

A. Tyrrell and G. Auer. Imposing a reference timing onto firefly synchronization in wireless networks. In *Proceedings of the 65th IEEE Conference on Vehicular Technology, VTC2007*, pages 222–226. IEEE Computer Society Press, Los Alamitos, CA, 2007.

A. Vemuri and M. Polycarpou. Neural-network-based robust fault diagnosis in robotic systems. *IEEE Transactions on Neural Networks*, 8(6):1410–1420, 1997.

V. Verma, G. Gordon, R. Simmons, and S. Thrun. Real-time fault diagnosis. *IEEE Robotics & Automation Magazine*, 11(2):56–66, 2004.

V. Verma and R. Simmons. Scalable robot fault detection and identification. *Robotics and Autonomous Systems*, 54(2):184–191, 2006.

A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K. Lang. Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37:328–339, 1989.

P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

G. Werner-Allen, G. Tewari, A. Patel, M. Welsh, and R. Nagpal. Firefly-inspired sensor network synchronicity with realistic radio effects. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, pages 142–153. ACM Press, New York, NY, 2005.

R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1, 1989.

A. F. T. Winfield, C. J. Harper, and J. Nembrini. Towards dependable swarms and a new discipline of swarm engineering. In *Swarm Robotics Workshop: State-of-the-art Survey*, pages 126–142. Springer Verlag, Berlin, Germany, 2005.

A. F. T. Winfield and J. Nembrini. Safety in numbers: fault-tolerance in robot swarms. *International Journal of Modelling, Identification and Control*, 1(1):30–37, 2006.

H. Woern, M. Szymanski, and J. Seyfried. The I-SWARM project. In *The 15th IEEE International Symposium on Robot and Human Interactive Communication, ROMAN 2006*, pages 492–496. IEEE Computer Society Press, Los Alamitos, CA, Sept. 2006.

M. Yamakita, Y. Taniguchi, and Y. Shukuya. Analysis of formation control of cooperative transportation of mother ship by SMC. In *Proceedings of the 2003 International Conference on Robotics and Automation, ICRA'03*, volume 1, pages 951–956. IEEE Computer Society Press, Los Alamitos, CA, 2003.

M. Yim, D. G. Duff, and K. D. Roufas. PolyBot: a modular reconfigurable robot. In *Proceedings of the 2000 IEEE International Conference on Robotics and Automation, ICRA'00*, volume 1, pages 514–520. IEEE Press, Piscataway, NJ, 2000.

M. Yim, K. Roufas, D. Duff, Y. Zhang, C. Eldershaw, and S. B. Homans. Modular reconfigurable robots in space applications. *Autonomous Robots*, 14(2-3):225–237, 2003.

M. Yim, W.-M. Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, and G. S. Chirikjian. Modular self-reconfigurable robot systems. *IEEE Robotics & Automation Magazine*, 14(1): 43–52, 2007.

E. H. Østergaard, K. Kassow, R. Beck, and H. H. Lund. Design of the ATRON lattice-based self-reconfigurable robot. *Autonomous Robots*, 21(2):165–183, 2006.