Programming and Managing Swarms of Mobile Robots: A Systemic Approach

vorgelegt von
Daniel Graff, M.Sc.
geb. in Hildesheim

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Odej Kao
Gutachter:	Prof. Dr. Hans-Ulrich Heiß
Gutachter:	Prof. Dr.-Ing. Reinhardt Karnapke
Gutachter:	Prof. Dr. César Augusto Fonticielha De Rose

Tag der wissenschaftlichen Aussprache: 22.02.2016

Berlin 2017
D 83

Abstract

Numerous heterogeneous devices—including stationary as well as mobile devices—exist that are equipped with a variety of sensors and actuators. This enables new kinds of applications, e.g., applications for monitoring the environment and initiating early warnings in case of natural catastrophes, applications for exploration missions as well as applications for monitoring structural integrity of buildings.

The design of the resulting cyber-physical systems is a challenging task. Distribution, concurrency and synchronization together with real space and time are challenging aspects that need to be addressed and carefully treated when implementing such systems. In order to relieve programmers from error-prone properties of cyber-physical systems and facilitate application development by enabling the programmer to solely focus on the program's intention, this thesis presents an approach of a cyber-physical operating system.

The thesis addresses the following aspects: in order to program applications, a suitable programming abstraction is required that enables to incorporate real space and time into a programming language and allows to bind program code to space and time.

Executing such applications requires a suitable runtime system. The design should follow a modular system architecture in order to achieve a simple mechanism to add new components or exchange active ones. This way, new devices featuring new capabilities can be integrated with less effort.

Allowing programmers to have access physical space and time requires a scheduling that considers—besides time—also physical space. The scheduling has to address both orthogonal dimensions inevitably resulting in a higher complexity. Coordinated movement for mobile devices becomes necessary.

Zusammenfassung

Es existiert eine Vielzahl heterogener, stationärer sowie mobiler Geräte, die mit unterschiedlichen Sensoren und Aktuatoren bestückt sind. Neuartige Anwendungen wie Frühwarnsysteme für Naturkatastrophen, Erkundungsmissionen oder auch Überwachungen von strukturellen Veränderungen in Bauwerken werden dadurch ermöglicht.

Das Design der daraus resultierenden cyber-physischen Systeme bringt eine Vielzahl an Forschungsfragen und Herausforderungen mit sich. Verteiltheit, Nebenläufigkeit und Synchronisation stellen zusammen mit der realen Raum-Zeit Herausforderungen dar, welche bei der Implementierung eines solchen Systems sorgsam behandelt werden müssen. Um den Programmierer von fehlerträchtigen Eigenschaften cyber-physischer Systeme zu entlasten und gleichzeitig die Anwendungsentwicklung zu vereinfachen, indem der Programmierer sich allein auf die Intention des Programms konzentrieren kann, wird in dieser Arbeit ein Betriebssystem für cyber-physische Systeme vorgestellt.

In der Arbeit werden die folgenden Themen bearbeitet: Die Anwendungsentwicklung erfordert eine geeignete Programmierabstraktion, welche Raum-Zeit-Bezüge in Programmiersprachen ermöglicht und dadurch das Binden von Raum-Zeit-Parametern an Programmteile realisiert.

Für die Ausführung solcher Programme wird die Entwicklung eines entsprechenden Laufzeitsystems notwendig. Das Design sollte einer modularen Systemarchitektur folgen, welche einfaches Hinzufügen von neuen Komponenten oder auch den Austausch von aktiven Komponenten auf einfache Weise ermöglicht. So können neuartige Geräte, welche neue Features mit sich bringen, mit wenig Aufwand in das System integriert werden.

Die Einführung von Raum-Zeit-Parametern ermöglicht auf einfache Weise das Programmverhalten zu steuern. Bei der Einplanung des Programms müssen sowohl Zeit als auch Raum, welche orthogonale Dimensionen darstellen, berücksichtigt werden. Dies führt unweigerlich zu einer Erhöhung der Komplexität. Zusätzlich wird eine Koordinierung der Bewegung mobiler Geräte notwendig.

Preface

Dedication

I dedicate this thesis to the most wonderful person I have ever known. She was significantly involved in many discussions that, finally, led to the decision that I started my PhD at the TU Berlin. She was my best personal advisor, the most trustworthy person I have ever met, the person who guided me during the long time and hard work on this thesis. The person which whom I had many discussions about this work. Based on her strong encouragement I finally finished this work successfully. She was a wonderful person and she will always be a wonderful person in my thoughts.

The most special thanks go to you, Roberta. I love you for everything that you have done for me in any direction. It is an honor to have met such a person like you.

In loving memories

Acknowledgements

This work would not have been possible with the help of many people:

First of all, I would like to thank my advisor Prof. Dr. Hans-Ulrich Heiß for giving me the opportunity to conduct research at his group and for providing me with everything necessary for completing it.

I thank Prof. Dr.-Ing Jan Richling and Prof. Dr.-Ing. Reinhardt Karnapke for helpful advice and support. I say thanks to all colleagues that were present during that time at the Communication and Operating Systems group (Arnd, Anselm, Tammo, Alex, Matthias, Daniela, Helge, Jörg, Jan, Stefan und Mohannad).

I thank Prof. Jan Rabaey for the inspiring stay as a visiting researcher at his institute, the Swarm Lab of UC Berkeley. In these seven months abroad, I made a significant progress for this thesis. California, in general, was an inspiring place in many directions.

Special thanks go to my mother Gabriele, my father Dieter, my sister Svenja and Sven. They always encouraged me while working on this thesis.

Many thanks go to all friends which directly or indirectly supported me by technical or social contributions: Robert, Alexej, Daniel, Rene, Jens and Anton.

Many thanks go to the Berkeley fellas Joseph, Barbara, Marcel, Severin and Matteo.

Finally, many thanks go to the entire *Team Berlin* for all help that you have provided, especially to Rike and Thomas.

Publications

Parts of this thesis are based on previous publications:

- Chapter 1 is partially based on work published in [32, 33, 34].
- Chapter 3 is partially based on work published in [33].
- Chapter 4 is mainly based on work published in [32, 33, 34]. It is partially based on preceding publications [25, 26, 27, 28, 30, 31, 36].
- Chapter 5 is mainly based on work published in [32, 33, 34]. It is partially based on preceding publications [27, 28, 30, 31].
- Chapter 6 is mainly based on work published in [29, 35].
- Chapter 7 is partially based on work published in [32].

A few additional co-authored publications have not been incorporated into this thesis [68, 69, 93].

Contents

List of Figures

List of Definitions

List of Tables

List of Listings

Chapter 1

Introduction

Based on emergent behavior, natural swarms that consist of ants, termites or birds are a fascinating phenomenon. Those *systems* are surprisingly robust due to massive redundancy. There is a broad field of research targeting bio-inspired approaches which are based on locality and simplicity. This thesis focuses on a paradigm of programming swarms of mobile robots on a systemic level. In the next section, the current state of the technological evolution is presented.

1.1 Past, Present and Future

Evolution describes the change of things. Different disciplines and different scientists interpret the term in a slightly divergent manner. In biology, for instance,

> "evolution is concerned with inherited changes in populations of organisms over time leading to differences among them" [38].

Different perceptions lead to different definitions of the term as shown in [38]. In information technology, evolution is concerned with the development of technology. In [5] and [6], a detailed description of the technological evolution is given that analyzes the stages ranging from *tools* over *machines* to *automation*. However, making a large step towards the present and only allowing a glance in the last century, a strong technological development is noticeable: The first transistor was presented in 1947. 24 years later, Intel launched the 4004 processor which already contained $2,300$ transistors and was built according to the $10,000$ nm manufacturing process. In 2014, Intel presented the 18-core Xeon Haswell-E5 processor which featured $5,560,000,000$ transistors and was based on the 22 nm process. This rapid development including strong processing power, extremely small manufacturing process resulting in very small dies, low energy consumption and economical reasonable prices had a strong influence on the economy as well as social life.

> "With unit cost falling as the number of components per circuit rises, by 1975 economics may dictate squeezing as many as 65,000 components on a single silicon chip"

was stated in [61] by Gordon E. Moore in 1965. He predicted that the number of transistors on a single die would double every year for the next 10 years. This prediction has become famous and is well known as "Moore's Law". Though, later on, the doubling period got larger and varied between 18 and 24 months, a roughly linear correlation is still observable. Some authors interpret Moore's Law not only in the original sense, but rather as

> "a metaphor for anticipated rapid rates of change-not only in semiconductors, but in economic and social contexts" [89].

The end of Moore's Law has been predicted several times in the past as well as the present. The law has now been valid for 50 years. As stated in [98], Moore predicts the end of the law by 2025. Certainly, there are limitations given by physics. If this will be the reason, or, as other authors assume

> "In fact, economics may constrain Moore's Law before physics does–an observation that others have called *Moore's second law*" [89]

remains an open topic from nowadays perspective. However, besides the tremendous increase of transistors on a single die, is the tremendous increase of devices worldwide. Statistics show that there are already more mobile devices on the planet than human beings worldwide. In 2014, 7.7 billion mobile devices have been recorded. Predictions say that in 2018 an amount of 12.1 billion mobile devices will be reached while the number of worldwide mobile users will be 6.2 billions [99]. That indicates an amount of two mobile devices per user. This statistic considers only mobile devices such as phones and tablets. The Wireless World Research Forum (WWRF) forecasts an even higher number of wireless devices. Their global technology vision states that

> "7 trillion wireless devices serving 7 billion people by 2017" [101].

This prediction has also been picked up by the TerraSwarm Research Center whose center mission is to enable distributed sense-control-actuate applications executed on a swarm platform using an universal system architecture. Their vision is a *TerraSwarm*:

> "Some industry observers predict that in ten years there will be thousands of smart sensing devices per person on the planet [..] (yielding a "tera-swarm"); if so, we will be immersed in a sea of networked real-world interface devices" [49].

Going one step further, the plethora of devices will not only surround human beings. Instead, there are

> "devices that are embedded in the environment around us and on or in our bodies" [50].

The robot market is also in a growing phase. The IEEE Spectrum showed in [20] that there was a total robot population of 4.5 million in 2006 (3.5 million service robots and 0.95 million industrial robots). In 2008, the number had almost doubled and reached 8.6 million (7.3 million service robots and 1.3 million industrial robots). The statistics originate from the International Federation of Robotics (IFR).

Another indicator that the development, production and the application of robots is an ongoing process is shown by the recent behavior of two large industrial companies: Amazon presented the conceptual idea of their drone-based system Amazon-Prime-Air [11] in 2013 while Google acquired its eighth robotic company—Boston Dynamics [78]—in the same year [88].

1.2 Motivation

Since numerous heterogeneous stationary as well as mobile devices equipped with a variety of sensors and actuators exist, new kinds of applications are feasible. Devices range from deeply embedded sensors and actuators over wearable devices to fully autonomous robots. Together, they form distributed sensing and actuating platforms that are highly interconnected. Based on different device manufacturers and system developers, a variety of different hardware, different system software with different system interfaces exists, exposing heterogeneity.

Plenty of applications require access to specific sensors and actuators in order to collect data (e.g., wind speed, temperature, humidity, seismographic activity, ..). Examples include: Traffic management systems that require access to traffic data and weather data. Flood prediction systems that require access to weather data and data about the water level of a certain river section. Long-term bridge monitoring systems that require access to traffic data, weather data and seismographic data. Robot-based exploration or observation systems that require access to specialized cameras and probes.

All these applications have in common that they require context awareness concerning physical space and time since their functional outcome heavily depends on these context parameters. Considering the above mentioned applications, there is a noticeable fraction of the applications that require similar or even the same sensors and actuators:

- Tsunami early warning systems that use pressure sensors in the ocean.
- Marine biologist that are interested in monitoring water temperature, current and salinity in order to make statements about marine life.
- Beach water quality monitoring to prevent people from going into water with bacterial contamination.

Current approaches tend to set up their own infrastructure (interconnected hardware components) that are perfectly tailored to solely run one application.

Instead of following this approach, virtualization techniques can be used in order to allow multi-program operation on the same physical infrastructure. This way, hardware

resources are shared among the applications by executing them in a time multiplex manner. This result in a significant reduction of deployment and maintenance costs. In order to achieve this, a system is presented in this thesis that manages and coordinates its resources. Applications are scheduled and managed by the system. Application developers profit from this approach since the system hides all heterogeneity, is responsible for resource management, synchronization and coordination and provides one clear defined interface against which applications are implemented.

1.3 Thematic Demarcation

In order to state the field of investigation, this section gives a coarse overview of related research disciplines.

Internet of Things The Internet of Things (IoT) is a network of physical objects that are interconnected and are uniquely identified. Those objects are equipped with a variety of sensors and actuators. Such devices can be remotely accessed and are able to exchange data. "According to the Cisco Internet Business Solutions Group (IBSG), IoT is simply the point in time when more "things or objects" were connected to the Internet than people."[1] According to this definition, the IoT was born between 2008 and 2009. Already in 1988, Mark Weiser mentioned the vision of *Ubiquitous Computing* and presented this vision in an article in 1991 in which he stated that "in the 21st century the technology revolution will move into the everyday, the small and the invisible" [103].

Cyber-Physical Systems There are many different definitions of cyber-physical systems (CPS). Yet, the question remains open if there will be a commonly acknowledged definition. CPS is often referred to as the next generation of embedded systems where, in contrast to embedded systems, the focus is not on the computational part but rather on the intense link, so the network, that connects local computing components. A well known definition for CPS originates from Edward Lee who said that "Cyber-Physical Systems [..] are integrations of computation with physical processes. Embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computations and vice versa." [48]. Designing such cyber-physical systems is a challenging task [47, 48].

New markets arise which also include the biological sector: "The trend in healthcare is *cyber-biological systems* where devices outside and implanted in the human body wirelessly communicate with physical systems."[2] In [76], principles about body-area networks and the human intranet are presented.

[1] Cisco IBSG, 2011.

[2] IEEE Austin COMSOC/SP Meeting about Big Data, Cyber-Biological Systems, and Pattern Recognition which was scheduled on Sep. 25, 2014. Speaker: Choudur K. Lakshminarayan, Principal Scientist, HP Software Research.

Swarm Intelligence Swarm intelligence is the collective behavior of decentralized systems that are based on self-organization. They are also referred to as systems that are inspired by biology since their behavior originates from nature [8]. These approaches are characterized by simplicity, locality and dynamics: they follow simple rules, perform only local interaction and are designed for large networks with highly dynamic behavior. Those algorithms are, for instance, Ant Colony Optimization (ACO), Particle Swarm Optimization (PSO), Artificial Bee Colony Algorithm (ABC) or Artificial Immune Systems (AIS). There is also a Multi-Swarm Optimization algorithm—a variant of the original PSO—which creates multiple sub-swarms, each of which is responsible for a certain region. This variant is used for optimization on multi-modal problems with multiple optima. Although the last algorithm implies to create sub-swarms, all these algorithms are designed in order to use the entire swarm. It is not intended to use a composition of algorithms on the same swarm.

Swarm Robotics "The swarm robotics inspired from nature is a combination of swarm intelligence and robotics" [96]. The intention of swarm robotics is to use bio-inspired swarm algorithms in order to coordinate multirobot systems. They can grow up to enormous size and can still be efficiently coordinated. Due to the decentralized organization, the system is very scalable and robust. Swarm robots are intended to be simple and cost-saving. Ant robots are a special case of swarm robots. The intention of ant robots can be interpreted as a light version of swarm robots as they are even smaller and cheaper. Ant robots belong to the field of micro-robots.

A plethora of divergent approaches can be found in literature. However, most *swarm* approaches have in common that they execute one application or one algorithm on the swarm. This thesis does not target bio-inspired algorithms. It targets cyber-physical systems with a major focus on swarms of mobile robots. Therefore, this thesis presents an approach that is based on a swarm operating system that serves as a mediation layer between the swarm components (devices) and the application space. The system allows to execute multiple, independently developed applications in isolation from each other.

1.4 Shortcomings of Current Approaches

There are many projects, research communities and approaches that deal with CPS, swarms and robots. However, there are still shortcomings:

Real space-time programming Nowadays, a plethora of programming abstractions exists. There are query-based abstractions for sensor networks and also abstractions that implement the notion of space. However, none of the abstractions support real (physical) space-time programming such that applications can be made context-aware by annotating parts of the application with spatio-temporal constraints (absolute and relative constraints). Furthermore, the abstractions do not feature a transaction semantic that enables guaranteed execution of a set of instructions on distributed resources

of the swarm. Resources are not addressed on a systemic level. Furthermore, as also Mottola and Picco state "[..] none of the programming solutions considered in our survey specifically addresses applications with mobile nodes or sinks. The requirements to meet in these scenarios are, however, quite different from the challenges in static applications. Location is usually of paramount importance, [..]. Therefore, programmers must implement, on a per-application basis, mechanisms such as neighbor discovery as well as store-and-forward mechanisms. Ideally, higher-level programming abstractions should be developed to shield programmers from these aspects." [64].

Systemic description Swarming or flocking often addresses certain formations that occur as emergent behavior based on simplicity and locality. Formations may have several intentions, e.g., uniform observations or measurements. Performing an observation task, it might be useful or even necessary to exactly specify such a formation. A systemic description is needed that allows to specify *where* and *when* certain instructions shall be executed and using *what* kind of resource.

Mobile cyber-physical operating system The number of devices that are interconnected by communication infrastructures increases rapidly. Different device manufacturers, different (local) operating systems with different system interfaces form heterogeneous distributed systems. Distributed applications and algorithms have to cope with the heterogeneous world. A cyber-physical operating system that hides all heterogeneity beyond the system interface, performs all resource allocation and management and uses time-multiplexing in order to share hardware resources among the requesting applications is still missing. Distribution, concurrency and motion shall be concurrent. Furthermore, the local operation of a device in terms of using its sensors and actuators together with its movement should also be completely managed by the operating system.

Automatic resource movement Mobile systems such as mobile robotic systems require collision-free path planning (a coordination in physical space and time). This produces spatio-temporal trajectories. Control algorithms are required in order to operate the robots actuators to follow the spatio-temporal trajectory. The planning as well as the control algorithm have to take the devices' capabilities into account, e.g., a wheel-based robot has a different movement and steering behavior than an unmanned aerial vehicle (UAV) or an underwater vessel. Also, external physical incidents have to be considered, e.g., strong current or wind have potential impact on the respective device. A management and control component as part of a cyber-physical operating system for mobile devices is a necessity for spatio-temporal applications.

All addressed issues are still missing and are essential building blocks for the vision of an autonomous swarm system.

1.5 Contribution of this Thesis

In order to address the shortcomings of current approaches, the contribution of this thesis is based on three major parts: a *programming model for CPS*, a *modular architecture for a cyber-physical operating system* and a *space-time scheduler*.

Programming Model for CPS In order to program applications for the swarm that have access to physical space and time, a suitable model is presented. A swarm application consists of actions which are interactions with the physical world by means of sensors and actuators. An action can be made context-aware by assigning spatio-temporal constraints to it. An action is then scheduled and managed by the cyber-physical operating system. In particular, the space-time scheduler—a core component of the operating system—is responsible for scheduling the action in space and time. Due to the constraints, the classical sequential workflow of an application is mixed up. A blocking execution semantic is not desirable. Therefore, the programming model is asynchronous and event-based in order to avoid blocked passages. Concurrent and distributed programming is often error-prone. Thus, the model allows the programmer to implement applications in a sequential manner (providing necessary transparencies), and translates the program into a concurrent and distributed application at runtime.

Cyber-Physical Operating System A dedicated cyber-physical operating system for the swarm is required that is perfectly tailored to execute such swarm applications. Therefore, the second part of this thesis deals with the design of a distributed system architecture. A strong emphasis is put on the mobility of its components. The design follows a service-oriented architecture approach. Based on modularity, new algorithms and functionality can be easily added and existing ones can be quickly exchanged. This is necessary in order to integrate new features (e.g., by incorporating additional sensors and actuators) and to deploy new control algorithms. The architecture's components are loosely coupled and use message passing to exchange information. A prototype has been developed which implements the designed swarm architecture.

Space-Time Scheduler The core component of the cyber-physical operating system is the space-time scheduler. Its task is to perform the resource management by allocating and releasing system resources in a timely manner. The scheduler has global knowledge about the world which consists of static and dynamic obstacles. Static obstacles are presented as arbitrary polygons that are time-invariant, i.e., their physical location does not change over time. Dynamic obstacles, in contrast, are entities that move over time. Therefore, their location is a function of time. Dynamic obstacles can be non-controllable entities whose space-time behavior is well known (a satellite on its orbit) or controllable entities that are managed by the operating system (a mobile robot). The scheduler schedules all actions from the set of active applications in space and time. In order to avoid collisions, the scheduler uses the concept of forbidden regions which are spatio-temporal blockings of a dedicated path.

1.6 Structure of the Thesis

The thesis is structured as follows: In Chapter 2 the current state of the art of swarm oriented projects is given. Chapter 3 presents the swarm idea as used in this thesis. The programming model is described in Chapter 4. The system architecture is presented in Chapter 5. Chapter 6 describes a modeling approach of group scheduling in space and time. Chapter 7 introduces the core-component of the swarm system, the space-time scheduler. Chapter 8 presents an evaluation of this work and Chapter 9 concludes this thesis.

Chapter 2

Related Work

This chapter shows the state of the art in swarm oriented projects. Section 4.2, section 6.2 and section 7.2 show additional related work in particular research fields (programming abstractions, Petri nets and scheduling), that have been required in order to realize this thesis.

2.1 Swarm-bots

The Swarm-bots project[1] targeted to investigate approaches to self-organizing and self-assembling technologies [59, 18]. The project addresses approaches inspired by nature, e.g., social insects [9]. The body of the robot called s-bot has a diameter of 116 mm [60]. The robot features a 400 MHz Intel XScale CPU running Linux and is equipped with 64 MB RAM [58]. The robot has several sensors, e.g., infrared, humidity, temperature, ambient light, accelerometers, microphones, camera for omnidirectional vision. In addition, the robot is equipped with a 3 axis gripper side arm together with several RGB LEDs. The robot uses a combined drive called treels which is a composition of tracks and wheels. Using the gripper which is mounted on a rotatable turret, the robot is able to grab other robots in its proximity at a dedicated position—a specific hook—and so connect robots in a secure way. This way, the robots form chains [66, 87] similar to rope teams in climbing sports. Doing so, the robots secure each other while overcoming obstacles.

2.2 I-Swarm

The I-Swarm (Intelligent Small World Autonomous Robots for Micro-manipulation) is a European research project that started in January 2004 and lasted until June 2008 [90, 105]. The project aims to develop large-scale miniature robots. The swarm is expected to contain 1000 robots with a size of $3 \times 3 \times 3$ mm [10] that should be

[1] Project has been sponsored by the Future and Emerging Technologies program of the European Commission, 2001 - 2005.

produced using mass production technologies. Solar cells are used in order to operate the robots. Communication is performed using infra-red LEDs and photodiodes in four different directions. The communication range is very low resulting in only regional information exchange targeting only a few robots. The robot is equipped with an ASIC based on a DW8051 core. Algorithms that have been developed during the project have been simulated while in the end of the project the actual hardware (the robot) was not completely operational [21]. In order to control large-scale swarms, the project aims to invent decentralized algorithms that are characterized by simplicity and only local interactions as found in nature. Using bio-inspired approaches virtual pheromones are used in order to perform local communication which is also known as stigmergy [7, 53]. There are different applications for this approach including collective exploration, shortest path finding and task allocation [39].

2.3 Swarmanoid

The Swarmanoid project[2] is the successor of the Swarm-bots project and, hence, uses the gathered results. The project has investigated a novel distributed robotic system that consists of heterogeneous robots that are able to connect in order to provide new capabilities to the system. There are three types of robots: eye-bots, hand-bots and foot-bots. All of them feature the same following hardware components: they are equipped with a 533 MHz i.MX31 ARM 11 CPU with 128 MB RAM and 64 MB Flash [17]. The foot-bot is an improved robot platform of the s-bot from the Swarm-bot project. The foot-bot has a diameter of 13 cm and is 28 cm tall. Its drive is also based on treels and reaches a maximum velocity of 30 cm/s. It is equipped with a docking ring and a gripping mechanism that allow multiple foot-bots to connect together.

The hand-bot is equipped with two large grappler in order to grab objects or to climb vertically. It has no motion capabilities and, therefore, grounded movement is only possible by connecting the hand-bot to other foot-bots. The diameter of the robot is between 41 - 47 cm (depending on arm positions) and 29 cm high. The robot is, in addition, equipped with a rope launcher on the top that catapults a magnet which allows to connect to ferromagnetic ceilings in order to lift up the robot.

The eye-bot is a flying platform with a diameter of 50 cm and a height of 54 cm that consists of a 4×2 co-axial rotor system. It is equipped with a ceiling attachment system in order to attach it to the ceiling and save energy. The eye-bot has a camera and is able to provide a bird's eye view in order to coordinate movement in 3D. Compared to the I-Swarm, the size of the swarm is rather small with about 60 robots in total.

[2]Project has been funded by the Future and Emerging Technologies program of the European Commission, 2006 - 2010.

2.4 SwarmRobot

The *swarmrobot.org*[3] project is an open-hardware microrobotic project for large-scale artificial swarms [42]. The intention of the project was to build a platform in order to simulate swarm behavior using real robots named Jasmine. In order to reach general acceptance and create incentives of the platform, the robots consist of cost-effective hardware components and appropriate software libraries enable quick implementation of user-specified swarm algorithms. The robots themselves have been manufactured with a size of $26 \times 26 \times 26$ mm and are, hence, approximately 9 times larger than compared with their counterparts from the I-Swarm project. The robots feature 2 kB RAM, 24 kB Flash ROM and a 1 kB nonvolatile EEPROM. They are equipped with 2 DC motors (forward and backward rotation) enabling a maximum velocity of 500 mm/s.

A couple of experiments have been performed on the Jasmine microrobots. In [43], an approach of collective energy homeostasis in a large-scale microrobotic swarm is presented. An intelligent docking station as well as a suitable on-board recharging electronics enable the robots to autonomously perform a recharge process. They show how to increase collective efficiency (avoid low energetic robots) by using collective decision making.

2.5 Symbrion & Replicator

The Symbrion & Replicator projects[4] are based on knowledge acquired from previous research: the I-Swarm project and the open-source project SwarmRobot.

The Symbrion (Symbiotic evolutionary robot organisms) and Replicator (Robotic evolutionary self-programming and self-assembling organisms) projects [44] consist of large-scale swarms of robots featuring sensors and actuators to interact with the physical world. Both projects are based on bio-inspired approaches featuring self-X properties. The systems are highly dynamic and so, if advantageous, the robots can aggregate into a symbiotic organism that is, in this form, better suited to solve a task in the current situation while sharing resources such as energy.

Aggregating into a higher symbiotic organism requires precise control of actuators of the individuals. In [52], different controller types required for actuation based on bio-inspired and evolutionary approaches are presented. The aggregation process requires mechanical assembly of the individuals units. In [45], scout and backbone robots featuring different capabilities are presented. It is shown how these robots can mechanically be connected. Based on the connection and the joints different kind of organism shapes are possible. A grand challenge is proposed in which 100 robots should survive autonomously for 100 days while necessary power sockets are mounted in areas which are not reachable for a single unit. Only using collective behavior and adopting into a higher symbiotic organism enables power supply.

[3]Project has been cross-funded by several other grants, amongst others by the I-Swarm and the Symbrion & Replicator projects.

[4]Projects have been funded by the European Commission, 2008 - 2013.

2.6 Kilobot

The Kilobot [84] is a swarm robot which puts a high emphasis on the development of very low cost units in order to enable large-scale swarms consisting of up to 1000 devices. It uses cost-efficient components such that the total costs of the hardware are less than \$15 and "takes 5 minutes to assemble" [80, 81]. With 33 mm, the diameter of the robot is a little bit larger compared to the Jasmine swarmrobot, but costs only approximately a tenth of the Jasmine which is priced at \$130.

Several experiments have been performed on Kilobot swarms. In [85], a distributed algorithm called DASH[5] has been presented. The algorithm has been extended to S-DASH[6] by including automatic scalability [86]. The algorithm requires to specify an arbitrary geometric shape, e.g., a cube, a triangle, a star. Using the user-specified shape (spatial constraints) the Kilobot swarm iteratively approaches the shape and "fills" it. Seed robots are used in order to define the origin and orientation of the coordinate system, i.e., the spatial points where the shape starts [83]. The algorithm is based on only local interactions between the robots. Using edge following and gradient formation, robots move in close proximity along the "edge" of other robots keeping a fixed distance d to others (the robots move iteratively) towards the so called *source* robot which has a gradient value of 0. Afterwards, the robot "enters" the user-specified shape and positions itself based on higher gradient values. In [82], an approach is shown in order to collectively transport complex objects. An experiment shows 100 Kilobots collectively performing the transportation task.

2.7 iRobot-SwarmBot

The SwarmBot is a trademark of the iRobot company. Its size is approximately $13 \times 13 \times 13$ cm and, hence, belongs to the fraction of larger robots. It is equipped with a 40 MHz ARM CPU and 648 KB RAM. For the drive, the robot uses 2 electric engines. Furthermore, the robot features light sensors, a camera and bump sensors. For communication, the robot uses the ISIS infra-red communication system [54].

In [55, 56], an approach for dynamic task assignment using the iRobot SwarmBot based on swarm algorithms is presented. Tasks are assigned to sub-groups of robots that collectively perform the task. The authors show four different algorithms and evaluate their behavior concerning dynamic task assignment.

2.8 The TerraSwarm Project

In [49], the TerraSwarm Research Center proposes a distributed operating system that serves as a mediation layer between applications and distributed resources such as sensors, actuators, storage and computing. It has to cope with distribution, heterogeneous

[5] Distributed self-assembly and self-healing (DASH).
[6] Scalable, distributed self-assembly and self-healing (S-DASH).

and shared resources as well as dynamic situations (mobility, connectivity) while providing context awareness. The presented approach does not feature autonomous resource movement that is based on spatio-temporal trajectories that are automatically computed from the system based on a resource usage of applications.

Chapter 3

Swarm Idea

This chapter describes the overall idea and vision of a large distributed sensing and actuating platform. Once established, new kinds of applications are possible. As a motivation, some exemplary applications are presented within this chapter and a classification of different application categories is shown. This chapter serves as a motivation for the remaining chapters.

3.1 Mobile Distributed Systems

Mobile distributed systems can be characterized by the degree of control these systems have about their own mobility:

The first class of devices, comprising for example smartphones and netbooks, cannot actively influence their location and motion as people usually carry them in their daily life. Thus, each device may be at a different location (with individual heading and speed) connected to different networks by links of variable quality. Moreover, these characteristics usually change unpredictably over time.

The second class of mobile devices exhibits an invariant movement and is, thus, deterministic. An example for this class are satellites located in an earth orbit moving on predetermined trajectories around the planet. Together with all applying forces such as gravity and friction, their location is a well defined and predictable function of time.

The third class is composed of devices such as mobile robots designed to play soccer that are able to move on their own and to individually control their movement. Their location is bound by the playing field on which they move in a coordinated manner. However, as soccer robots may make autonomous decisions, the speed and heading will be hardly predictable by an external entity without complete system knowledge.

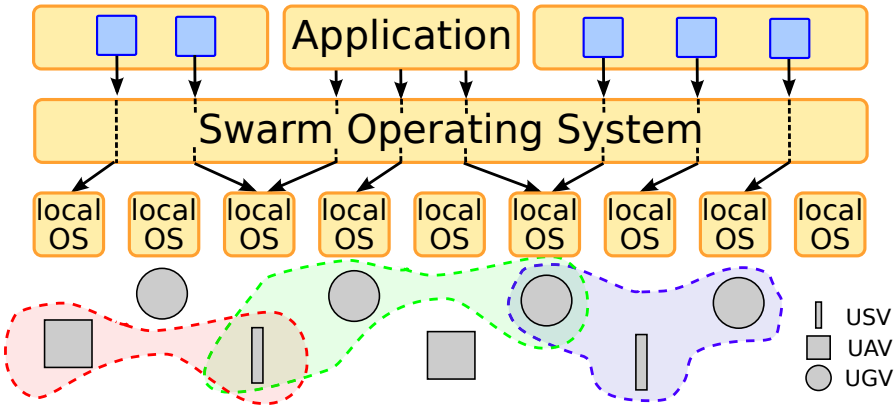


Figure 3.1: Role of the swarm operating system which serves as a system layer.

As one common characteristic, all those devices (independent of their class) are able to achieve an emergent behavior by cooperation and interaction. Applications aiming for such behavior usually focus on the collective rather than on the individual single device. This view is called a system view, in contrast to a node view, where each member of a group is targeted separately. A system view is advantageous since it reduces the need to think of and deal with concurrency which is inherently error-prone. For that reason, many approaches for distributed systems aim to hide distribution and thereby concurrency.

3.2 The Approach

The term globalization is a well known concept that is present in many disciplines such as politics, economy, academia and technology. It describes the process of higher (global) integration. The Internet has already connected people worldwide. Information can be exchanged in seconds. A plethora of applications that use the Internet, e.g., telephone programs such as Skype, induce the feeling of being in proximity although the physical distance might be several thousands of kilometers. Initially launched in 1969 with only four nodes, the predecessor of the infrastructure known today as Internet was called the Arpanet. An impressive statement—that has become true from nowadays perspective—is the vision of the "Galactic Network" that has been forecasted by Joseph Licklider in 1962. According to recent statistics of 2015, over 3 billion people, which is nearly 50% of the global population, are connected to the Internet.

Automation is the technology of tomorrow which emphases concepts such as the Semantic Web, Smart Cities and Smart Factories. Imagine a future in which programs can be implemented or services can be accessed which leads to cyber-physical interactions. The net or a subnet becomes a large distributed sensing and actuating platform as shown in Figure 3.1. The following is especially designed for the category of mobile robots, i.e., mobile sensor-and-actuator networks. Programs can be implemented in a conventional manner which leads to coordinated robot actions.

Imagine a swarm of a plethora of heterogeneous mobile robots consisting of UAVs[1], UGVs[2], USVs[3] and ROVs[4] featuring different sensors and actuators and, thus, different capabilities. The system has very precise knowledge about all its resources, i.e., the service that the resources provide. All these robots require fine-grained control algorithms for their actuators, e.g., in order to move along a pre-defined trajectory or to control the motion of robot arms with mounted sensors. Furthermore, dedicated drivers must be available in order to access specific sensors. Since this requires expert knowledge in the respective problem domain, this thesis presents a programming model that provides the following features:

- **High Level Instructions:** The system provides high level instructions and introduces an abstraction that hides complexity (distributed control of sensors and actuators) beyond the systems interface.
- **Context Awareness:** Using spatio-temporal constraints, the programmer can assign runtime parameters to code fragments which specify the physical location and time at which the execution shall take place.
- **Transparencies:** Application development is facilitated by introducing suitable transparencies: the programmer is able to assign physical space constraints, but is released from selecting a suitable robot candidate. Thus, *location transparency* is provided. Furthermore, *concurrency* and *distribution transparency* enable to program in a sequential manner without the need to cope with race conditions and synchronization.
- **Neither Quantitative nor Qualitative Allocation:** The quantitative and qualitative allocation is performed by the swarm operating system depending on the applications' resource requests, the capabilities of the resources and the availability of resources.
- **Collision-Free Trajectories:** Multiple robots that move in the same physical space have to be carefully coordinated. In addition, if task execution is timely restricted, a spatio-temporal collision-free trajectory planning is required. This aspect is also completely hidden from the programmer.

3.3 Definitions

In order to establish a clear separation between the used terms, a few definitions are given in the following:

Definition 1 (Local Operating System) *Local operating system refers to a standard operating system such as Linux, Windows, TinyOS, Contiki or Reflex.*

[1]Unmanned aerial vehicle
[2]Unmanned ground vehicle
[3]Unmanned surface vehicle operating on the surface of the water
[4]Remotely operated underwater vehicle

Definition 2 (Swarm Operating System) *The swarm operating system defines the system software that is required in order to manage and coordinate the swarm.*

Definition 3 (Physical World) *The physical world is every process that happens in the real world.*

Definition 4 (Physical Entity) *A physical entity is an object which has a certain geometry and is part of the physical world.*

Definition 5 (Cyber World) *The term "cyber world" defines all computation which is executed on hardware. The cyber and physical world are disjoint.*

Definition 6 (Physical Event) *Every happening in the physical world is reflected as a physical event. A happening is defined as a state change. A state change could be, for instance, a relocation of a physical entity. A relocation can be measured in different ways: RFID tags that are attached to this entity and are scanned in order to determine the location. Also, GPS modules can be used. In [70], drifters*[5] *are applied in order to determine the location and geometry of an oil spill [71].*

Definition 7 (Cyber Event) *A cyber event is a happening in the cyber world. A physical event can be reflected as a cyber event and vice versa. A spatio-temporal event model for CPS has been presented in [95].*

Definition 8 (Physical Movement) *Physical movement defines the motion of a physical device.*

Definition 9 (Virtual Movement) *The term "virtual movement" defines the "motion" of a virtual entity, e.g., a program. Virtual movement is existent if the program executing device physically moves itself or if the program is migrated to another device.*

Definition 10 (Physical Swarm) *The term "physical swarm" refers to the sum of physical computing devices that are part of the swarm.*

Definition 11 (Virtual Swarm) *A virtual swarm is an execution context of a swarm application. In particular, it is a time-varying mapping of application parts to devices. A virtual swarm exists as long as the corresponding application exists.*

Definition 12 (Systemic Description) *The term "systemic description" defines the system-wide addressing, allocation and usage of system resources without the need to know where these resources are available and how to access them. Thus, location-, distribution-, concurrency- and motion-transparency is provided. Furthermore, a systemic description can include spatio-temporal constraints in order to constrain instructions in an absolute or relative manner.*

[5] Drifters are oceanographic floating devices equipped with sensors that move along with the current.

3.4 Example Applications

In the following sections, some exemplary applications will be presented:

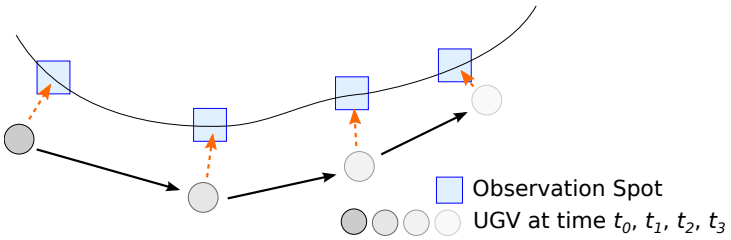
3.4.1 Stationary Monitoring

An application that uses stationary monitoring is defined as performing one or more interactions with the physical world at the same location, e.g., taking a picture of a point of interest. This could be a single instruction or periodically repeated. In this case, the application stays *virtually* over the physical location while the executing devices may change over time. In case of UGVs, the simplest schedule would include only one robot that, once positioned correctly, stays at that location until the application is finished. The schedule may also incorporate multiple robots such that each of the robots takes over a part of the execution. The same holds for certain UAVs, e.g., quadrocopter which are able to "stay" over a certain location. Usually, UAVs that are constructed using wings are not able to stay over a certain location; they are rather required to fly along trajectories. In this case, trajectories have to be computed that cross the point of interest several times at certain points in time. Considering another class of devices (satellites), the situation changes again since such devices orbit, on a predefined trajectory, a planet. Their location is, thus, a function of time. A stationary application needs to be migrated each time the satellite leaves a certain space.

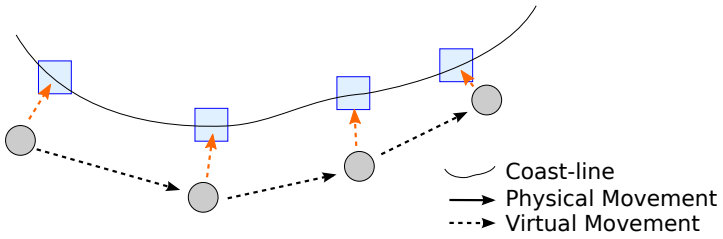
3.4.2 Shore Monitoring

In the second example, the goal is to monitor the shore line, e.g., for detecting contamination due to an oil spill. For this, a series of observations shall be performed which is shown in Figure 3.2 indicated by the observation spots. The application consists, therefore, of four instructions with at least spatial constraints. If necessary, temporal constraints can be assigned in addition in order to express temporal dependencies. By simply adding temporal constraints, the execution order and multiplicity can be easily modified. Possible configurations are, for instance, all monitoring shall be performed at the same point in time, sequentially according to a certain order or multiple times.

Depending on the constraints and the number and current position of robots, different solutions for this problem are possible. If the temporal constraints demand a simultaneous execution, then multiple robots are required. If the temporal constraints are given in increasing order or no temporal constraints are specified, this enlarges the solution space. Discarding all temporal constraints allows a one robot solution. As shown in Figure 3.2(a), the robot moves along the trajectory and executes all instructions in a sequential manner. An advantage of this solution is that only one robot is required. The disadvantage is that the larger the distances are the more time and energy is required. Thus, instead of physically moving the robot along the trajectory, the application moves virtually by "migrating" from robot to robot as shown in Figure 3.2(b). This also allows any arbitrary constraint configuration as long as enough robots are present and are able to reach the observation spots in time.



(a) Physical movement (one robot)



(b) Virtual movement (four robots)

Figure 3.2: Monitoring multiple points along a trajectory.

3.4.3 Exploration

This application shows a robot based exploration scenario. Since every robot has its capabilities and properties, not all robots are able to perform every task. Usually, UGVs are much slower than UAVs. In addition, UGVs are limited by their ground movement ability. The more rough a terrain is the more complicated is the crossing. So each robot has its intended use.

Figure 3.3 shows the cooperation between multiple different robot types in order to reach a common goal. Implementing this application manually requires precise domain knowledge for the different robot types and their application. All these issues are completely transparent for the programmer when using the systemic approach presented in this thesis.

3.4.4 Object Monitoring

In this application, an object (e) shall be monitored from three different sides at the same point in time and with the same distance as depicted in Figure 3.4. Doing this manually would require to select a subset of robots, move and position them correctly. Afterwards, a synchronization must be triggered in order to establish a common knowledge among each other in order to start taking the pictures.

Finally, the pictures have to be collected in order to perform some processing with them. This is already an error-prone approach since it involves not only distribution and concurrency, but also real time and space. Furthermore, in case of a robot failure, a suitable error handling must be triggered.

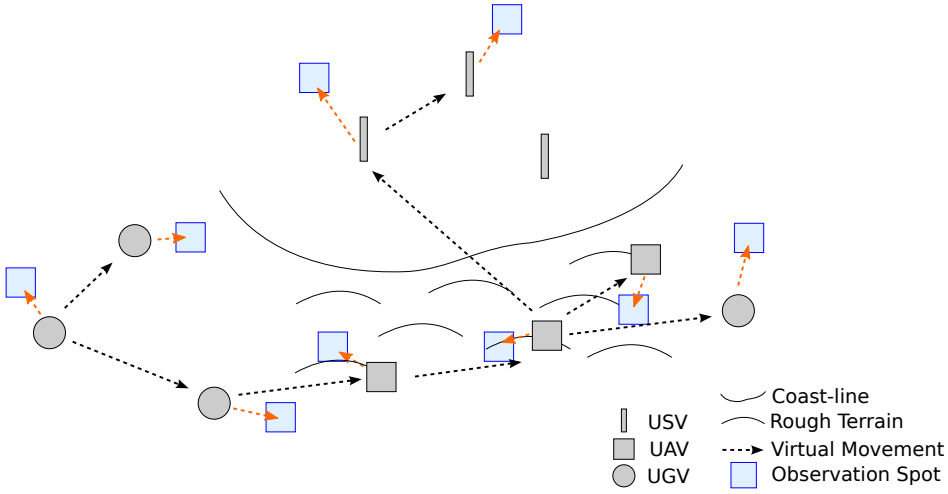


Figure 3.3: Exploration by different robot types.

The new feature in this example is that the task requires a space-time rendezvous. Furthermore, the object of interest is dynamic and moves over time. Since the intention of this application is to monitor the object, this can be done by simply specifying a systemic description using spatio-temporal constraints. The spatial constraints are given by

$$\begin{aligned} \forall i \in \{0,1,2\}\ \exists j = (i+1) \mod 3 \mid \\ dist(p_i, p_j) == D_1 \wedge \\ dist(p_i, e) == D_2 \end{aligned}$$

In order to enforce a simultaneous execution, the following temporal constraints are required:

$$\forall i \in \{0,1,2\}\ \exists t_i == T$$

The parameters T, D_1 and D_2 are intervals that indicate acceptable values. The constraints state that each picture (p_i) that shall be taken must have pairwise the same distance to each other $(dist(p_i, p_j))$. In order to satisfy these constraints, the robots must obtain the formation of an equilateral triangle. Furthermore, the distance between each robot and the observed object (e) shall also be the same $(dist(p_i, e))$. Using the parameters D_1 and D_2, the triangle formation can be scaled. After specifying the spatial constraint, the temporal constraint forces the simultaneous execution.

Using this systemic description, neither quantitative nor qualitative aspects have to be specified and the solution emerges implicitly. Furthermore, since the object is capable of moving—either actively or passively—the formation will transparently follow

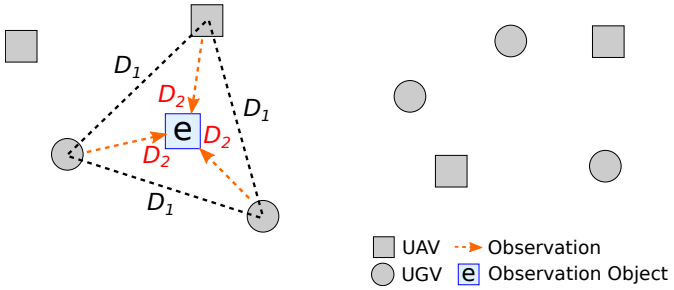


Figure 3.4: 3-sided observation by different robot types.

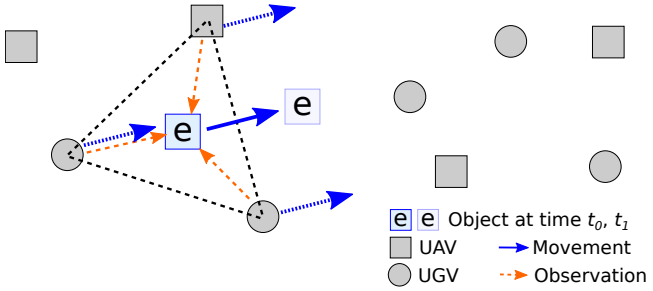


Figure 3.5: Observation of moving object by different robot types.

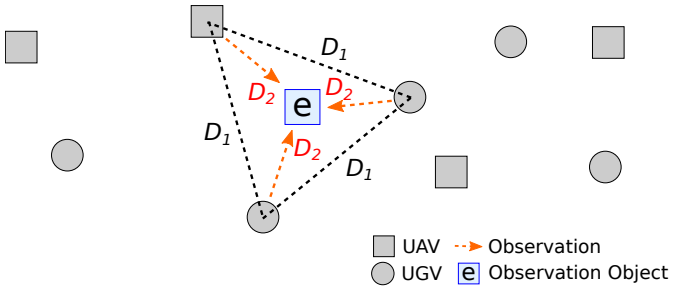


Figure 3.6: Remapping: changing the involved robots while maintaining the observation.

the object based on the systemic description as depicted in Figure 3.5. As given by Definition 4 and Definition 6, a physical entity can be tracked by either using a GPS module which is connected to the entity or in case of the oil spill by specific floating devices that periodically report GPS coordinates. As given by Definition 11, a remapping can be performed, i.e., the involved robots change over time while maintaining the desired formation as shown in Figure 3.6.

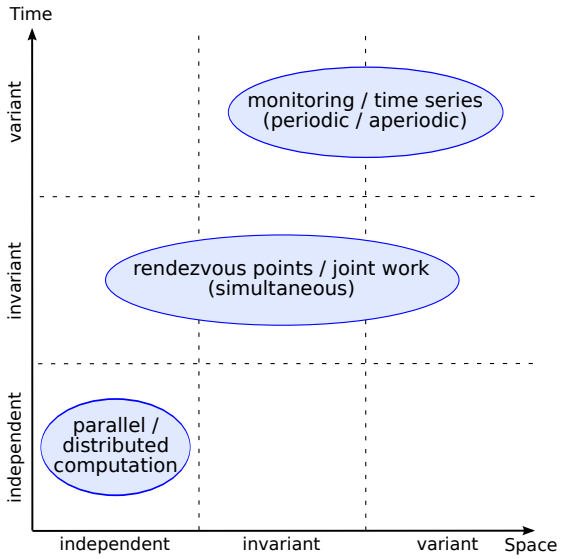


Figure 3.7: Classification of swarm applications.

3.5 A Classification of Applications based on the Space and Time Dimensions

In this section, classes of applications are considered. There are different kinds of applications—those that require context awareness since their functionality heavily depends on context and those that have no relation to context.

Figure 3.7 shows a classification of possible types of applications that has been first presented in [33]. Applications can be classified according to their relation to space (physical coordinate on a surface):

- **Independent**: The application has no notion of space. A typical example could be a parallel or distributed computation problem, for instance, computation of prime numbers or the Mandelbrot set.
- **Invariant**: The application is bound to exactly one location and stays virtually over that point, e.g., monitoring of the water level of a certain river section.
- **Variant**: The application is bound to multiple locations, e.g., monitoring the wind speed along the shore, monitoring the current along an underwater slope, exploring an unknown region or detecting changes in tectonic plate movements along a fault.

The same classification can be done for time:

- **Independent**: The application has no notion of time. Examples are the same as for space-independent ones.

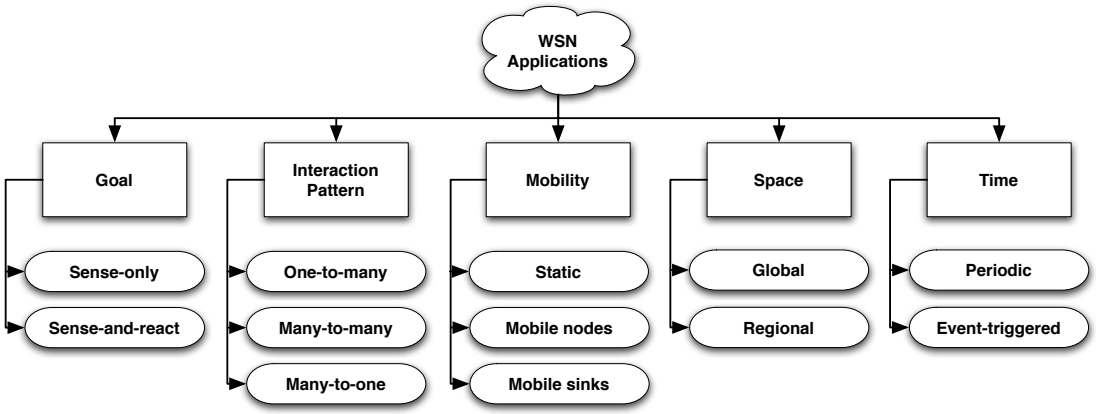


Figure 3.8: WSN applications, after Mottola and Picco [64].

- **Invariant**: The application shall be executed at one certain point in time. This may require multiple executing components. If space is independent, an application could be a friend finder where multiple people want to meet at a given point in time but the meeting point itself is not important. If space is invariant, then space matters additionally. An application that is space variant would be the simultaneous observation of a point of interest.
- **Variant**: This class of applications is bound to multiple points in time which is typical for a time series.

In fact, it is possible to combine classes. For instance, a time series of seismographic measurements in an earthquake region belongs to a class where each entry in the time series is time-invariant and space-variant while the series itself is time-variant.

3.6 Dimensions for WSN Applications

In a survey about programming wireless sensor networks (WSN) [64], the authors have produced a taxonomy of wireless sensor network applications as a result of analyzing WSN applications as depicted in Figure 3.8. This taxonomy has, beside space and time, additional dimensions such as *goal*, *interaction pattern* and *mobility*. Using the programming model that is presented in Chapter 4, different types of applications can be developed allowing combinations of the dimensions shown in Figure 3.8. In the following, a summary of the dimensions and their application for the swarm programming model is given. The concrete realization of the dimensions in the programming model is presented in Chapter 4.

Goal The goal defines the applications behavior. WSN applications have first become popular by deploying *sense-only* applications. The intention was to gather data using

multiple sensor nodes, route this data to a common sink and gather it for an offline analysis, afterwards. The second type were so called *sense-and-react* applications. In this scenario, new kind of applications emerged since applications were now able to sense data and react to it using actuators.

A swarm application can be implemented either as sense-only application or as a sense-and-react application.

Interaction Pattern The interaction pattern defines the communication behavior of the application. A *one-to-many* interaction pattern states, e.g., to send one instruction to multiple nodes in order to request or change a certain behavior. *Many-to-many* describes the interaction which involves multiple senders and multiple listeners. This occurs in applications which have multiple data sinks. Typically, the *many-to-one* interaction pattern describes sense-only applications since data is measured locally at each node and then sent to a dedicated sink for post-processing.

Depending on the intention of a swarm application and how the application is implemented, all three interaction patterns are possible.

Mobility The mobility addresses the degree of freedom of the executing nodes. A *static* node is typically a sensor that is mounted at a fixed position and, therefore, does not move over time. This is the most common scenario for WSN. Going one step further and allowing mobility, *mobile nodes* are able to change their position by physical movement. In addition, data sinks can also be mobile resulting in *mobile sinks*.

Similar to the two previously mentioned dimensions, the programming model supports all three mobility types, whereat the emphasis is laid on the mobility part.

Space With space, the authors distinguish between the applications execution scope. A *global* scope indicates that the applications is executed system-wide, i.e., on every node in the network. In contrast, *regional* refers to a certain region, i.e., a subset of nodes in the network that execute an application.

Swarm applications can be arbitrary developed so that every allocation partition, i.e., every allocation of nodes, is possible.

Time With time, the authors distinguish between the execution semantic: while *periodic* indicates a periodically executed application, *event-triggered* defines the state in which the application is able to react to certain events.

Swarm applications can be developed using both ways. It is also possible to combine them by executing certain instructions periodically and some others event-based.

3.7 Conclusion

Due to the increasing amount of heterogeneous devices, this chapter presents the approach of a swarm operating system that hides all heterogeneity beyond the system's interface and is responsible for resource management.

Cyber-physical systems require to cope with real space and time, in addition to concurrency and distribution, which make application development even more complicated.

Using the concept of systemic descriptions, application development is strongly facilitated and enables the programmer to specify spatio-temporal conditions. Based on this approach new kinds of applications are possible and have been presented in this chapter together with a classification of applications based on the space and time dimensions.

Chapter 4

Swarm Programming Model

After stating the vision of the swarm approach, new kinds of applications are possible. Such applications require a new, suitable programming abstraction. Thus, this chapter introduces a programming abstraction for cyber-physical systems. The emphasis is put on facilitating application development by relieving the programmer of error-prone aspects such as distribution, concurrency and motion. The abstraction should enable the programmer to specify *what* shall be done (the objective of a program or a part of it) rather than *how* it is achieved (every step that finally results in the solution of the objective). A program consists of high level instructions with input/output behavior. Those actions serve as basic building blocks and can be arbitrarily stuck together. Their execution context can be controlled by attaching spatio-temporal constraints.

4.1 Introduction

Since the invention of the computer, suitable abstractions have always been an important field of research. Abstractions are useful and sometimes even necessary in order to reduce complexity by dividing the problem space into smaller sub-problems, each of which is solved by a separate piece of code. The resulting functionality of that piece of code is provided to a layer on a higher level. In software development, it is recommended to keep the software modular such that each module addresses a clear defined problem and provides a clear defined functionality. This principle is also called separation of concerns and was first mentioned in [16].

Each module must have a clear defined interface in order to use the module's functionality. For instance, a suitable abstraction for communication in a software system should provide an interface with *send* and *receive* operations. This features transparency of the underlying communication system which may use different hardware components in order to establish wired or wireless communication. A well defined and well known model for communication systems is the OSI reference model [108], which has been invented in order to achieve interoperability. The model was originally designed with seven layers, each of which has a clear defined responsibility. Nonetheless, modularity also demands for code reuse. A suitable abstraction including an interface with suitable operations

must be invented such that the underlying implementation is both well optimized and exclusively reused by invoking its well defined operations. If, for some reason, the interface using developer requires more or different functionality other than provided by the interface, which leads to a (partial) re-implementation, states a non-well designed abstraction. Partial re-implementation is an error-prone concepts that should be, in any case, avoided.

Numerous programming languages based on different programming paradigms have been invented which provide individual levels of abstractions.

This chapter is organized as follows: Section 4.2 shows the state of the art of existing programming models. In order to design a new programming model for the swarm, Section 4.3 lists different language aspects for wireless sensor network programming abstractions. A selection of language aspects is chosen which influences the design of the programming model for the swarm. On this basis, Section 4.4 introduces the *swarm model*, which is the basic principle for this thesis, on a rather coarse-grained level. The model is further divided into the *capability/driver-model* (Section 4.5) and the *application/lib-model* (Section 4.6). All remaining chapters refer to different aspects that are stated in the swarm model. Section 4.7 shows how parts of the application code that is programmed using the *application/lib-model* are transformed into a dependency graph. Finally, Section 4.8 summarizes this chapter.

4.2 Related Work

There are different kinds of programming abstractions for distributed, concurrent and parallel systems as well as for sensor networks. Detailed surveys are provided in [94, 64]. Following a holistic approach, nesC [23], which is an extension to C, is a programming language for deeply networked systems which was created for TinyOS. Programs are built from components that have internal concurrency. While nesC is a node-level language (code is written for an individual node), Pleiades [46] provides an abstraction to implement a central program that has access to the entire network (also known as macroprogramming [104]). SpatialViews [65] is an extension to Java which allows to define virtual networks that are mapped to physical nodes according to their physical location and the services they provide. Execution is distributed among the nodes in the virtual network performed by code migration. Furthermore, it is possible to constrain execution based on timing restrictions.

In [12], a programming and execution environment for micro-aerial vehicle swarms is presented. Applications are a composition of low level drone behaviors and high level goals that are submitted by a user for execution on the swarm. The approach is behavior-based and, in contrast to the work presented in this thesis, does not feature the concept of application constraints in order to define spatial and temporal conditions.

4.3 Language Aspects

When designing programming abstractions for WSNs, different language aspects have to be considered and carefully selected. In [64], the authors have analyzed WSN applications and the applied programming abstractions. As a result, they produced the taxonomy that is shown in Figure 4.1.

On a coarse-grained level, programming abstractions are often categorized as either to use the concept of *node-centric* programming—programming based on the nodes perspective including communication—or macroprogramming—programming the network rather than the individual nodes. In order to provide a more detailed analysis about the kind of programming abstraction, the authors lists different language aspects in their taxonomy. These language aspects are described in the following. Afterwards, a composition of language aspects are selected which state the foundation of the programming abstraction for mobile robot swarms as presented in Section 4.4, which is one of the major contributions of this thesis.

4.3.1 Language Aspect Dimensions

A major issue in distributed systems is communication. Therefore, the taxonomy lists three aspects of communication: the communication *scope*, the type of *addressing* and communication *awareness*.

Scope The communication scope addresses the type and range of communication. In particular, it defines the reachability of arbitrary nodes from a dedicated node in a network. The authors further subdivided scope into three subgroups:

- *Physical neighborhood* Programming abstractions that allow physical neighborhood communication only allow to communicate with other nodes that are in direct radio range and thus, communication is limited to physical proximity.
- *Multi-hop group* Multi-hop significantly extends the communication range of the physical neighborhood approach by allowing communication amongst nodes that are in the same group that are several hops away from each other. Multi-hop group communication can be further subdivided into the following two categories:
  - *Connected* Connected describes the property that all nodes that are in the same group must be directly connected to each other. Therefore, every node in that group can reach any other node in the same group by only communicating over nodes also belonging to that group.
  - *Non-connected* Non-connected described the property that a group must not necessarily be connected, i.e., a node can communicate with another node of the same group by using other nodes that do not belong to that group.
- *System-wide* This enables communication system-wide, i.e., all possible subsets of nodes are able to communicate with each other.

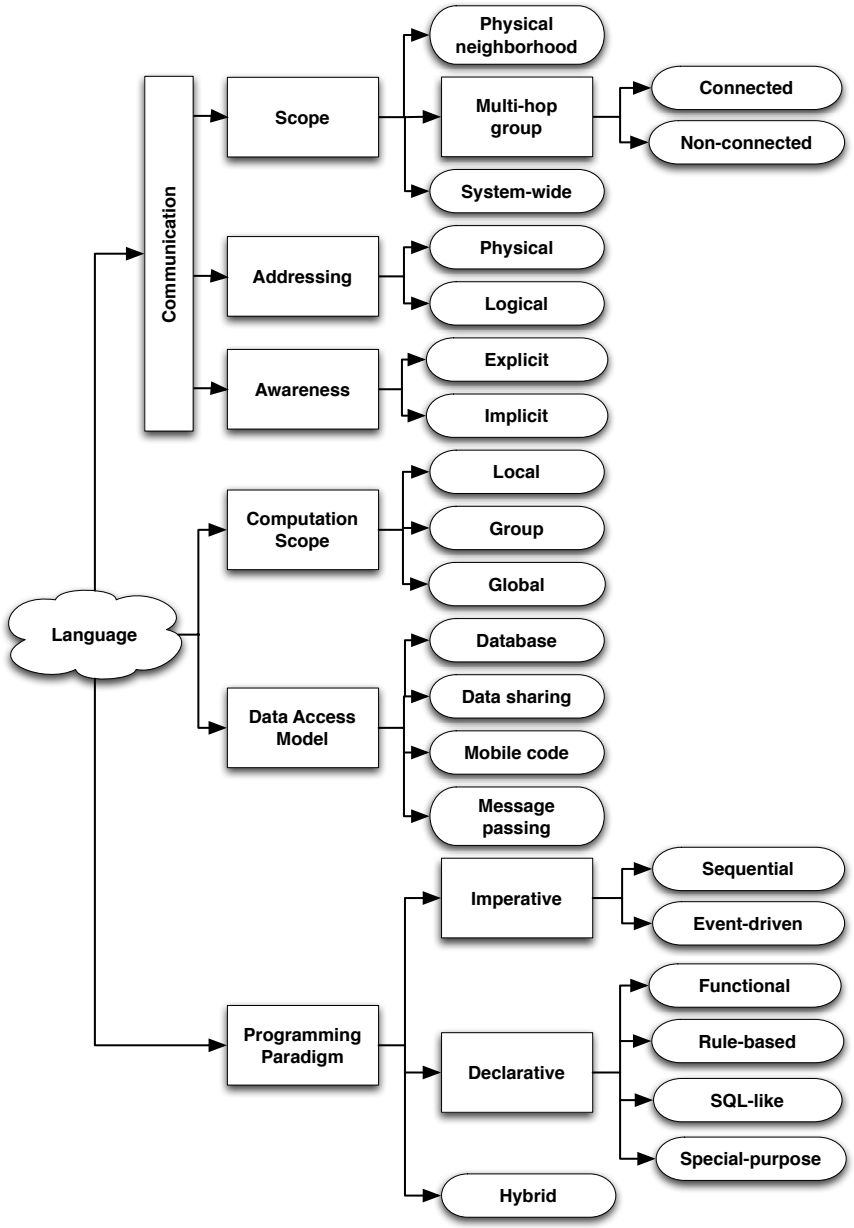


Figure 4.1: A taxonomy of language aspects in WSN programming abstractions (after Mottola [64]).

Addressing The second dimension describes the addressing which states the way a node is identified in the network. There are two categories:

- *Physical* Physical addressing means that a node is addressed by a static identifier that does not change over time, e.g., a MAC address or a physical memory cell.
- *Logical* Logical addressing usually occurs as an abstraction of the physical addressing by introducing an abstract identifier that is mapped to a node but with the characteristic that the mapping may change over time, i.e., the logical identifier points to a different node. This usually facilitates application development since certain complexity is hidden from the programmer and a larger degree of transparency has been established. In terms of memory cells, a logical address is an abstraction of the physical cell. A mapping function performs the translation from logical addresses to physical ones which is performed in paging [97].

  Another example, not for memory cells, but for sensor nodes, is the concept of *logical neighborhoods*, which has been presented in [62, 63]. Here, the approach is to define a logical neighborhood based on static, e.g., sensor type or dynamic, e.g., sensor readings, characteristics. Once defined, the programmer interacts with the logical neighborhood and not with the physical devices. The underlying runtime system is responsible for creating a mapping between both. The mapping can change over time, e.g., based on different sensor readings which is transparent for the programmer.

  In *abstract regions* [104] the concept is to create so called abstract regions that contain up to k-nearest neighbors around a certain node. Application programmers program the entire region rather than programming an individual node. The authors show a suitable example in order to track objects using the approach of abstract regions.

Awareness The third dimension in the communication aspect targets the awareness which states if programmers are aware of the communication or if the communication itself is transparent for the programmer. Awareness is divided into the two categories:

- *Explicit* Explicit communication describes that the programmer is completely aware of communication. In this case, programmers are responsible for the message generation as well as sending and receiving the message. This comprises, amongst others, the serialization and deserialization of messages, generating header information as well as interpreting header fields when receiving a message, calculating and attaching checksums, segmenting data into several message chunks as well as performing a reassembly. Furthermore, the programmer is responsible for handling asynchronous message delivery and, therefore, the programmer is responsible for establishing the correct message order and address fault tolerance.
- *Implicit* Implicit communication describes a higher level of abstraction by introducing transparency such that all the mentioned issues are hidden behind a certain

level of abstraction. The programmer is not even aware of communication, e.g., remote procedure calls (RPC) appear to the programmer as simple method invocation although the underlying framework has to perform all the message-based communication which either addresses in full or in partial the issues stated above, depending on the level of abstraction that the framework uses. An example for implicit communication is applied in the approach of *abstract regions* [104], in which programmers store and retrieve data items in shared variables. A shared variable is comparable to a tuple space. Dedicated *get*- and *put*-operations are provided to the programmer which perform the required message exchange and, thus, communication is transparent.

Computation Scope The computation scope defines the scope in which computation takes place. In particular, the scope describes the set of nodes that are affected by executing a single instruction. The computation scope is divided into the following three categories:

- *Local* Local states the situation in which only one node is involved in the execution of the instruction.
- *Group* The group scope describes that an execution of a single instruction leads to an execution on each node in the group and, thus, provides a higher abstraction compared to only local instruction execution.
- *Global* An instruction execution leads to a global execution, i.e., each node in the network executes the same instruction.

Data Access Model Besides computation and communication, a major aspect remains in the programming abstraction of how data is accessed. In the following four different approaches are described:

- *Database.* In the database oriented data access model, the network is treated as a global relational database. Programmers create SQL-like queries in order to request data. The statements are sent across the network where each node evaluates the statement locally and sends back its result. This is a simple, yet powerful, approach, designed to gather data in, especially, static sensor networks.
- *Data sharing.* Data sharing describes the concept of remotely accessible variables or tuples. The most well known approach are tuple spaces which allow to store and retrieve data in/from the tuple space.
- *Mobile code.* In contrast to data sharing in which data is usually accessed remotely, mobile code migrates the code that requests access to data onto the node on which the data resides, e.g., mobile agents. Thus, data is accessed locally.
- *Message passing.* Data is accessed by message exchange.

Programming Paradigm The programming paradigm has a large influence on the model elements of the programming abstraction and, thus, on how the application is developed. The data flow and flow control significantly depends on the programming paradigm.

- *Imperative.* Probably the most applied and well known paradigm is imperative programming. The programmer uses explicit instructions in order to change the program's state. Every step of an algorithm has to be mapped to instructions of the respective imperative programming language. The programmer has to explicitly specify *how* the program operates. Two sub-categories can be identified:
  - *Sequential.* Sequential programming is characterized by only having one thread of control. This thread sequentially executes the program's instructions. Doing so, facilitates programming since the programmer does not have to take error-prone aspects such as concurrency and synchronization into account.
  - *Event-driven.* In event-driven programming the control flow is mostly determined by events. The programmers install event handler in order to listen to events, e.g., a certain threshold has been exceeded. If an event occurs, the programmer is notified and is able to handle the event.
- *Declarative.* In the declarative programming paradigm, the programmer specifies the goal (*what* should be achieved) and not *how* it is achieved. This should support programmers in order to concentrate on the program's logic and minimize side effects. Declarative programming can be further subdivided:
  - *Functional.* Functional-based programming is based on evaluating mathematical functions. Common representatives of this category are, e.g., Mathematica or Haskell.
  - *Rule-based.* A rule-based system describes a system which contains *if then else*-statements. An example is an expert system. Rule-based systems are often used in artificial intelligence.
  - *SQL-like.* In this case, SQL-like queries are applied in order to gather data.
- *Hybrid.* Hybrid approaches combine both imperative and declarative programming paradigms by mixing code implemented in an imperative way with declarative aspects.

4.3.2 Language Aspect Selection

When designing a new programming abstraction, a careful selection of language aspects has to be taken into account. A main requirement for the programming model for swarms of mobile robots is to provide high level instructions such that the programmer is relieved from most complexity. The following explains the composition of language aspects that have been selected as a basis on which the programming model for swarms should be established:

Scope. The communication scope is set to *system-wide* and therefore, any subsets of nodes shall be able to communicate with each other. This is an important criteria since the programmer shall not care about communication at all. Instead, given the systemic description, it is up to the system to establish and handle communication. The programmer does not even take notice that a dedicated physical device exists. Indeed, the programmer is somehow aware that physical units exist that execute the code since this is simply required in a cyber-physical system, but the number, position and especially the technical properties of the devices are completely transparent for the programmer.

Addressing. The next communication dimension is addressing. As stated before, addressing can occur using a physical address or a logical address. While the physical address is static, i.e., the physical address always points to the same physical unit, there is a mapping from logical addresses to physical that may change over time. Similar to the concept of *logical neighborhoods* [62, 63] or *abstract regions* [104], the programming model for swarms of mobile robots used in this thesis is also based on *logical addressing* by specifying constraints which represent a logical unit and is then mapped to a physical device.

Awareness. The third dimension of communication targets awareness which can be further distinguished in implicit or explicit awareness. Since most complexity shall be hidden from the programmer, the goal is to keep communication *implicit*.

Computation Scope. As scope for computation, the programming model shall support both *local* and *group* computation. Therefore, it shall be possible to express an instruction for a single node as well as defining groups or formations similar to the concept of *abstract regions* but using spatio-temporal constraints.

Data Access Model. Data is accessed using *message passing*. However, this mostly done by underlying runtime system. Therefore, the programmer simply uses local method invocations in order to obtain data.

Programming Paradigm. The applied programming paradigm uses imperative programming for implementing the application's logic in an object-oriented way. Time and space aspects are specified as side conditions. In order to fulfill the conditions, a spatio-temporal coordination problem has to be solved which is transparent for the programmer. Therefore, time and space conditions are induced in a declarative manner leading to a hybrid approach.

4.4 Swarm Model

In order to support the development of applications according to the classification (Figure 3.7, page 23) while providing context awareness inside the application and guaranteeing important transparencies in distributed systems such as location-, motion- and

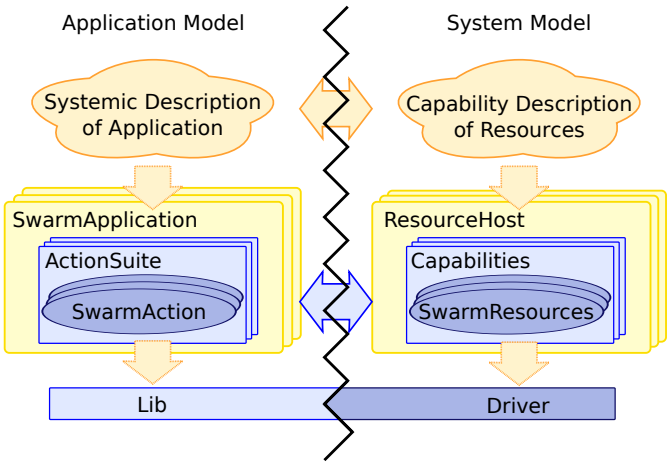


Figure 4.2: Swarm model.

distribution-transparency, the following swarm model as depicted in Figure 4.2 is presented. The model consists of a *system model* and an *application model*:

4.4.1 System Model

The system model addresses system developers who intend to extend the system by new capabilities or devices. The system model describes the capabilities and properties that the system provides. Each device has its own description of capabilities and properties that contribute to the global system. A capability uses up to multiple *SwarmResources*. A resource can be for instance *{Camera: Resolution: 1024x768, Color: RGB, FPS: 25}* or *{Temperature-Sensor: Range: [-50, +50], Resolution: 0.1}*. A capability is defined as a certain functionality that requires resources, e.g., *take picture* or *measure temperature*. Here, a 1:1 mapping between the capabilities and the resources exists. But there are also capabilities that require multiple resources, e.g. *take video* requires a camera and a microphone resource in order to deliver the audio and visual elements. Capabilities are implemented and accessible by a dedicated driver. A property describes the devices geometry, e.g., the shape of a robot and its maximum velocity. Stationary devices have a velocity of 0.

Definition 13 (ResourceHost) *A resource host is a device with certain capabilities and a description about its geometry.*

Definition 14 (SwarmCapability) *A capability is a well defined functionality that the system provides. Capabilities are associated with a certain resource host and can be accessed and used by applications.*

Definition 15 (SwarmResource) *A swarm resource is a physical resource, e.g., a sensor or an actuator with certain properties which is accessed and used by one or several capabilities.*

4.4.2 Application Model

The application model describes the programming paradigm that is based on a systemic description in order to develop swarm applications. The application model addresses application programmers who intend to develop new applications for the swarm. In the following, three programming entities are introduced that state the model: *SwarmApplication*, *SwarmAction* and *ActionSuite*. For simplicity, the terms *application*, *action* and *suite* are used synonymously in this thesis to *SwarmApplication*, *SwarmAction* and *ActionSuite*.

Definition 16 (SwarmApplication) *An application is called SwarmApplication if it is a program implemented using this application model and executed on the swarm system.*

Definition 17 (SwarmAction) *A SwarmAction is a certain instruction which is stated or issued by the application programmer. An action has to be mapped to a capability for execution. Each action is executed concurrently and independent of other actions. Furthermore, an action is executed on the "next suitable" resource host where the respective capability is hosted. Thus, a set of actions may be arbitrarily distributed among several nodes in a network. Actions may have input and output parameters in order to control the effect of an action or obtain its produced result. As basic building blocks, actions can be stuck together by connecting their inputs and outputs.*

Definition 18 (Spatio-Temporal Constraints) *Using the concept of spatio-temporal constraints, the execution of an action can be restricted in space and time. Constraints can be used in an absolute or relative manner.*

Definition 19 (SwarmActionSuite) *A SwarmActionSuite is a container for actions that logically belong together, i.e., a set of actions that fulfill a certain objective. Each action must be assigned to a particular SwarmActionSuite. All actions in the same suite may have arbitrary relative dependencies among each other. No dependencies between two actions of two different suites are allowed. Thus, all depending actions must reside in the same suite. If two actions are independent of each other, they can either reside in the same suite or in different suites.*

A SwarmApplication consists of multiple actions that are executed concurrently and independently and, if required, also in a distributed manner. All actions reside in a dedicated suite which allows intra-suite, but not inter-suite dependencies. Dependencies occur when input and output values of different actions are stuck together.

According to the *imperative programming paradigm*, an algorithm can be explicitly programmed using this application model by issuing actions and combining them in a certain manner. A simple program could be, for instance, to measure the temperature at a certain location and switch a control lamp to red in case a given threshold is exceeded. According to the programming model, first the actions have to be identified. In this case, there are two actions: a (measure temperature) and b (switch on LED).

Constraints	spatial	temporal	logical
absolute	x	x	
relative	x	x	x

Table 4.1: Constraint types.

Next, both actions have to be connected, e.g., $b.in = a.out$. This forms a logical dependency $b \rightarrow a$ (b depends on a). Based on the dependency, b requires data produced by a. If both actions reside in the same address space[1], this is trivial by simply communicating using shared memory.

However, if both actions are executed on different machines, data has to be exchanged using message passing. This is completely transparent in the programming model. The programmer only has to connect both actions and the underlying runtime system is responsible for exchanging data, either using shared memory or using message passing. Thus, communication is hidden behind the system's interface.

According to the *declarative programming paradigm*, certain side conditions can be assigned to actions that constrain their execution. This is done using spatio-temporal constraints that express a certain intention without stating how this is achieved. Spatio-temporal constraints are subdivided into the following categories as shown in Table 4.1.

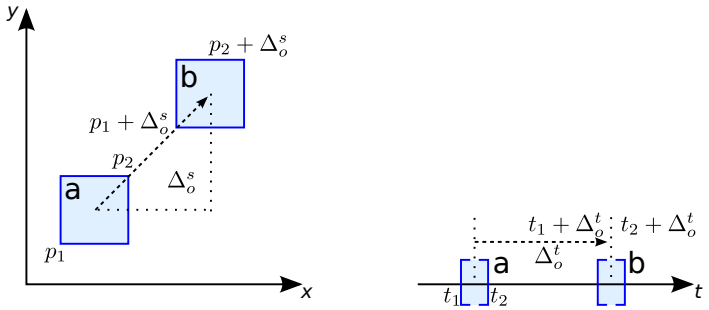
There are three types of constraints: *logical*, *spatial* and *temporal*. The types can further be subdivided into *absolute* and *relative* constraints. A *logical* constraint represents simple dependencies between actions. Connecting input and output of actions, e.g., $b.in = a.out$, forms logical dependencies that express predecessor-successor relations. This constraint type can only be used in a relative manner since it has to depend on another action.

A *spatial* constraint defines a physical space window in which the execution shall take place, e.g., $a \in [p_1, p_2]$, with p_1, p_2 being physical coordinates that span the space window. This constraint type can be both absolute and relative. While $a \in [p_1, p_2]$ is used in an absolute manner, $b \in a + \Delta_o^s$ states a relative spatial constraint depending on the absolute spatial constraint of a. Using relative constraints, the original spatial constraint is translated using the offset Δ_o^s: $[p_1, p_2]$ is translated to $[p_1 + \Delta_o^s, p_2 + \Delta_o^s]$ as depicted in Figure 4.3(a).

A *temporal* constraint defines a time window in which the execution shall take place, e.g., $a \in [t_1, t_2]$, with t_1, t_2 being absolute points in time. Temporal constraints can also be specified in a relative manner: $b \in a + \Delta_o^t$. The original temporal constraint $[t_1, t_2]$ is translated using the offset Δ_o^t: $[t_1, t_2]$ is translated to $[t_1 + \Delta_o^t, t_2 + \Delta_o^t]$ as depicted in Figure 4.3(b).

There is a *static* and a *dynamic* application model. The static model requires that all actions have been specified together with the spatio-temporal constraints before runtime of the program. In this case, the resource usage is static, i.e., the system knows already before runtime when and what kind of resources are requested.

[1] Actions are located in the same process address space, but might appear in different threads.



(a) Spatial constraints. (b) Temporal constraints.

Figure 4.3: Spatio-temporal constraints.

An offline scheduling could be applied before runtime which checks resource availability and is, therefore, able to allocate resources in advance, resulting in guaranteed resources. The drawback of the static version means less degree of freedom for the programmer since all possibly required resources have to be carefully considered before runtime. In the *dynamic* version dedicated mechanisms are provided that allow to react to certain circumstances. Therefore, programmers obtain full control of application behavior.

4.4.3 lib/driver-Concept

The set of available actions as well as the suites are encapsulated in the system library. The library (such as, e.g., the *libc* for the *C programming language*) provides a set of operations and data structures in order to program applications for the swarm. Those operations include system calls that enable to access capabilities (realized by drivers) in system space from user space.

Figure 4.4 shows the lib/driver concept which is divided into three layers: the *core-system*, *pluggable capabilities* and the *application space*. There is no direct coupling between a lib and a driver. The scheduler, which is a core-service of the runtime system, decouples invocations from the *lib*-side to the *driver* in space and time. The core-system is thereby the layer which represents the base system. The *proxy* and *skeleton* are the respective communication endpoints. They are responsible for serializing and deserializing remote procedure calls and performing the communication by message passing. The system can be extended by new capabilities and simply plugged in by using the existing stubs and skeletons. Thus, the second layer (*pluggable capabilities*) is extendable by new features. Capability developers only need to implement the functionality of the new capability which is then plugged into the system as *driver*.

Definition 20 (Lib) *The set of all actions in user-space together with the ActionSuite and dedicated data structures that the programmer is able to use is defined as the lib.*

Definition 21 (Driver) *A driver is the counterpart for a particular action, i.e., the implementation which actually performs the action.*

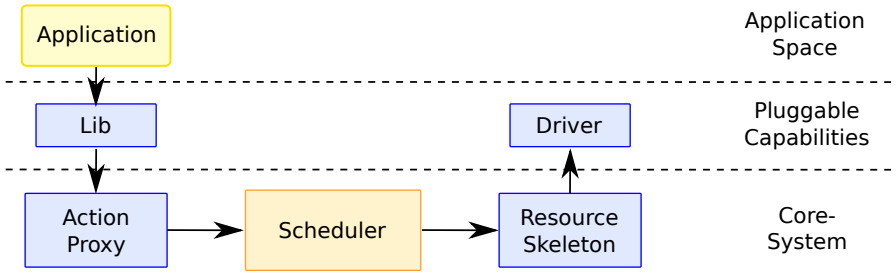


Figure 4.4: lib/driver-concept using loose coupling.

4.5 Capability/Driver-Model

New capabilities can be added "on the fly". System developers are able to add new features to the system by implementing new capabilities. Those can be added during the runtime of the system. Each resource host may have an arbitrary amount of capabilities.

4.5.1 Development

When developing new capabilities, the following steps have to performed:

1. The new capability has to be developed by simple functions.

2. A suitable counterpart (the *stub*) which is then a part of the swarm system library (*lib*) is automatically generated based on the capability description together with code connecting the capability to the system.

3. The capability is then deployed on dedicated nodes.

4. The system loads the new capability and plugs it in as *driver*.

Using separation of concerns, a capability developer only has to develop the desired functionality without addressing, e.g., communication. All communication, invocation and management tasks are performed by the core-system.

Listing 4.1 shows a simple capability: a temperate sensor. A capability must be inherited from the base class `SwarmCapability`. The two shown methods `init()` and `destroy()` are invoked once (at start time as well as in the shut down phase) in order to initialize and close resources. All functionality is implemented using simple functions that have to be declared with the `@Function` annotation in order to make it accessible from application space. In this example, one function (`measureTemp()`) has been implemented, i.e., accessing the local resource, obtaining the value, etc.

The second capability is an LED switcher as shown in Listing 4.2. The requester of this capability specifies an input value, the color, that an LED has to be adopted.

```
1 public class TempSensor extends SwarmCapability {
2
3   public void init() { /* configure .. */ }
4   @Function
5   public double measureTemp() { /* function .. */ }
6   public void destroy() { /* shut down .. */ }
7 }
```

Listing 4.1: Capability temperature sensor.

```
1 public class LED extends SwarmCapability {
2
3   public void init() { /* configure .. */ }
4   @Function
5   public void toggleLED(Color c) { /* function .. */ }
6   public void destroy() { /* shut down .. */ }
7 }
```

Listing 4.2: Capability LED switcher.

4.5.2 Code Generation

After the functionality is implemented, some additional code is required in order to connect the capability to the system. This is automatically generated. Listing 4.3 shows the generated `execute()` method that is responsible for dispatching between different functions of the same capability.

Each capability has an `execute()` operation that is, when needed, invoked by the core-system. Using the `getJob()` operation, further context data which is required for the current invocation is provided. The data structure `SwarmJob` contains all necessary input parameters: the function to execute as well as input parameters for the function itself. In order to provide the output of this driver implementation to the requester, the result is set using `setResult()`.

All communication, i.e., serialization and deserialization as well as message passing, is performed by the resource skeleton as depicted in Figure 4.4. The return value of the `execute()` method indicates the state of the execution of the driver functionality: success or error.

Listing 4.4 shows the generated code for the LED capability. Using `getParamValue()`, the input value is obtained. Both listings only show the generated code for dispatching, i.e., the code implemented by the programmer is neglected here.

Based on the capability implementation, the corresponding *lib* is generated as shown in Listing 4.5. For each function in the capability, a separate action is generated. The new action type must be inherited from `SwarmAction` which provides necessary management functions. Before the generation takes place, the capability is parsed for all functions that are declared within the particular class.

Each generated action becomes typed based on its return value of the respective function, if non-void. This type indicates the output of this action. In addition, the typifica-

```
public class TempSensor extends SwarmCapability {

  public boolean execute() {
    SwarmJob job = getJob();

    switch(job.getFunction()) {
      case MEASURE_TEMP:
        double temp = measureTemp();
        job.setResult(temp);            // output
        break;
    }
    // return true if success
  }
}
```

Listing 4.3: Generated dispatcher for TempSensor.

```
public class LED extends SwarmCapability {

  public boolean execute() {
    SwarmJob job = getJob();

    switch(job.getFunction()) {
      case TOGGLE_LED:
        // obtaining input parameters
        Color c = (Color) job.getParamValue(0);
        toggleLED(c); break;
    // return true if success
  }
}
```

Listing 4.4: Generated dispatcher for LED.

tion is used in order to create typed *future* objects as described in Section 4.6.7 (page 50). The constructors are generated as follows: the first parameter is the `SwarmActionSuite as` to which an instance of this action belongs. The second parameter allows spatio-temporal constraints in order to restrict execution in space and time.

However, in general, there are four constructors that will be generated: one that supports only spatial constraints, one that supports only temporal constraints, one that supports both and one that supports no constraints. For simplicity, the other constructors are neglected here. By viewing this generated code, it becomes obvious that, although the action is a temperature measurement and, thus, the measured temperature value is the only point of interest here, the *stub* does not provide such a return value. All output values are provided using event handlers which are explained in Section 4.6.

Listing 4.6 shows the generated code for the second capability, the LED switcher. Since `toggleLED(..)` requires an input parameter, the color that changes the state of the resource, the constructor obtains an additional parameter: the color. By instantiating this class, the programmer has to specify the color that the LED should adopt. The

```
public class MeasureTempAction
              extends SwarmAction<Double> {

  public MeasureTempAction(SwarmActionSuite as,
   Constraints... c) {
    super("TempSensor", as, c);
  }
}
```

Listing 4.5: Generated minimal-*stub* for temperature measurement.

```
public class ToggleLEDAction extends SwarmAction {

  public ToggleLEDAction(SwarmActionSuite as, Constraints... c,
   SwarmDataObject<Color> p1) {
    super("LED", as, c);

    this.addParam(p1);
  }
}
```

Listing 4.6: Generated minimal-*stub* for LED switcher.

remaining code is generated the same way as for the temperature measurement. The generated code shows that the input parameter (the `Color` object) has been wrapped into a `SwarmDataObject<T>`—a container class which is necessary for the internal management; `T` defines its type. The class `SwarmDataObject<T>` can not be instantiated since it is abstract. There are three inherited classes: `SwarmDirectDataObject<T>`, `SwarmTransferDataObject<T>` and `SwarmExpressionDataObject<T>`. The former one contains an explicit value while the second one only contains a reference to the actual value. This semantic allows the creation of *future* objects. The latter one allows the creation of conditional statements as shown in Section 4.6.7 (page 50).

4.5.3 Lifecycle

After the development of a new capability, it can be plugged into the system. This starts the capability lifecycle which is shown in Figure 4.5 and consists of three stages:

- *init:* The core-system invokes the `init()` method of the driver. During init, the required resources can be loaded, configured and checked if they function properly. In case of a successful initialization phase, the driver becomes registered in order to allow the scheduler to manage it. Each capability is initialized only once.
- *execute:* In case of a service request, the core system invokes the `execute()` method which leads to an execution of the driver functionality. Depending on the requester, the result is handled accordingly by sending it to respective destinations. The driver is usable as long as no `destroy()` has been triggered.

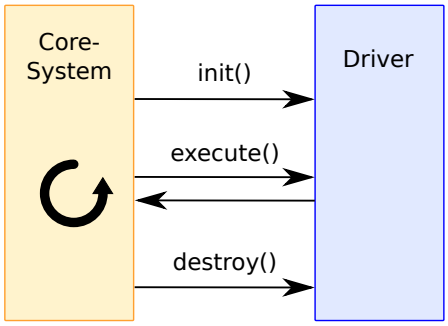


Figure 4.5: Capability lifecycle.

- *destroy:* If the capability should be removed from the system, the `destroy()` method is invoked. This unloads and closes all used resources.

The `init()` and `destroy()` method are optional and can be omitted by the programmer. Adequate default values are supplied in this case.

4.6 Application/Lib-Model

Similar to capabilities, new applications can be developed and deployed at any time. The core-system does not need to be stopped for that. The following shows some example applications.

4.6.1 Development

Developing SwarmApplications, the programmer should consider the following steps:

1. Logically specify the application's objective.
2. Split the resulting problem (in order to reach the objective) into actions.
3. Link depending actions (connecting input / output of actions).
4. Assign spatio-temporal constraints if necessary.
5. Put depending actions into the same action suite.
6. Install event-handlers if dynamic behavior is required.

Listing 4.7 shows a simple SwarmApplication that performs a plain temperature measurement. In line 4 the action suite is created. In line 5 the temperature measurement action is created. The first parameter is the action suite, i.e., action `a` belongs to suite `as`. Finally, in line 6, the operation `schedule()` is invoked on `as` which leads to a system call of the underlying runtime system which schedules the action by allocating a suitable

resource. After invoking the schedule operation, the state of the suite becomes "pending", i.e., a temporary state which is explained in detail in Section 4.6.5. Any additional invocation of `schedule()` is refused by the system. As explained in Section 4.6.2, adding more actions or constraints to the suite is still enabled. The call to `schedule()` is asynchronous. In particular, the invocation generates a dedicated message and sends it via message passing to the internal scheduler. Thus, the operation returns immediately without any result. This is done to avoid blocked procedures. As no spatio-temporal constraints have been specified in Listing 4.7, the temperature measurement could take place at any location.

```
1 public class AppTempMeasurement extends SwarmApp {
2
3   public void main(String[] argv) {
4     SwarmActionSuite as = new SwarmActionSuite(this);
5     SwarmAction a = new MeasureTempAction(as,.., null);
6     as.schedule();
7   }
8 }
```

Listing 4.7: Simple application.

If the dimensions of the network are very large in terms of physical space, measuring the temperature somewhere is usually useless. Therefore, Listing 4.8 shows how constraints are created. Line 1 creates an absolute space constraint that spans a rectangle between the points (x_1, y_1) and (x_2, y_2). Depending on the semantic of the application, it might also be necessary to create a temporal constraint (line 2), e.g., measure temperature at certain points in time or perform multiple measurements with some interval in-between. In this example, the action `a` shall be executed in the time interval $[t_1, t_2]$.

```
1 SpatialConstraints sc = new AbsoluteRectangleConstraints(x1, y1, x2, y2);
2 TemporalConstraints tc = new AbsoluteIntervalTemporalConstraints(t1, t2);
3 SwarmAction a = new MeasureTempAction(as, .., sc, tc);
```

Listing 4.8: Simple application extended by spatio-temporal constraints.

4.6.2 SwarmActionSuite Interface Operations

The programmer has different options in order to interact with the system. As shown in Listing 4.7 and Listing 4.8, the programmer is able to create new actions and let the system schedule them. If an already scheduled action becomes obsolete, e.g., due to a changing environment, the programmer can react accordingly by unscheduling or rescheduling a dedicated action. The programmer has access to the following system operations which are available at every instance of a `SwarmActionSuite`:

- `schedule()`: Schedules all actions that are contained in the ActionSuite in space and time. The suite may contain an arbitrary amount of actions.

- `unschedule()`: An unschedule operation requests the scheduler to remove all actions that are contained in this action suite by unscheduling them, i.e., performing a deallocation of allocated resources.

- `reschedule()`: If actions shall be added to an existing and already scheduled suite or spatio-temporal constraints shall be either modified or new constraints shall be added to existing or new actions, then the suite needs to be rescheduled. In this case, the programmer has to invoke the reschedule operation which requests the rescheduling of this suite.

In some scenarios, e.g., the three-sided observation which has been introduced in Section 3.4.4, implementing a certain aspect, in this case the three sided coordination, scheduling only one action is not sufficient, but rather a set of actions needs to be scheduled in order to reach a certain objective or partial objective. The outcome is a binary function:

$$f(x) = \begin{cases} 1, & \textit{pictures taken} = 3 \\ 0, & \textit{pictures taken} < 3 \end{cases}$$

Only if all three actions—that take one picture each—have been executed not violating their constraints, the outcome is 1, 0 otherwise. Thus, partial execution of a set of logically grouped actions is not desired by the programmer. In order to support a simple mechanism that enables the programmer to specify logically grouped action sets in which the programmer is interested in the common execution of all actions, the concept of *contracts* is introduced in Section 4.6.3. Each of the presented interface operations creates, when successful, a contract.

4.6.3 Contracts

A contract is an agreement between two parties that, once concluded, both sides have to stick to the conditions which are part of the contract. The concept of contracts is used here in order to create conditions between the user program and the underlying system. The advantage of using contracts is to have clear defined regulations: a program benefits from a contract since it has, once successful created, guaranteed resources which it can use and, hence, concepts such as the three-sided observation (Section 3.4.4) become feasible. On the other side, the system clearly knows where and when a resource is accessed and by which application, so accounting is performed precisely which is explained in the following.

Since the swarm system is used by multiple parties and to establish a certain level of fairness, a simple cost model can be assumed:

An application has to pay for the usage of a resource. The more resources are used, the more expensive it gets. Thus, the application's behavior, in terms of resource usage, shall be minimalist in relation to achieving its objective: the application should not use more resources than necessary. This situation is realizable with contracts. A contract states quantitative and qualitative aspects about the resource usage. After the contract

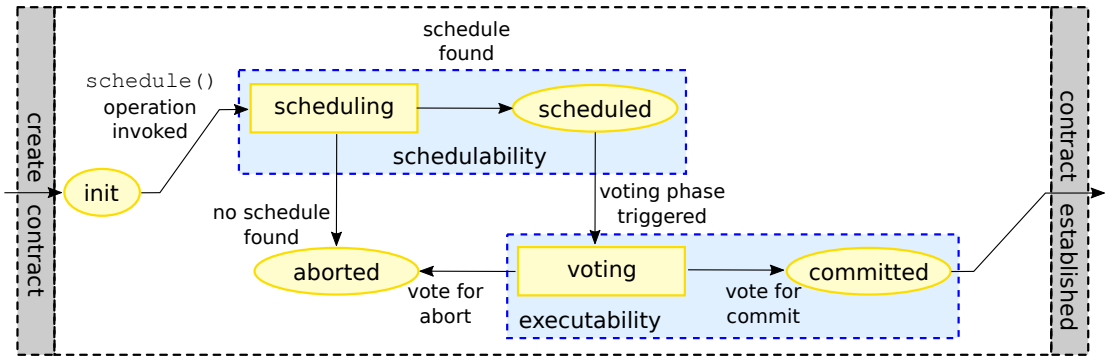


Figure 4.6: Contract creation lifecycle state space.

has been concluded, the application as well as the system are aware of the costs that originate from using the resources. The "trading" takes place when the contract is actually applied, i.e., the resources are used. This avoids that applications try to claim to much resources.

The system provides that all its system operations which are presented in Section 4.6.2 are contract-based, i.e., the application obtains guaranteed resources. Each of the operations start a new contract and first check if the contract can be realized from the system's side (sufficient resources). If successful, the contract is created. This is explained in detail in Section 4.6.4.

4.6.4 Contract Creation Lifecycle

Every contract creation phase has a lifecycle. Figure 4.6 shows the state space of the lifecycle. The call to `schedule()` is asynchronous and message passing is used in order to communicate with the scheduler. Once the scheduler has obtained a suite with all containing actions and their dependencies, it starts to schedule them in space and time. Since a new contract shall be created, the non-schedulability of one action leads to an abort of the entire suite, i.e., all actions that belong to the same suite will also be aborted. During contract creation, the following two predicates have to be checked and verified in order to successfully create the contract:

Definition 22 (Schedulability) *The scheduler checks general schedulability of the actions contained in the ActionSuite (short: sched()). So, the scheduler checks if all required resources can be made available under the given spatio-temporal constraints (this also includes physical movement). In case the scheduler successfully found a schedule, then the suite is called schedulable. Otherwise the suite is not schedulable.*

Definition 23 (Executability) *Schedulability is a necessary condition for executability (short: exec()). Executability is the state in which all nodes (which are involved in the distributed execution of the ActionSuite) jointly voted for commit and, thus, state*

that they are able to execute their assigned actions. The voting protocol is explained in Section 5.5.1.

- *init:* If a new contract shall be created, its initial state is the init state. A call to `schedule()` triggers the scheduling of all actions that are part of that suite. If the scheduler found a schedule that incorporates all actions, then the state changes from *init* to *scheduled*. However, if no schedule could be found then the state is changed to *aborted*. The scheduling is successful if all actions in that suite could be assigned to resources satisfying the constraints. If at least one action could not be scheduled, then the entire scheduling for that suite fails. In the aborted state, the creation of a new contract has *failed*. In the *scheduled* state, the predicate *schedulability* becomes true.

- *scheduled:* The scheduled state reflects that a given set of actions has been successfully scheduled, i.e., there are sufficient resources available for all actions in the suite. As stated in Definition 22, schedulability does not guarantee that the set of actions in the suite are also executable (Definition 23). A distributed voting phase is triggered in order to determine if a schedulable suite is also executable. The voting operation results either in the *aborted* or in the *committed* state.

- *aborted:* If a suite is either not schedulable (due to insufficient system resources) or it is schedulable, but not executable, then the state is set to aborted. The aborted state reflects that the contract has been canceled, i.e., all contained actions have either already been canceled or they are in the canceling process.

- *committed:* Once a suite is schedulable and executable, i.e., all participating nodes have commonly agreed to execute the actions, the suite is committed which indicates *success*. As a result, the respective contract has been created.

4.6.5 SwarmActionSuite Lifecycle

An ActionSuite has different states. The state space is shown in Figure 4.7. A transition from one state of the state space to another is triggered by invoking one of the interface operations described in Section 4.6.2. Section 4.6.4 shows the lifecycle that has to be performed in order to create a new contract. This involves to check *schedulability* and *executability* of all actions in a suite. Therefore, it is called contract creation lifecycle while this section shows the different states of the suite itself. The operations shown in Figure 4.7 (re-/un-/schedule) go through the entire contract creation lifecycle, respectively. The following describes the different states that the suite can adopt:

- *init:* Once created, a suite is in its *init* state. In its init state, actions can be added to the suite and constrained with spatio-temporal constraints. Once done, a call to `schedule` triggers the contract creation lifecycle as described in Section 4.6.4. In case the contract creation fails, the state changes from init to *aborted*. If a contract has been created, the state changes to *scheduled*.

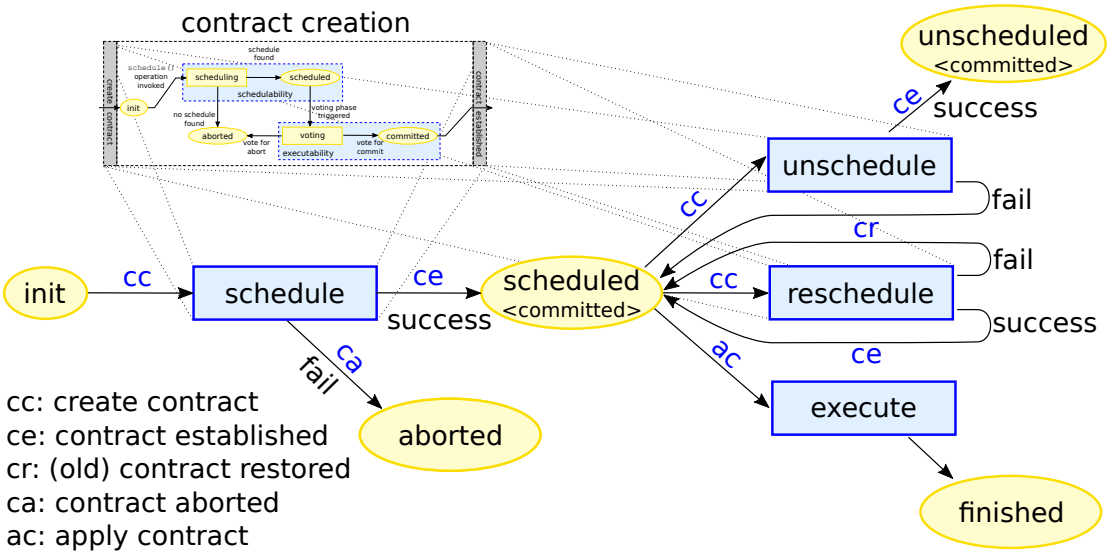


Figure 4.7: ActionSuite lifecycle state space.

- *aborted:* The aborted state is a final state, which indicates that no contract has been created.
- *scheduled:* The scheduled state is a state in which a schedule has been already committed and, thus, a contract has been created. Dedicated system resources are granted and allocated to this suite. The system can not revoke a contract. However, from the scheduled state three further transitions are possible: *unschedule*, *reschedule* and *execute*. While the former two operations have to be explicitly triggered by the programmer, the latter is implicitly triggered by the system. At every point in time only one transition can be active. Adding new actions, modifying existing constraints or adding new constraints to a suite requires a rescheduling. Performing a rescheduling tries to establish a new contract with the system by replacing the current one. This can, therefore, be also interpreted as a re-negotiation about system resources. The scheduler attempts to find a new schedule that considers the modified actions and constraints. In case of success, a new contract is created which replaces the current one. If the scheduling fails, the new contract can not be created. In this case, the current contract is simply restored or remains active. Independently of the result the transition *reschedule* always changes the state from scheduled to scheduled and, thus, a rescheduling always leads to a committed and active schedule.
- *unscheduled:* In the scheduled state the transition *unschedule* starts an attempt to free all allocated system resources of the suite. The scheduler is requested for performing the deallocation. If possible all actions become unscheduled and are removed from the current schedule which indicates success. This changes the state from *scheduled* to unscheduled. Doing so, a new contract that replaces the

current one is established. The new contract is the zero contract in which no conditions are listed and, thus, both parties have no obligations. Unscheduled is a final state. If the unscheduling fails, e.g., time has already progressed such that the remaining time for performing the unscheduling process including the notification of the involved nodes is too short, the unscheduling fails. In this case, the current contract is restored and the scheduled state remains active with all obligations that have been concluded.

- *finished:* From the *scheduled* state using the *execute* transition changes the state to finished by applying the contract. This transition is chosen implicitly and cannot be forced by the programmer. Since all actions in a suite in the *scheduled* state have certain execution context parameters (concrete points in time and physical machines) those actions will be executed according to the schedule. Once executed, the state changes from *scheduled* to *finished*. This is the final state of a suite. The suite then has been successfully executed.

4.6.6 Event Model

In order to be aware of state changes or obtaining output values of actions, the programmer can install *event handlers* on different levels. There are event handlers that indicate the progress state of contracts as well as ActionSuites. Since actions are executed asynchronously, it is also necessary to install event handlers on the action level in order to read output values. The event handlers are shown in Listing 4.9. There are different event listeners on three different application levels:

- **Action-Level**: It is possible to install listeners on different levels. On a fine-grained level, listeners can be installed for every action that has been created. There are two kinds of action listeners: `SwarmActionListener` and `SwarmActionResultListener`. The former one only provides the `OnFinish()` event handler. `SwarmActionResultListener` is inherited from `SwarmActionListener` and extends it by adding the `OnData()` event handler. The `OnFinish(..)` handler is invoked if the associated action has been executed. By installing the `SwarmActionResultListener`, it requests the output of the action to be sent to this node, i.e., the one where the code is currently executed. `OnData(..)` is invoked with the respective data. This has no influence on others actions that depend on this action. In this case the data is sent to both data sinks. Since installing the `SwarmActionResultListener` explicitly requests data, this should only be done when needed in order to avoid additional communication.

- **ActionSuite-Level**: After the `schedule()` system call is invoked, four types of events can occur. The suite produces no data, and, thus, the events notify about the current state of the suite's lifecycle including the contract creation lifecycle. By installing the `SwarmActionSuiteListener`, the program can be notified in the following way: Once the action suite is scheduled `OnScheduled(..)` is invoked. That is when the scheduler checked schedulability and, hence, the state changes

```
1  ActionSuite as = new ActionSuite();
2  Action a = new Action();
3
4  a.addActionListener(new SwarmActionResultListener() {
5    public void OnFinish(UUID jobId) {  }
6    public void OnData(Object data) {  }
7  });
8
9  as.addListener(new SwarmActionSuiteListener() {
10   public void OnScheduled(ScheduleResult rs) {  }
11   public void OnCommit(UUID tid) {  }
12   public void OnAbort(UUID tid) {  }
13   public void OnFinish(UUID tid) {  }
14 });
15
16 as.schedule();     // system call
```

Listing 4.9: Example with event-listener.

to *scheduled*. If no schedule could be found and the state changes to *aborted*, `OnAbort()` invoked. If schedulability and executability is guaranteed, the contract is established and the state changes to *committed* which leads to an invocation of `OnCommit()`. However, since `reschedule()` and `unschedule()` go through the exact same lifecycle for creating a new contract, the order in which the event handler are invoked does not change. For instance both operation first lead to an invocation of `OnScheduled` in case of success. The provided parameters then include the current context information about the new schedule which is either an updated schedule or the zero schedule. In case that all actions in the suite have been executed, the state changes to *finished* and `OnFinish(..)` is invoked.

- **Application-Level**: On a coarse-granular level, it is possible to install the `OnAppFinish(..)` listener. This listener is called when all application threads and action threads have been executed. If the application goes into this state, the creation of additional actions is forbidden.

4.6.7 Dependent Actions

Until now, the shown examples only contained one action. In most cases the application consists of more than one action. An example could be to measure the temperature and switch an LED that indicates if the temperature has exceeded a certain threshold—a simple sense-and-react application. For this, two capabilities are needed that have been introduced in Section 4.5 (page 39). This could be simply done by using the programming model and creating two actions `a` and `b` where `b` depends on `a` as shown in Listing 4.10. The first action (`a`) measures the temperature. Using the event model, an event listener is installed on action level in order to obtain the output value, i.e., the measured temperature value. `OnData(..)` delivers that value. After the value is obtained

```
1  public class TempControl extends SwarmApp {
2
3    public void main(String[] argv) {
4      SwarmActionSuite as = new SwarmActionSuite(this);
5      SwarmAction a = new MeasureTempAction(as, ..);
6
7      a.addActionListener(
8                new SwarmActionResultListener() {
9        public void OnFinish(UUID jobId) {       }
10       public void OnData(Object data) {
11         handleData(data);
12       }
13     });
14     as.schedule();
15   }
16
17   private void handleData(Object data) {
18     SwarmActionSuite as = new SwarmActionSuite(this);
19     Color c = (data < THRESHOLD) ? GREEN : RED;
20     SwarmDataObject s =
21           new SwarmDirectDataObject<Color>(c);
22     SwarmAction b = new ToggleLEDAction(as, .., s);
23     as.schedule();
24   }
25 }
```

Listing 4.10: 2 depending actions and the use of event handler.

(line 11), the method `handleData(..)` creates the second action. Here, another suite is created to which `b` is added. Depending on the result of `a`—if the value is under a given threshold—the color of the LED is switched. Since the programming model is based on asynchronous actions and asynchronous system calls, there is no blocking behavior in the execution phase. After invoking the schedule operation on `as` which contains `a`, the method returns immediately and the thread which executes the main program is done. The second action is created after `a` has been executed. Installing an event handler for action `b`, on suite level or on application level is possible, though not necessary.

The example shows the usage of two separate suites, i.e., two contracts are created. Since it is possible that the contract creation fails due to insufficient system resources the following scenarios are possible:

- Two contracts are created (`a` and `b` are executed)
- No contract is created (neither `a` nor `b` are executed)
- One contract is created (`a` is executed, but not `b`)

Since the creation of `b` requires that `a` has been successfully executed, creating only a contract for `b` is not possible. However, if the programmer intends that both actions

```
1  public class TempControl extends SwarmApp {
2
3    public void main(String[] argv) {
4      SwarmActionSuite as = new SwarmActionSuite(this);
5      SwarmAction a = new MeasureTempAction(as, ..);
6      SwarmAction b = new ToggleLEDAction(as, ..,
7                  a.getResultRef());
8
9      as.schedule();
10   }
11 }
```

Listing 4.11: Simple application extended by event handler.

are executed, both actions have to reside in the same suite as depicted in Listing 4.11. Putting both actions in the same suite provides the following advantages:

- Less code: the programmer has to implement less code in contrast to the version shown in Listing 4.10.
- Less communication: the output of `a` is directly sent to `b`.
- No event handler: no event handler has to be installed which results in a simple sequential program implementation.

In this example, both actions are created at the same time. Since the actual result of `a` has not been obtained yet, `b` gets a reference rather than the actual value. The method `getResultRef()` is implemented in the base class `SwarmAction`. The return value is a `SwarmTransferDataObject<Double>` which is typed at runtime based on its instantiation, i.e., the respective action. Since the method is invoked on `a`, this method will return a proxy for a `Double`-value (according to the generated stub in Listing 4.5). This concept is related to the concept of *futures* [4].

By invoking the schedule operation, both actions are scheduled with the side condition that `b` logically depends on `a`. In this case, the result of `a` is directly sent to `b`. This scenario requires a modification of the presented driver from Section 4.5 since the driver currently awaits a color as input. A subsequent driver modification is not desired. Therefore, in the third variant, the concept of conditional statements is introduced.

Listing 4.12 shows an example in which a conditional statement is created (lines 6 and 7). The statement is created on action `a` and is interpreted as follows:

```
if (a < THRESHOLD) THEN color = GREEN ELSE color = RED
```

Using lazy evaluation, the statement is not evaluated after creation, but exactly before execution of `b`. This concept enables to implement such logic constructs inside the application and, hence, from the application programmer, but postpone the evaluation. Furthermore, no driver modifications have to be performed.

Spatio-temporal constraints have been omitted for simplicity so far. According to Listing 4.8 (page 44), all actions can be further constrained.

```
1  public class TempControl extends SwarmApp {
2
3    public void main(String[] argv) {
4      SwarmActionSuite as = new SwarmActionSuite(this);
5      SwarmAction a = new MeasureTempAction(as, ..);
6      SwarmExpressionDataObject<Color> e =
7        a.createExpression("<", THRESHOLD, GREEN, RED);
8
9      SwarmAction b = new ToggleLEDAction(as, .., e);
10
11     as.schedule();
12   }
13 }
```

Listing 4.12: Simple application extended by event handler.

4.6.8 Application Lifecycle

Applications consist of concurrent actions as described in Section 4.4.2 (page 36). The execution model is comprised of a *main-thread* (the thread which executes sequentially the program's instructions) and up to several *action-threads* (threads that execute actions concurrently). The execution model is hidden from the programmer.

Each of the threads has its own lifecycle. The composition of lifecycles forms the application lifecycle as shown in Figure 4.8. The lifecycle starts in the *init*-state of the main-thread. Once the instructions in the application are being executed, the state changes from *init* to *local computation*.

In this state, new actions can be spawned. If so, a new ActionSuite is created (containing up to multiple actions) and scheduled by the system scheduler. The dashed arrows indicate the spawning of a new action. The state of the current thread does not change. It creates another thread with a new spawned state.

If all instructions of the main-thread have been executed, the state changes from *local computation* to *local computation finished*. In this state, the application "waits" for its termination. If no actions have been spawned or all of them are already executed, then the state changes to *Application finished*. This is a final state and marks the application as completely executed.

Besides the main-thread an arbitrary amount of actions-threads may exist (locally or distributed across the network). All actions have their own states. Once an action is spawned, it will be executed, at some point in time. This changes the state to *execute*. If all actions in the same suite are executed, the state changes to *ActionSuite finished*. If all ActionSuites have been executed, the state changes to *All ActionSuites finished*. If the main-thread is also in its *local computation finished* state, then the state changes to *Application finished* which indicates that the application is completely executed.

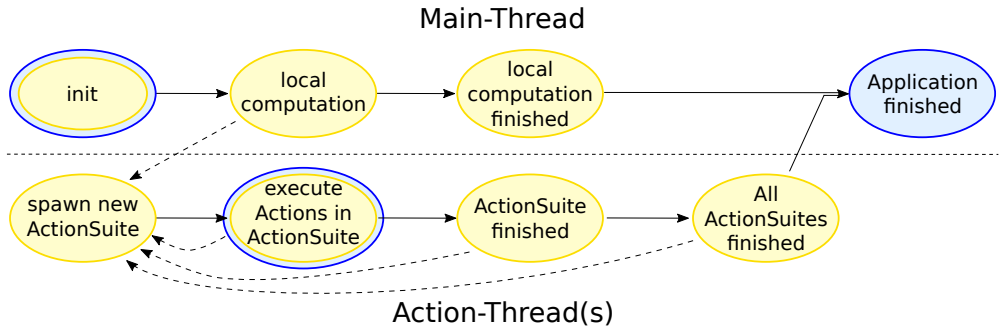


Figure 4.8: Application lifecycle state space.

4.7 Dependency Graph Generation

The specification of actions together with their associated constraints and interconnections form dependencies. Referring to the *lib/driver* model (Section 4.4.3), the dependency management as well as sanity checks are a part of the *lib* and, thus, are performed locally.

A dependency graph is generated in which each action is represented as a vertex with its absolute temporal and spatial constraints as properties. In particular, for each constraint type a dedicated directed, acyclic dependency graph is generated. Since there are three types (logical, spatial, temporal), three graphs are created: the spatial and temporal graph additionally have weighted edges. All of them reside in the SwarmActionSuite and are created upon instantiation. The following explains the three dependency types:

- *Logical dependency:* A logical dependency is created if an action shall depend on an already existing action. A logical dependency is written as $b \rightarrow_l a$ and states that the output of a is required as input for b. There are two operations that create a logical dependency:
  - `b = new SwarmAction(.., a.getResultRef())`
    creates $b \rightarrow_l a$.
  - `b.addLogicalDependency(a)`
    creates $b \rightarrow_l a$.
- *Temporal dependency:* A temporal dependency specifies a temporal relation between two actions: $b \xrightarrow{t^*}_t a$. In this case, b shall be executed t^* time units after a with $t^* > 0$. A relative time constraint is added by:
  - `b.addTimeConstraint(a, new SimpleTime(5))`
    creates $b \xrightarrow{5}_t a$.

- *Spatial dependency:* A spatial dependency specifies a spatial relation between two actions: $b \xrightarrow{(x,y)}_s a$. In this case, b shall be executed x space units in x- and y space units in y-direction apart from a. For this, the city block distance[2] is used.

  - `b.addSpaceConstraint(a, new CityBlockDistance(5,-5))` creates $b \xrightarrow{(5,-5)}_s a$.

All three graphs must be acyclic. Each of the above mentioned instructions triggers a checking of all three graphs. In case a cycle has been detected the last modification is reverted and the constraint is rejected.

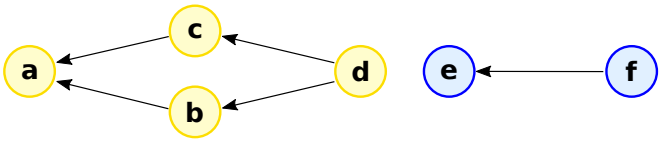
4.7.1 Scheduling a New ActionSuite

In the following, the construction of the dependency graphs are explained according to a small example. Listing 4.13 shows the creation of six actions (`a, .., f`). In this example, it is abstracted from the concrete action type and instead it is only referred to as `SwarmAction`. Every suite must have at least one independent action. Here, action `a` is independent since it does not depend on other actions. They are also called *root* actions. The composition of action `a, .., f` (line 4-10) lead to the generation of the

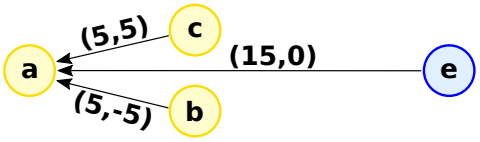
[2]City block distance is also called Manhattan distance or taxicab metric is a rectilinear distance (L_1 distance or ℓ_1 norm).

```
1  SwarmAction a, b, c, d, e, f;
2  SwarmActionSuite as = new SwarmActionSuite(this);
3
4  a = new SwarmAction(as);
5  b = new SwarmAction(as,a.getResultRef());
6  c = new SwarmAction(as,a.getResultRef());
7  d = new SwarmAction(as,b.getResultRef(),c.getResultRef());
8
9  e = new SwarmAction(as);
10 f = new SwarmAction(as,e.getResultRef());
11
12 a.addTimeConstraint(new TimeInterval(1240, 1260));
13 a.addSpaceConstraint(new Rectangle(10,10,20,20));
14
15 b.addSpaceConstraint(a, new CityBlockDistance(5,-5));
16 c.addSpaceConstraint(a, new CityBlockDistance(5, 5));
17 e.addSpaceConstraint(a, new CityBlockDistance(15,0));
18
19 c.addTimeConstraint(b, new SimpleTime(0));
20 f.addTimeConstraint(e, new SimpleTime(5));
21
22 as.schedule();
```

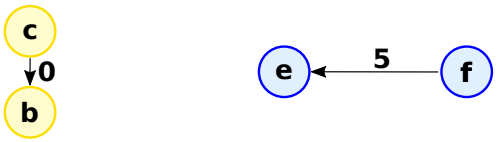
Listing 4.13: Actions with mixed dependencies.



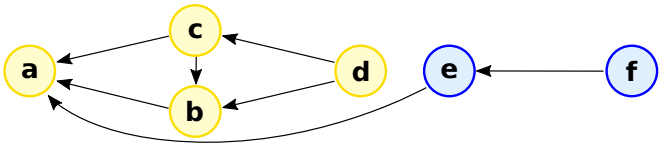
(a) Logical dependencies between actions (a, .., f).



(b) Spatial dependencies between actions (a, .., f).



(c) Temporal dependencies between actions (a, .., f).



(d) Meta graph of actions (a, .., f).

Figure 4.9: Dependency graphs for actions (a, .., f).

logical dependency graph which is depicted in Figure 4.9(a). The graph is not connected. Figure 4.9(b) shows the relative spatial dependencies between the actions. They are expressed using the city block distance (line 15-17).

Finally, Figure 4.9(c) shows the relative temporal dependencies. Action `b` and `c` are constrained to be executed at the same point in time, but at different locations. Using this set up all actions have to reside in the same suite. Figure 4.9(d) shows the combined (meta) graph which includes all dependencies.

4.7.2 Rescheduling an Existing ActionSuite

In the *init* state (Section 4.6.5, page 47), the only allowed operation is `schedule()` which requests the scheduling of the suite. After the invocation, the suite is in a *pending* state. A modification of constraints and/or actions of a suite is allowed at any time. System calls are disabled in the pending state. Once the suite is in *scheduled* <*committed*> state, the operations `reschedule()` and `unschedule()` become available.

Listing 4.14 shows a code fragment which is an extension of Listing 4.13. In line 3 a new action `g` is added which creates a logical dependency to `e`. Afterwards, the action

```
1  SwarmAction g;
2
3  g = new SwarmAction(as,e.getResultRef());
4
5  as.remove(d);    // removes action d
6
7  f.addSpaceConstraint(e, new CityBlockDistance(5,-5));
8  g.addSpaceConstraint(e, new CityBlockDistance(5, 5));
9
10 g.addTimeConstraint(f, new SimpleTime(0));
11
12 as.reschedule();  // rescheduling
```

Listing 4.14: Reschedule ActionSuite.

`d` together with its logical dependencies to `b` and `c` is removed. The updated logical dependency graph is depicted in Figure 4.10(a). In lines 7 and 8, a new relative spatial constraint is added for `f` and `g`, respectively. The updated spatial dependency graph is depicted in Figure 4.10(b). In line 10 a new relative temporal constraint is added which creates a dependency from `g` to `f` expressing that both actions shall be executed at the same point in time. The updated temporal dependency graph is depicted in Figure 4.10(c).

Finally, in line 12, `reschedule()` is invoked which requests a rescheduling of this suite if the suite is in *scheduled* <*committed*> state only. As long as the operation is not completed, the suite is again in a pending state. System calls are, thus, disabled again. The impact of the operation is either that a new contract is created which comprises the modifications of the suite or the old contract is restored in case of non-schedulability.

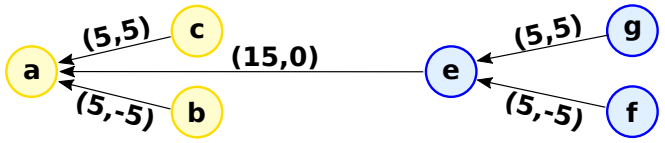
An action can only be removed from a suite if no other action depend on it. Therefore, the attempt of removing `a` (`as.remove(a)`) would not succeed since other actions (`b`, `c` and `e`) depend on it as depicted in the combined (meta) graph in Figure 4.10(d). Cascading removal of actions is by design not supported.

However, a great advantage in using contracts here is that the programmer is aware that, at least, one contract is obtained after the operation has been completed.

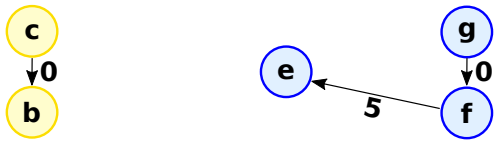
4.7.3 Unscheduling an Existing ActionSuite

If a contract for a given suite shall be removed, the suite must also be in *scheduled* <*committed*> state. If so, the operation `unschedule()` requests the removal of all scheduled actions. If this succeeds, all contained actions are removed from the internal schedule, the contract is discarded and the suite gets into the *unscheduled* <*committed*> state. The entire suite including the dependency graphs is marked for removal.

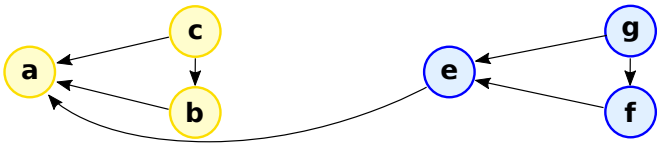
(a) Logical dependencies between actions (a, .., g).



(b) Spatial dependencies between actions (a, .., g).



(c) Temporal dependencies between actions (a, .., g).



(d) Meta graph of actions (a, .., g).

Figure 4.10: Dependency graphs for actions (a, .., g).

4.8 Conclusion

This chapter presented the swarm programming model. After showing the state of the art of programming models, Section 4.3 discussed language aspects in wireless sensor networks whereas Section 4.3.1 focused on different dimensions and Section 4.3.2 stated which language aspect dimensions shall be considered for designing the new programming abstraction for the swarm. Section 4.4 - 4.6 explained the core of the programming model while Section 4.7 showed how dependencies are handled internally when using the interface operations. This chapter is concluded by stating a summary about the features of the programming model:

- *No blocking:* The entire model is designed such that no blocking procedure calls appear. Executing actions as well as performing system calls are asynchronous.
- *Distribution transparency:* Each action may be executed on different nodes which might be even necessary according to given spatio-temporal constraints. However, this is fully transparent for the application programmer.

- *Concurrency transparency:* The concept of threads and concurrent execution is hidden from the programmer and suitable event handlers are provided in order to obtain status information or data produced by actions.
- *Implicit communication:* All communication is hidden from the programmer and deeply buried inside the system.
- *Systemic description:* Using the programming model, the programmer is able to specify
  - *What* shall be done: a certain objective which is expressed as a composition of single instructions (actions) that are grouped into suites according to their dependencies.
  - *Where* and *when* it shall be done: declarative annotated side conditions which state the execution context using spatio-temporal constraints
- *Resource reservation:* Based on the concept of contracts, the programmer is able to perform resource reservation according to the spatio-temporal constraints in advance. The system either confirms or rejects a contract.
- *Rebooking:* If already obtained, a contract can be arbitrarily modified which expresses the dynamic of the approach. Modifying a contract requires, first, a modification of the specification and, second, an approval of the system. Once confirmed, the new contract is established.
- *Transparent movement:* If an application requires that robots have to move in physical space, the movement (steering, navigation) is completely transparent. Also, the type of robot (flying, floating, grounded) is transparent.
- *Less Code:* Only few lines of source code are sufficient in order to program certain functionality as expressed in the examples in this chapter. The error-proneness is strongly reduced.

Chapter 5

Swarm Runtime System

In order to execute swarm applications, a dedicated distributed infrastructure is required. This chapter addresses this issue and presents a service-oriented architecture for the swarm runtime system. There are two types of services: local services are executed on every node that is part of the system and are required for local node management, i.e., accessing sensors and actuators. Global services are executed on some node of the system and are used for global system management which includes resource allocation and scheduling. As usual in computer systems, a certain allocation of resources produces a load of the system, i.e., a characteristic that states to which extent the resources of the system are being utilized. Therefore, this chapter also introduces metrics in order to determine the system's utilization.

5.1 Introduction

In order to support the execution of swarm applications developed according to the proposed programming model, a distributed swarm infrastructure is required that provides efficient, non-conflicting resource allocation and management as well as synchronization and coordination.

This chapter is organized as follows: Section 5.2 introduces the concept of swarm virtualization and action management. Section 5.3 presents the distributed architecture of the runtime system. Section 5.4 defines the system utilization. In Section 5.5 the operation of the system including an applied variant of the two-phase commit protocol is presented. Section 5.6 shows the system interface. Finally, Section 5.7 summarizes this chapter.

5.2 Action Management and Swarm Virtualization

In order to enable multi-program operation on the swarm, virtualization on resource level is used. A programmer has no direct access to a physical resource. Actions require certain resources for a particular amount of time. Programmers create actions (with

spatio-temporal constraints) on virtual resources. A virtual resource is mapped to a physical resource at runtime by the system scheduler. Given a set of actions on virtual resources (from possibly multiple applications), the scheduler computes a spatio-temporal mapping from virtual to physical resources. A physical resource is, hence, used in a time sharing manner.

The sum of all actions on virtualized resources at a given point in time is called the *virtual swarm* which is the execution context of the application. In particular, a virtual swarm is a time-varying mapping of actions on virtualized resources to physical resources as shown in Figure 5.1.

As an improvement, actions can be merged into one in order to reduce the overall system utilization. Two actions can be merged if

- They request a compatible capability.
- The intersection set of the respective constraints is not the empty set. For this two intersection sets have to be created: a temporal and a spatial one.

In the following, two actions a and b are considered with the following specifications:

- cap_a *{TempSensor: Resolution: 0.1}*, $t_a \in [0, 10] \wedge l_a \in [(0, 0), (10, 10)]$.
- cap_b *{TempSensor: Resolution: 0.5}*, $t_b \in [2, 7] \wedge l_b \in [(2, 2), (8, 8)]$.

Both actions request the same capability (cap_a and cap_b), a temperature sensor, but with different configurations (different resolutions). The timing constraints of the actions are given by t_a and t_b. The spatial constraints are given by l_a and l_b. Before two actions are merged, the following rules have to be checked:

$$(\exists cap^* | cap^* \subseteq cap_a \wedge cap^* \subseteq cap_b) \tag{5.1}$$

$$(\exists t^* | t^* \subseteq t_a \wedge t^* \subseteq t_b) \tag{5.2}$$

$$(\exists l^* | l^* \subseteq l_a \wedge l^* \subseteq l_b) \tag{5.3}$$

First, the compatibility of capabilities has to checked (Rule 5.1). A compatible capability is, hence, cap^* = *{TemperatureSensor: Resolution: 0.1}* since this capability has a more fine-grained resolution which also satisfies the needs of action b. Next, Rule 5.2 checks the temporal constraints: two actions can be merged if the intersection set is not the empty set. The intersection is: $t^\cap = t_a \cap t_b = [0, 10] \cap [2, 7] = [2, 7]$. Finally, Rule 5.3 checks the spatial constraints by creating the intersection set: $l^\cap = l_a \cap l_b = [(0, 0), (10, 10)] \cap [(2, 2), (8, 8)] = [(2, 2), (8, 8)]$.

Since cap^*, $t^* \in t^\cap$ and $l^* \in l^\cap$ exist, the two actions a and b can be merged into a new action c: $cap_c = cap^*$, $t_c = t^\cap$ and $l_c = l^\cap$.

Depending on the constraints, it is possible that a merged action c is not schedulable but the original actions a and b are schedulable. This is due to shrinking the solution

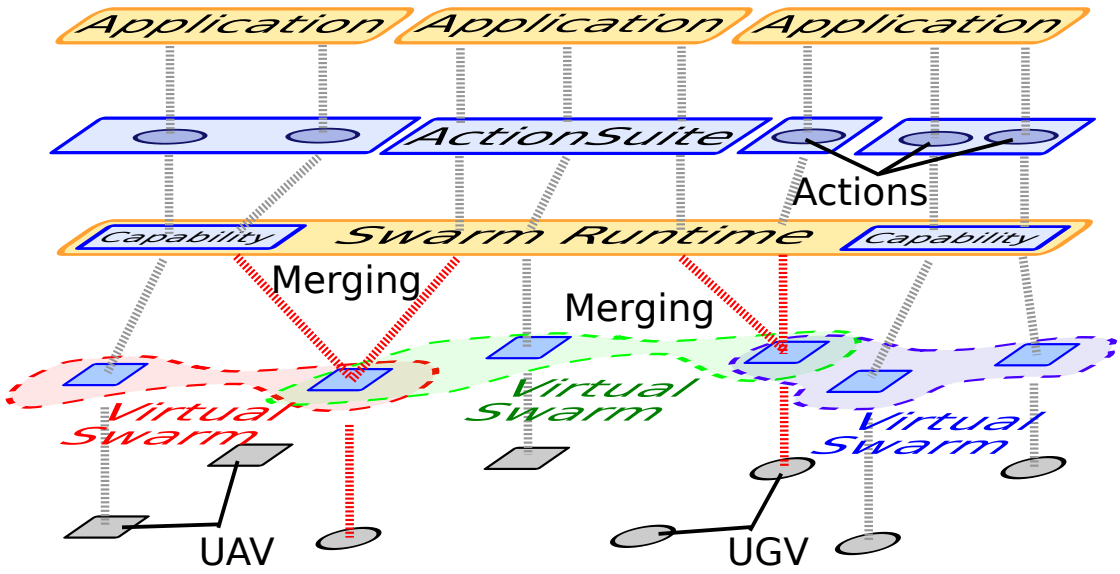


Figure 5.1: Action management and swarm virtualization.

space. In that case, the merged action c is discarded and the original actions a and b are "restored" and scheduled separately.

The example depicted in Figure 5.1 shows three applications creating six actions in total that are distributed across four suites. Since each application has its own virtual swarm, there are also three virtual swarms depicted. Two pairs of actions could be merged into one, respectively, resulting in a total of five actions that are mapped to physical resources.

5.3 Architecture

Figure 5.2 shows the system architecture which consists of system services (used for global system management) and node services (used for local node management). Services provide necessary functionalities in order to operate the system. They are loosely coupled and are kept modular, enabling interchangeability. In Section 5.3.1 the local system services and in Section 5.3.2 the global system services are explained.

5.3.1 Local System Services

The set of local system services are executed on every node that is participating in the (distributed) swarm system. The following explains the individual services in detail.

NodeManager

The node manager is the central management service on each node and has several responsibilities. If a new application shall be executed, it is first sent to the *LoadBalancer* which

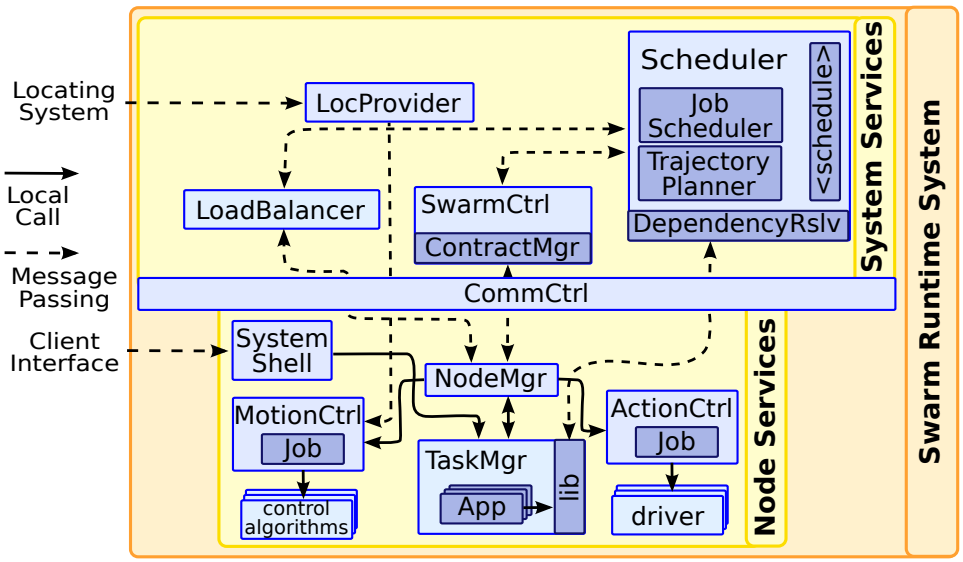


Figure 5.2: Architecture of the swarm runtime system.

decides on which node the application is started. According to the result, the respective node manager (on the node chosen by the load balancer) informs the *TaskManager* which starts the application. As the central management service, the node manager is responsible that the schedule for this node is kept. Therefore, it may temporarily interrupt a running application in case the system load is high.

If new jobs have been scheduled by the *Scheduler*, they are transferred via the *SwarmCtrl*. There are two possible types of jobs: *movement jobs* and *action jobs*. The former one is forwarded to the *MotionCtrl* while the latter one is forwarded to the *ActionCtrl*. In particular, the node manager owns a local run-queue and acts as a dispatcher: all jobs in the queue are ordered according to its attached timestamp which states at which point in time the respective action shall be executed. The first element is popped from the queue (depending on the timestamp) and handed over to the respective service which then starts to execute it. Due to the time-based execution semantic, the system idles in the period between the current point in time and the point in time when the next job has to be executed.

SystemShell

The system has a command line interpreter in order to interact with the system, obtaining system states and starting new applications.

ActionControl

The action control is responsible for driver management. The corresponding lifecycle has been explained in Section 4.5.3. The node manager instructs the action control, if a new action shall be executed. The action itself contains information about which system capability shall be used. The action control dispatches to the respective *driver* which executes it. If return values are provided, then the action control triggers the node manager to send the data to a certain target, as specified by the program logic (see Section 4.6).

MotionControl

The motion control is responsible for movement. Movement is defined here as the change in position of the current node. The local nodes give up their autonomy and are controlled by the system instead. Arbitrary movement is, thus, not allowed. Two things are necessary: first, the node has to know the exact route (where to go) and, second, it needs to know its current location.

The motion control becomes active when a new job arrives from the node manager. The job contains a spatio-temporal trajectory which is a set of space-time points (x, y, t). The motion control has access to multiple control algorithms. In order to follow the trajectory, the elected algorithm is responsible for fine-grained control of the actuation system of the node, e.g., for some UGVs, the actuation system is comprised of wheels. For each device class a suitable control algorithm has to be available in order to steer the node. The different velocities, necessary to follow the trajectory, are implicitly derived, based on the space-time points.

There are different options in order to obtain the current location: based on odometry, context information such as location and heading is estimated based on a starting location using sensors, e.g., wheel rotation for wheel-based robots. GPS can be used complementary in order to reach a higher precision. However, the architecture presented in this thesis shows the use of an external locating system that computes the location and heading of the robots. The locating system is explained in the appendix (Chapter B).

TaskManager

The task manager monitors and manages all local services. If a new application shall be started (as instructed by the node manager), it loads the respective code, e.g., from disk and starts the local execution by jumping to the application's entry point. The execution takes place in a different thread of control that is managed by the task manager.

CommCtrl

As a central element for communication, the communication control establishes the connections between the node and system services. In Figure 5.2, solid arrows indicate regular (local) method invocations. All local services simply use method invocation while system services are distributed across the network and, therefore, communication

with them requires message passing (dashed arrows). Each node has one communication control that handles all necessary data exchange.

5.3.2 Global System Services

The global system services are distributed across the network. They communicate using message passing. The following explains the global services in detail.

LoadBalancer

The load balancer is responsible for distributing the load across the nodes. In particular, if a new application is started, the load balancer assigns the application to the node with the lowest future load. The calculation of the load (node utilization) is explained in Section 5.4. Currently, the load balancer decides based on the absolute utilization u; with u_r being the utilization of robot r. If a new application is submitted for execution, the load balancer determines the node r^* which has the minimum utilization u_{r^*}: $u_{r^*} := min(u_{r_1}, u_{r_2}, ..u_{r_n})$.

Movement and action jobs are node-specific and can not be assigned to other nodes since they have been scheduled to dedicated nodes based on the attached spatio-temporal constraints. Depending on application behavior, there might be some nodes involved in heavy computation while others might be more in idle state. This situation occurs, e.g., when required resources are only available on particular nodes. Hence, the load balancer obtains its importance since it redirects applications to nodes with fewer (future) load. Applications themselves have no execution restriction according to space and time; this is only valid for the contained actions.

Space-Time Scheduler

The scheduler is the core-service and is responsible for scheduling all actions together with their spatio-temporal constraints in space and time. In particular, a call to `schedule()` triggers the scheduler. The scheduler receives an action suite with actions and constraints and tries to find a mapping of *all* contained actions to nodes under consideration of the constraints in order to create a contract (Section 4.6.3). The scheduler is comprised of three components, the *DependencyRslv*, the *JobScheduler* and the *PathPlanner*.

DependencyRslv The dependency resolver is the first stage that is addressed when a schedule operation is triggered. As described in Section 4.7, each action suite contains a dependency graph. The responsibility of the dependency resolver is to produce a topological sorting on the elements of the dependency graph. This determines the order in which the actions are scheduled.

JobScheduler The job scheduler is the second stage. According to the topological sorting, the elements are successively transferred to the job scheduler. Each time the job scheduler is invoked, it evaluates the action together with its constraints (including the

required capability) and uses a heuristic in order to pick a robot and proposes it as a candidate to the trajectory planner.

TrajectoryPlanner The trajectory planner is the final stage which checks schedulability of the action on the proposed robot candidate by calculating a spatio-temporal trajectory. It must be assured that the spatio-temporal trajectory is collision-free from static as well as dynamic obstacles. Furthermore, the trajectory must have been calculated such that the robot is able to reach the final destination and execute the action without violating any of the attached constraints. Overall schedulability of an entire action suite is checked if all individual actions of that suite are schedulable.

SwarmControl

The swarm control is the mediator between the individual nodes and the system scheduler. In particular, it obtains scheduled jobs from the scheduler. The jobs are enqueued in a local run-queue and sent to the respective nodes.

As described in Section 4.6.4, after the scheduler has successfully checked schedulability, executability has to be checked as well before a new contract is created. The responsibility of the *ContractMgr* is, hence, to check executability and, if successful, a new contract is created. As a result the respective application is notified. The protocol which checks executability is introduced in Section 5.5.1. The period in which executability has to be checked is also called *uncertainty period* since it is not clear during this time if a new contract is created or not. Avoiding inconsistencies between parts of the system, the scheduler internally uses the concept of *alternatives* which temporarily includes an alternative schedule: one schedule reflects the situation in which the new contract is created and the other schedule reflects the situation in which the contract is not created. In both cases—either executability is granted or not—the scheduler has to be notified in order to commit one of the schedule alternatives and discard the other one.

The responsibility of notifying the scheduler to either commit or discard a schedule lies in the swarm control.

LocationProvider

The location provider is a simple component that provides context parameters (physical coordinates, heading, robot identification number) of all nodes that are part of the presented swarm system in this thesis. It obtains such parameters by an external locating system that is explained in the appendix (Chapter B). It delivers the values to the respective nodes. Those values are requires by the motion control. Using decentralized locating, e.g., GPS, would even simplify the architecture since each node would know its current location.

5.4 System Utilization

At each point in time, a robot is in one of the following states exclusively[1]:

- *execution mode:* a robot is currently involved in the execution of a job.
- *movement mode:* a robot is currently performing movement, i.e., the robot is in a movement process in order to execute a job at a particular position.
- *idle mode:* in idle mode the robot is neither in execution nor in movement mode and, thus, "idles" at its current position.

The system has a utilization $u(t_1, t_2)$ with $[t_1, t_2]$ being the time interval in which u is calculated; $t_2 > t_1$. Each utilization is normalized: $u \in [0, 1]$. The length of the time interval is defined by $\Delta_t := t_2 - t_1$. The system utilization u is defined as a function of the utilizations of the individual nodes:

$$u(t_1, t_2) := \frac{u_1(t_1, t_2) + u_2(t_1, t_2) + .. + u_n(t_1, t_2)}{n} \tag{5.4}$$

with u_r being the local utilization of robot r. There are different utilization functions that express different utilization types:

5.4.1 Job Utilization

The job utilization $u_r^j(t_1, t_2)$ defines the utilization of robot r based on *action jobs* in the time interval $[t_1, t_2]$. In particular, it expresses the fraction in which the robot is occupied based on action jobs that have to be executed. The utilization is given by

$$u_r^j(t_1, t_2) := \frac{\sum_{j \in J_r \mid j \in [t_1, t_2]} dur(j)}{\Delta_t} \tag{5.5}$$

For the calculation all jobs $j \in J_r$ of robot r are considered that will be executed in the time interval $[t_1, t_2]$. The function $dur(j)$ delivers the duration of job j. If a job lies partially in the interval, then $dur(j)$ delivers only the fraction that is inside $[t_1, t_2]$.

5.4.2 Motion Utilization

The motion utilization $u_r^m(t_1, t_2)$ defines the utilization of robot r based on *movement jobs* in the time interval $[t_1, t_2]$. In particular, it expresses the fraction in which the robot is occupied based on movement. The utilization is given by

$$u_r^m(t_1, t_2) := \frac{\sum_{m \in M_r \mid v(m) > 0} dur(m)}{\Delta_t} \tag{5.6}$$

In order to calculate $u_r^m(t_1, t_2)$ all movement jobs of robot r are considered that will be executed in the time interval $[t_1, t_2]$. A movement job $m \in M_r$ contains a spatio-temporal

[1] This assumption is made due to hardware-specific reasons (CPU with single core) of the testbed.

trajectory which may consist of multiple segments $(m_1, m_2, m_3, ..)$[2]. Each segment may have different velocities v: $v(m_i)$. The function $dur(m)$ delivers the time that is required in order to perform the movement. If m contains segments, then the duration is computed for each segment: $dur(m) = dur(m_1) + dur(m_2) + .. + dur(m_n)$. The duration is only considered if $v > 0$:

$$dur(m) = \begin{cases} 0, & \text{if } v(m) = 0 \\ dur(m), & \text{otherwise} \end{cases} \tag{5.7}$$

5.4.3 Relative Motion Utilization

The relative motion utilization $u_r^{m^*}(t_1, t_2)$ defines the utilization of robot r based on *movement jobs* in the time interval $[t_1, t_2]$ while considering the actual velocity. The motion utilization, $u_r^m(t_1, t_2)$, in contrast, treats the velocity as a binary function—$v = 0$ or $v > 0$. Only segments which state a velocity $v > 0$ are considered. The relative motion utilization $u_r^{m^*}(t_1, t_2)$ considers all velocities and expresses to which extent the robot's capacity is used[3]. The relative motion utilization is given by

$$u_r^{m^*}(t_1, t_2) := \frac{len(m)}{\Delta_t \cdot v_{max}} \tag{5.8}$$

The function $len(m)$ delivers the total length of the trajectory of m (physical length). The denominator, $\Delta_t \cdot v_{max}$ expresses the maximum physical path length that the robot is able to move during the time interval given by $[t_1, t_2]$ with $\Delta_t = |t_2 - t_1|$. The value of $u_r^{m^*}$ indicates the ration between the effective covered distance and the maximum theoretic distance.

5.4.4 Utilization

The utilization $u_r(t_1, t_2)$ defines the combined utilization of both $u_r^j(t_1, t_2)$ and $u_r^m(t_1, t_2)$ with respect to robot r. The utilization is given by

$$u_r(t_1, t_2) := u_r^j(t_1, t_2) + u_r^m(t_1, t_2) \tag{5.9}$$

5.4.5 Relative Utilization

The (relative) utilization $u_r^*(t_1, t_2)$ defines the combined utilization of both $u_r^j(t_1, t_2)$ and $u_r^{m^*}(t_1, t_2)$ with respect to robot r. The utilization is given by

[2]This section describes utilizations based on job-level. A movement job m contains a spatio-temporal trajectory whose concrete computation is not addressed here, but given in a later chapter.

[3]Future work might address to reschedule nodes with small utilizations $u_r^{m^*}(t_1, t_2)$ (low velocities) in order to schedule more action jobs if the motion utilization $u_r^m(t_1, t_2)$ itself is high. The relative motion utilization plays an important role when energy-aware scheduling is considered. In this case, a trade-off between energy consumption and high velocities has to be found. Energy-aware scheduling is not considered in this thesis and is, hence, declared as future work.

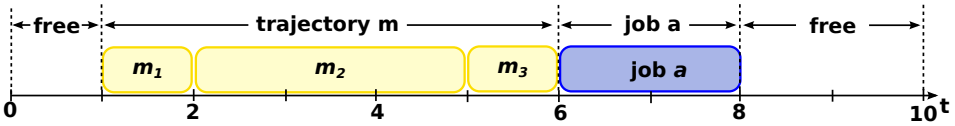


Figure 5.3: Schedule of robot r.

$$u_r^*(t_1, t_2) := u_r^j(t_1, t_2) + u_r^{m^*}(t_1, t_2) \tag{5.10}$$

5.4.6 Idle Time

The idle utilization $u_r^i(t_1, t_2)$ defines the fraction in which the robot idles and, thus, is free. Idling indicates that the robot is neither executing a job nor performing movement. The idling utilization is given by

$$u_r^i(t_1, t_2) := 1 - u_r(t_1, t_2) \tag{5.11}$$

5.4.7 Relative Idle Time

The relative idle utilization $u_r^{i^*}(t_1, t_2)$ indicates the fraction of available capacities. In relative idle time a motion that is performed with a velocity $v < v_{max}$ is interpreted as a special case of idle time since the robot has more capacity than actually used. If $u_r^i(t_1, t_2) > u_r^{i^*}(t_1, t_2)$, this indicates that the robot does not constantly move with v_{max}. In general, $u_r^i(t_1, t_2) \geq u_r^{i^*}(t_1, t_2)$ holds. The relative idling utilization is given by

$$u_r^{i^*}(t_1, t_2) := 1 - u_r^*(t_1, t_2) \tag{5.12}$$

5.4.8 Example

Figure 5.3 shows an example schedule of robot r. The schedule contains one trajectory m and one job a. The trajectory consists of three segments (m_1, m_2, m_3). The maximum velocity is given by $v_{max} = 2$. Table 5.3 shows the different utilizations as a function of the time interval given by $[t_1, t_2]$ (Table 5.1 shows the parameter setup and Table 5.2 shows auxiliary functions). Since there is only one robot the system utilization u is equal to the (relative) utilization of robot r. The scenario is set up as follows (Table 5.1): The first trajectory segment m_1 has a physical length of 1 ($len(m_1)$). The movement along m_1 takes 1 time unit ($dur(m_1)$) while moving with a constant velocity of 1 ($v(m_1)$). The second segment m_2 has a physical length of 0 ($len(m_2)$), i.e., r is not moving ($v(m_2) = 0$). After a duration of 3 time units ($dur(m_2)$), the robot continues moving along the third segment m_3 for 1 time unit ($dur(m_3)$) with a velocity of 2 ($v(m_3)$). Trajectory segments with a velocity of 0 may appear due to several reasons, e.g., the current path is blocked for a given amount of time and, hence, the robot has to wait until the path becomes free again.

	m_1	m_2	m_3
$len(..)$	1	0	2
$dur(..)$	1	3	1
$v(..)$	1	0	2

Table 5.1: Parameter set-up.

	$dur(a)$	Δ_t	$dur(m)$	$len(m)$
$[0, 10]$	2	10	2	3
$[3, 7]$	1	4	1	2

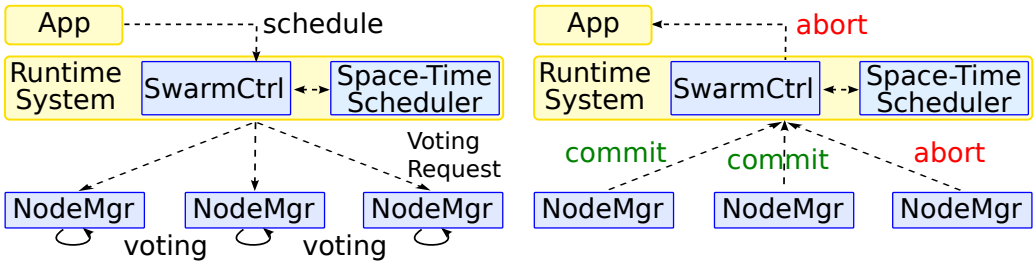
Table 5.2: Auxiliary functions.

	u_r^j	u_r^m	$u_r^{m^*}$	u_r	u_r^*	u_r^i	$u_r^{i^*}$	u
$[0, 10]$	0.2	0.2	0.15	0.4	0.35	0.6	0.65	0.4 (0.35)
$[3, 7]$	0.25	0.25	0.25	0.5	0.5	0.5	0.5	0.5 (0.5)

Table 5.3: Utilizations of robot r and system utilization u.

5.5 System Operation

Using the system shell (Section 5.3.1), new applications can be started and system states as well as utilizations introduced in Section 5.4 can be monitored. If an application is submitted for execution, the load balancer determines a suitable node and the node manager starts initiating the execution by jumping to its entry point as described in Section 5.3.1. Once a new application is started, its lifecycle is initiated and traversed as introduced in Section 4.6.8. If the execution of the application's code reaches an instantiation of an action suite, then its lifecycle is initiated as explained in Section 4.6.5. An invocation of one of the `schedule()` operations (Section 4.6.2) of an action suite results in serializing and sending the associated actions together with the dependency graph (Section 4.7) to the scheduler by message passing. This initiates the contract creation lifecycle as described in Section 4.6.4. The scheduler schedules the actions under consideration of the spatio-temporal constraints and the dependency graph in space and time as explained in Chapter 7. Once the scheduler has produced a schedule for the action suite (*sched()* holds), the result is handed over to the swarm control in order to communicate and assign the generated individual jobs to the respective nodes. Although the scheduler has already calculated and assigned actions to nodes, this has still to be communicated over the network (via message passing) to the nodes which inevitably consumes time. If real-time communication is possible, firm statements about the worst case message delay can be assumed. This upper time bound for communication can simply be incorporated in the entire workflow (ranging from the scheduling to finally inform the nodes). However, in a scenario in which mobile robots communicate using wireless multihop networks, it is hard to guarantee hard real-time communication. Therefore, an upper bound for the message delay can not be assumed. Since all actions in the action suite shall be executed



(a) Voting phase. (b) Commit phase.

Figure 5.4: Variant of two-phase commit protocol.

(according to the spatio-temporal constraints), executability (*exec()*) has to be checked in addition which requires to incorporate a distributed voting phase, i.e., all nodes involved in the execution of an action suite are informed and perform a voting.

5.5.1 Variant of Two-Phase Commit Protocol

In order to decide executability, a variant of the *two-phase commit protocol* (2PC) [37] is applied as depicted in Figure 5.4. The swarm control sends all jobs scheduled by the scheduler to the involved nodes and requests the nodes to perform a local voting (Figure 5.4(a)). This starts the voting phase. A node must vote to *commit* if the execution of the job is possible. In the case that the node is not able to execute the job (due to insufficient resources or too late arrival of messages), it is allowed to reject the job by voting with *abort*.

The variant of the two-phase commit protocol operates according to the original 2PC in two stages: first, in the voting phase, the nodes perform a voting. Only if all nodes vote to commit, the transaction is globally committed. If at least one node has voted to abort, the entire transaction is aborted. Once a node has voted for commit or abort it has to stay with its vote. After all votings from the nodes have been collected, the coordinator (swarm control) makes a decision. In the second stage, the coordinator tells all participating nodes if the global transaction should be committed or aborted. If at least one abort is received, the entire transaction is aborted.

The period between the local vote and the final decision is called *uncertainty period*. During that time a node can not unilateral commit or abort since it does not know the global decision. Timeouts are used in the second phase if a commit / abort message from the coordinator got lost which triggers a retransmit. This works for transient errors. The disadvantage of the two-phase commit protocol is that permanent errors, i.e., the message never arrives, requires user interaction. There is no default value that could simply be adopted. There are variants of that protocol such as the *three-phase commit protocol* (3PC) [92] and the *enhanced three-phase commit protocol* (E3PC) [41]. They improve the original 2PC protocol to a certain extent. Both variants create quorums in case of failures in order to make further progress. While the 3PC can only make progress in case of one failure, the E3PC proposes that a quorum always makes progress. However, since

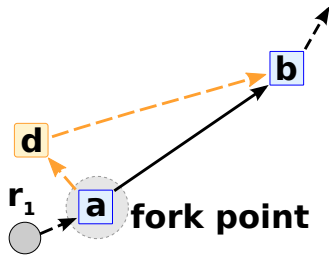


Figure 5.5: Spatio-temporal fork point.

the work of this thesis is based on space and time, the following situation could occur:

A robot r_1 has already two confirmed scheduled jobs (a, b) as depicted in Figure 5.5. Its current schedule is $s_1 = \{a, b\}$ which states that r_1 first has to move to a and afterwards to b. A new action d shall be scheduled. The scheduler has computed a new schedule $s_1^* = \{a, d, b\}$. The predicate *sched(d)* is true. The new job d is scheduled between both other actions. The resulting spatio-temporal trajectories are $(a \rightarrow d \rightarrow b)$. After *sched(d)* holds, *exec(d)* has to be verified as well before the action is finally committed for execution. Therefore, the new schedule s_1^*, which shall replace s_1, is sent to r_1 initiating the voting phase. The message that carries s_1^* is defined as $m_{s_1^*}$. Depending on the arrival time $t^{arr}(m_{s_1^*})$ of $m_{s_1^*}$ at r_1 and the time at which r_1 has arrived at the fork point t_{fp}, the following situations can occur:

- $t^{arr}(m_{s_1^*}) > t_{fp}$: In this case, the message arrives too late, i.e., r_1 had no information of the new job d as it left the "fork point". In this scenario, the fork point is not really a fork point since r_1 was not aware that it was a fork point. Since the entire system is strictly space and time based (this applies also to the trajectory $(a \rightarrow b)$), *sched(d)* is true, but *exec(d)* is false (the robot cannot go back in time). In this case, the robot has to reply to the voting request with an abort message.

- $t^{arr}(m_{s_1^*}) <= t_{fp}$: In this case, the message arrives either in time or ahead of time concerning the fork point. The robot has a chance to take the detour $(a \rightarrow d \rightarrow b)$ and execute d. In this case, r_1 votes for commit.

Using the classic 2PC, the uncertainty period starts right after the robot has voted and ends at the point in time when the global commit or abort message, which has been sent by the coordinator, is received. In the following this interval is given by $[t_1^{up}, t_2^{up}]$. During that time participants can not unilateral commit or abort since they do not know the global decision of the coordinator. Assuming the case that the message arrived not after the fork point: $t^{arr}(m_{s_1^*}) <= t_{fp}$. The point in time at which the global commit / abort message arrives is $t^{arr}(m_{c/a})$. The following order on the timestamps of the events hold: $t^{arr}(m_{s_1^*}) < t_1^{up} < t^{arr}(m_{c/a}) < t_2^{up}$. The order indicates that first the voting request is received which initiates the uncertainty period. After the global decision message is received, the uncertainty period is terminated. Using the classic 2PC the following situations can occur:

- $t^{arr}(m_{c/a}) <= t_{fp}$: In the case, the message containing the global decision arrives ahead of time or in time, i.e., before or at the point in time when r_1 has reached the fork point, then r_1 moves to d in case the global decision is a commit or moves to b if the global decision is an abort.

- $t^{arr}(m_{c/a}) > t_{fp}$: In this case, r_1 reaches the fork point during the uncertainty period. Since the entire system is based on space and time, the robot has to make a decision in which direction it will continue to move. If the robot has voted for abort, then the route over d is not possible either way, independent of the global decision message. In case r_1 has voted for commit, the global decision might be also a commit or an abort. This situation is not determinable.

Therefore, the 2PC protocol is modified as follows: After a node has voted for commit or abort, it has to stick to its vote, i.e., if r_1 has decided to commit, it will move along the trajectory $a \rightarrow d \rightarrow b$ and will execute d. Otherwise it moves directly to b (abort vote). Only in case that all participating robots have commonly voted for commit the designated application that has issued the scheduling is notified accordingly (5.4(b)). This has the advantage that the system is completely autonomous; no user interaction is required. If one node votes for abort and the others vote for commit, then swarm control triggers an `unschedule()` operation immediately in order to discard the respective actions and correct the trajectory respectively. The unschedule operation only involves one action, e.g., if x nodes have voted for commit, then x individual unschedule operations for the nodes will be performed. If an unschedule operation is triggered while the robot is already on its way, the trajectory cannot be changed. It is still possible that simply the job is not executed. However, from the application's point of view (in user space) the action suite has an all-or-nothing semantic since either all values are provided or none of them. Applying the simple cost model that has been introduced in Section 4.6.3, as incentive an application only has to "pay" for resources if a contract is actually applied, i.e., *sched()* and *exec()* are both true and no re- or unscheduling has been performed.

If event handlers are installed (Section 4.6.6), the verification of *sched()* and *exec()* together with the respective results are provided for the application in order to react to the state changes.

5.5.2 Control Flow

This section describes the control and message flow of scheduling a new action suite including the role and intention of the involved components. Figure 5.6 shows the order and the amount of messages that have to be sent in order to (successfully) schedule the action suite. Each node, that is involved in the entire process, is marked with a different color in Figures 5.6 and 5.7.

As explained in Chapter 4, the *library* (*lib*) is used in order to create new actions. Figure 5.6 presents the execution of an application. Using the *lib*, a new action is created and the *schedule* operation is invoked. In order to communicate with the scheduler, a dedicated component (*SchedulerProxy*) is used that creates the respective messages and

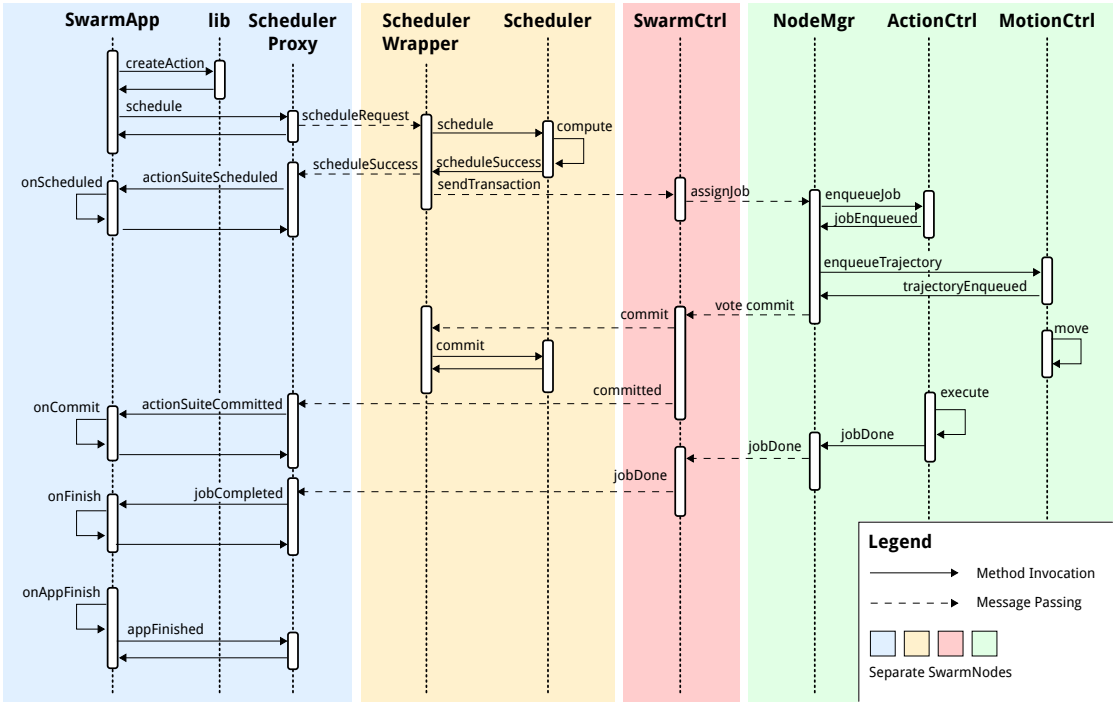


Figure 5.6: Successfully scheduling of an ActionSuite.

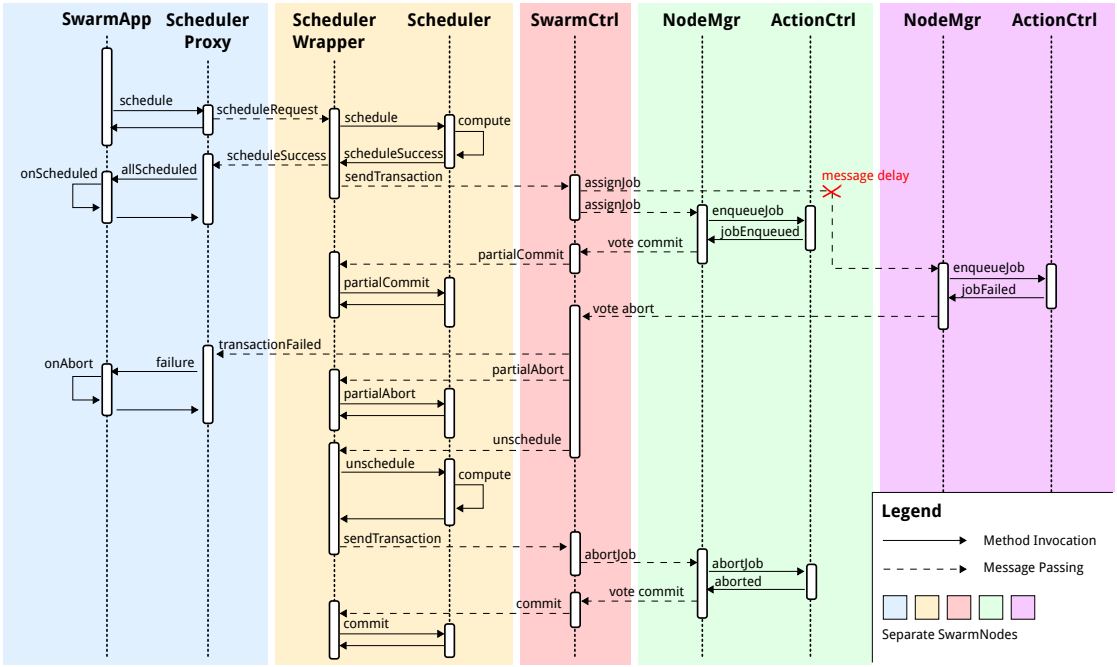


Figure 5.7: Failure during scheduling of an ActionSuite.

uses the *CommCtrl* for sending the messages. The *schedule* operation returns after the message has been sent producing asynchrony. Each node uses the *CommCtrl* for sending and receiving messages. The component has been omitted for clarity. The counterpart for the *SchedulerProxy* is the *SchedulerWrapper* which is a wrapper component around the *Space-Time Scheduler*. The component receives messages and invokes the respective operations on the scheduler. In this case, it invokes the *schedule* operation with the associated action suite. After the schedule has been computed, the *SchedulerWrapper* obtains the result and sends a message (containing the result) to the *SwarmCtrl*. At the same time, it sends a second message back to *SchedulerProxy* which informs the application that the suite is marked as scheduled (invoking the `onScheduled` listener).

The computed schedule is a (distributed) transaction. The *SwarmCtrl* sends each node that is involved in the transaction a message which contains the node-specific information of the schedule and starts the variant of the two-phase commit protocol (Section 5.5.1). Once a *NodeMgr* has received its specific part of the schedule, it delivers the contained jobs to the *ActionCtrl* and *MotionCtrl*, initiating the local voting.

In Figure 5.6, a *commit* vote is assumed. Hence, the *NodeMgr* sends a *commit* message back to the *SwarmCtrl* which performs some local accounting (according to the variant of the two-phase commit protocol). The message is forwarded to the *SchedulerWrapper* which invokes the respective operation, i.e., a commit here. If all nodes have voted for commit (according to the variant of the two-phase commit protocol), then the *SwarmCtrl* sends a message back to the *SchedulerProxy* in order to notify that the transaction has been committed for execution which leads to an invocation of the `onCommit` listener.

Concurrent to the described control and message flow, at some point in time (according to the calculated schedule), the *MotionCtrl* and the *ActionCtrl* become active and execute their associated jobs. In case of success, the *NodeMgr* produces a last notification message and sends it to the *SwarmCtrl* which performs some local accounting again. The message is forwarded to the *SchedulerProxy* which invokes the `onFinish` listener. In case, all action suites have been executed, the final listener `onAppFinish` is invoked.

In Figure 5.7, a scenario is presented that shows an error during the control and message flow. The schedule (computed by the *Scheduler*) involves two nodes. The *SwarmCtrl* generates two messages accordingly and sends them to the respective nodes.

One of the messages arrives in time at the node and is enqueued locally. A *commit* message is sent back to the *SwarmCtrl* which leads to a partial commit. However, the other message for the second node is delayed. Hence, the message arrives too late such that the movement or action job can not be enqueued (Section 5.5.1). As a consequence, the *NodeMgr* has to vote with *abort*. This leads to a global abort of the transaction, i.e., the application is notified and `onAbort` is invoked. The *SwarmCtrl* has to instruct the *SchedulerWrapper* in order to perform two operations: first, a partial abort has to be performed and, second, an unschedule operation has to be performed. This opens a second transaction. Hence, the *SchedulerWrapper* informs the *SwarmCtrl* which informs the respective *NodeMgr* in order to unschedule the previously committed job. After this transaction is committed by discarding the the job, the first transaction is “rolled back”.

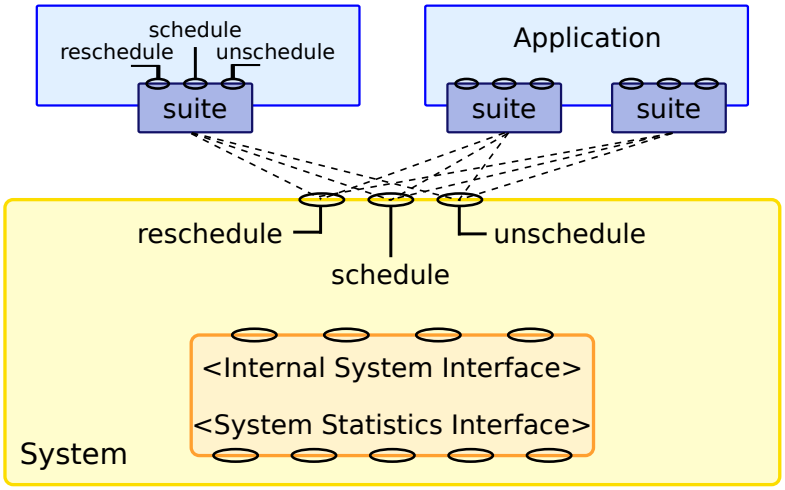


Figure 5.8: System interface.

5.6 System Interface

The system provides different types of interface operations used by different units of the system. Figure 5.8 shows the arrangement of interface operations.

5.6.1 External System Interface

The external interface operations are designed for applications. These operations are accessible by invoking the correspondent interface operations of the action suite as described in Section 4.6.2 (page 44).

- `schedule()`: Schedules all actions of an ActionSuite in space and time under consideration of the spatio-temporal constraints and inter-action dependencies.
- `reschedule()`: Reschedules an existing and already scheduled ActionSuite. This allows modification of the spatio-temporal constraints of the actions.
- `unschedule()`: If already scheduled actions of an ActionSuite have become obsolete for some reason, unschedule removes them from the global schedule and frees system resources.

5.6.2 Internal System Interface

As described in Section 5.5 (page 71), the external system interface operations trigger to check the predicate *sched()*. Afterwards, *exec()* has to be verified as well. Depending on the result, the system automatically invokes one of the following operations and commits or aborts an open transaction:

- `commit(tid)`: Commits the open transaction *tid*.
- `commit(tid, r)`: Partially commits the transaction *tid* for robot r.
- `abort(tid)`: Aborts the open transaction *tid*.
- `abort(tid, r)`: Partially aborts the transaction *tid* for robot r.

5.6.3 System Statistics Interface

The system statistics interface allows access to obtain the system load information which has been defined in Section 5.4. This information is required by the load balancer.

- `calcJobLoad(r, from, to)`: Calculates the job load of robot r in the interval [*from, to*].
- `calcMotionLoad(r, from, to)`: Calculates the motion load of robot r in the interval [*from, to*].
- `calcLoad(r, from, to)`: Calculates the load of robot r in the interval [*from, to*].
- `calcIdle(r, from, to)`: Calculates the idle fraction of robot r in the interval [*from, to*].
- `calcRelativeMotionLoad(r, from, to)`: Calculates the relative motion load of robot r in the interval [*from, to*].

5.7 Conclusion

Executing (distributed) applications according to the programming model (Chapter 4), requires a suitable distributed infrastructure. This chapter has presented an architecture for a distributed swarm operating system which consists of (local) node services and (global) system services.

The former ones are executed on each node that is participating in the distributed swarm system. The services are required for operating the local node. The node manager is the central local management instance. It delegates incoming jobs (action jobs and movement jobs) to the respective services: action and motion control.

The latter ones are used in order to operate the system, i.e., resource allocation and management which is performed by the core-service: the space-time scheduler. In order to assure consistency in a distributed system, a variant of the two-phase commit protocol has been introduced which is implemented in the swarm control service. Combining features from both services allow to provide advance resource reservation. This is a feature of the programming model: application programmers are able to use this mechanism in order to create contracts with the system. This assures guaranteed spatio-temporal resource allocation.

A utilization refers to as a measure used in order to derive the degree of free capacities. In this chapter, metrics have been introduced which define the parameters necessary in order to calculate the system's utilization. The system utilization is comprised of the utilization of the local nodes. Each (node) utilization is influenced by the assignment of action jobs and movement jobs. A relative utilization function is considered in addition which states the relation between average velocity and maximum (feasible) velocity. The utilization measure is used by the load balancer in order to distribute the load among the nodes.

Chapter 6

Group-Scheduling Problems (Offline)

Given a set of applications that spawn spatio-temporal actions, a mapping of actions to execution units of the runtime system is required. This chapter addresses the problem of spatio-temporal group scheduling. The technique presented in this chapter is suitable for offline scheduling problems. The scheduling problem is modeled using Timed Petri nets. Afterwards, an optimal schedule can be found by translating the modeled problem into a shortest path problem.

6.1 Introduction

The mapping of tasks (actions) to nodes remains one major objective. In this section, the problem of spatio-temporal group scheduling is addressed. Petri nets [73] are a well-known method to model dynamic systems with discrete states. They are well suited for modeling the concurrent behavior of distributed systems.

The remainder of this chapter is structured as follows: Section 6.2 shows related work in this field. In Section 6.3, assumptions of the world and the problem statement are presented. Section 6.4 introduces Timed Petri nets which are used in order to model the problem as described in Section 6.5. Section 6.6 presents a solution to the problem and shows how a schedule with minimal makespan is calculated by mapping the modeled problem to the shortest path problem. A case study including an evaluation of the approach is given in Section 6.7. Finally, Section 6.8 concludes this chapter.

6.2 Related Work

There are numerous Petri net publications with diverse fields of investigation [100]. The original Petri net does not include a notion of time. However, there are a number of extensions to Petri nets enabling them to deal with time. Probably the two most famous approaches are *Time Petri nets* [57] and *Timed Petri nets* [79] (TPN), also called

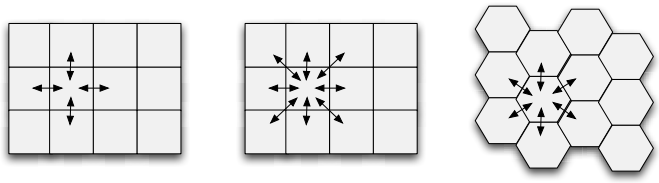


Figure 6.1: Examples for discrete topologies.

Duration Petri nets (DPN). Besides discrete Petri nets, e.g., TPN and DPN, there are also *Continuous Petri nets* [13] and *Hybrid Petri nets* [3] which are typically used in order to model a system which has a discrete part (e.g., Boolean state variable) and a continuous part (e.g., real number indicating liquid flow). Hybrid Petri nets including *Timed Hybrid Petri nets* are presented in [14]. Many applications of Petri nets target scheduling problems, e.g., deadlock avoidance, performance evaluation, or feasibility of real-time and non-real-time problems or workflow scheduling [106]. There is also work that uses dynamic programming [107] to solve an assembly line scheduling problem described by Petri nets as well as a shortest/longest path problem used to find optimal real-time schedules [74]. However, none of the publications in the Petri net literature deals with spatio-temporal group scheduling.

6.3 Assumptions and Model

A task γ is denoted as a tuple $(d, p, p', \rho, \Gamma')$. The parameters have the following intention: d indicates the duration of the task, p and p' are the beginning and ending locations[1] of the task, respectively. A task may also be bound to a fixed location—in that case p and p' are identical. Therefore, this model even allows movement and task execution at the same time[2]. A task can be executed jointly, i.e., multiple robots are required for the execution; $\rho <= |\Upsilon|$ denotes the number of required robots and $|\Upsilon|$ denotes the total number of robots. Tasks may have predecessors Γ'; all predecessors have to be executed prior to γ.

The physical space of the world is assumed to be 2D. The resulting surface is discretized and mapped to a specific topology as shown in Figure 6.1. Using space discretization forms cells. Each cell $c \in C$ in the topology indicates a space in which an arbitrary amount of robots can be placed. The amount a of robots per cell can change over time: $a(c,t) \in \{x \in \mathbb{N} \mid 0 \leq x \leq |\Upsilon|\}$ and for each t holds: $\sum_{c \in C} a(c,t) = |\Upsilon|$. The topology specifies the geometry of the surface, the discretization (cell shape) and the multiplicity of movements. A robot can change its location by moving in discrete steps to a neighboring cell along the indicated arrows. In analogy to the discrete space model, a simplified discrete time model is also used:

[1] Location defines a physical coordinate in real space.

[2] Due to hardware-specific reasons (CPU with single core) of the testbed, the assumptions in Section 5.4 are different than stated here.

time progresses in time steps. The movement model is based on a binary state: a robot moves or does not move.

At each time step, a robot has different options if it is not already involved in a movement process or a task execution: the robot stays locally in the cell (idle), it moves along one of the arrows to a neighboring cell or it starts a task execution which may involve movement additionally.

A set Γ of tasks is given. The goal is to find a spatio-temporal schedule with minimal makespan such that these tasks $\gamma_i \in \Gamma$ are executed according to their requirements.

6.4 Timed Petri Nets

This section introduces and describes the formalism of Timed Petri nets which is used in the remaining sections of this chapter. The formalism has been published in [29] and states a modification of the original definition which has been published in [75]. Timed Petri nets has been chosen since it allows to model timely behavior which is a necessity in space-time scheduling.

The following notation is used: $\mathbb{N}^+ = \mathbb{N}\backslash\{0\}$ is the set of natural numbers $\mathbb{N}$ without 0; $\mathbb{Q}$ is the set of rational numbers; $dom(f)$ and $codom(f)$ is the domain resp. codomain of a function f. $M(m,n)$ is a matrix with m rows and n columns. For a matrix M, $M^{(i)}$ denotes the i-th row of M, starting with 0. $M^{(,j)}$ denotes the j-th column of M, starting with 0 and $M^{(i,j)}$ denotes the i-th row and j-th column of M.

Definition 24 (Timed Petri net (TPN))
A TPN or DPN is a graph $N = (P, T, F, V, D)$, where

- *P, T, F are finite sets with $P \cap T = \emptyset$, $P \cup T \neq \emptyset$, $F \subseteq (P \times T) \cup (T \times P)$ and $dom(F) \cup cod(F) = P \cup T$, where the elements $p \in P$ are called* places*, the elements $\tau \in T$ are called* transitions*, and the elements of F are called* arcs*.*
- *$V : F \to \mathbb{N}^+$ is a weight of the arcs.*
- *$D : T \to \mathbb{N}^+$ is a* duration function*. $D(\tau)$ denotes the delay of transition τ.*

Definition 25 (State of a TPN) *The state of a TPN is a tuple $S = (m, h)$, where*

- *$m : P \to \mathbb{N}$ is called a* marking *of the net. The marking assigns to each place p a number of* tokens *denoted by $m(p)$.*
- *$h : T \to \mathbb{N}$ is a transition clock vector. The notation $h(\tau)$ describes the clock vector of the transition τ.*

A TPN has a dedicated state (m_0, h_0), called *initial state*. In an initial state, h_0 is always the zero vector. When the dynamics of a TPN is considered (see below), the initial state serves as a starting point of all considerations.

To any transition $\tau \in T$ belongs a pre-set $\bullet\tau$ and a post-set $\tau\bullet$, that are given as $\bullet\tau = \{p \mid p \in P \wedge (p, \tau) \in F\}$, and $\tau\bullet = \{p \mid p \in P \wedge (\tau, p) \in F\}$, respectively. Each transition $\tau \in T$ induces the markings τ^- and τ^+, defined as follows:

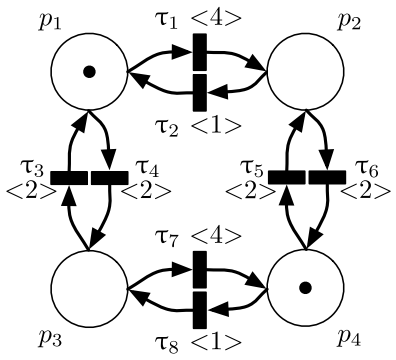


Figure 6.2: Example net, τ<t> denotes firing time t of τ.

$$\tau^- = (v_1, \ldots, v_{|P|})^T \mid v_i = \begin{cases} V(p_i, \tau) & \text{if } (p_i, \tau) \in F \\ 0 & \text{else} \end{cases}$$

$$\tau^+ = (v_1, \ldots, v_{|P|})^T \mid v_i = \begin{cases} V(\tau, p_i) & \text{if } (\tau, p_i) \in F \\ 0 & \text{else} \end{cases}$$

Then $\Delta\tau$ denotes the difference $\tau^+ - \tau^-$. A transition τ is *enabled* at marking m iff $\tau^- \leq m$ and $h(\tau) = 0$.

In order to introduce a state equation later, the definition of a marking is extended to a *time marking*:

Definition 26 (Time Marking) *A time marking $\mu \in M(|P|, d)$ is a matrix with $d = max\{D(\tau_i)|i \in \{1, .., |T|\}\}$ + 1, where each row represents a specific place p. Each column denotes a (partial) marking of a place for different time steps. The first column represents the present (at time t), i.e., equals the current marking. The second one denotes partial* changes *(i.e., future additions from time elapsing) in the net at $t + 1$ (*not *the marking at $t + 1$). The third column then represents partial changes at $t + 2$ and so on.*

There is a mapping $S \to \mu$ and a mapping $\mu \to m$. However, no mapping $\mu \to S$ exists.

Figure 6.2 shows an example net. It is assumed that the net is in an initial state. Then the time marking μ_0 is given by

$$\mu_0 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Next, TPN's dynamics are considered. A TPN can change its state by *firing* and by *time elapsing*. Elements that have been changed by firing are denoted with the symbol $\hat{}$ (e.g., $\hat{m}$) and elements that have been changed by time elapsing are denoted with the symbol $\tilde{}$ (e.g., $\tilde{m}$). Since a TPN is *enforced* to progress, the *maximal step* is introduced.

Definition 27 (Maximal Step) *$\mathcal{B} \subseteq T$ is called a maximal step at the marking m (resp. at time marking μ, since $m = \mu^{(,0)}$) with the transition clock vector h iff*

$$\sum_{\tau \in \mathcal{B}} \tau^- \leq m \tag{6.1}$$

$$\forall \tau, \tau \in \mathcal{B} \rightarrow h(\tau) = 0 \tag{6.2}$$

$$\neg \exists \mathcal{B}^* ((\mathcal{B} \subset \mathcal{B}^*) \wedge (\mathcal{B}^* \textit{ satisfies (6.1) and (6.2)})) \tag{6.3}$$

If at least one enabled transition exists, transitions of the TPN must *fire*. Only maximal steps fire in a TPN. If there are more than one maximal step that may fire, one of them is selected arbitrarily. The example net shown in Figure 6.2 with the initial marking m_0 has four maximal steps:

$$\begin{aligned} \mathcal{B}_1 &= \{\tau_1, \tau_5\} \\ \mathcal{B}_2 &= \{\tau_1, \tau_8\} \\ \mathcal{B}_3 &= \{\tau_4, \tau_5\} \\ \mathcal{B}_4 &= \{\tau_4, \tau_8\} \end{aligned}$$

Definition 28 (Firing) *A TPN with the time marking μ and with a maximal step $\mathcal{B}$ that becomes enabled at time t will change its state in the following way.*

$$\hat{m} = m - \sum_{\tau \in \mathcal{B}} \tau^- \tag{6.4}$$

$$\forall \tau \in \mathcal{B}, \hat{h}(\tau) = D(\tau) \tag{6.5}$$

$$\forall k \in \{1, \ldots, d\}, \hat{\mu}^{(,k)} = \mu^{(,k)} + \begin{cases} -\sum\limits_{\tau \in \mathcal{B}} \tau^- & \textit{iff } k = 0 \\ \sum\limits_{\tau \in \mathcal{B} | D(\tau) = k} \tau^+ & \textit{else} \end{cases} \tag{6.6}$$

The example net (Figure 6.2) has four possible maximal steps on μ_0. Let μ_1 be the time marking which the net reaches after firing $\mathcal{B}_2$, i.e., $\mu_0 \xrightarrow{\mathcal{B}_2} \hat{\mu}_1$. This leads to

$$\hat{\mu}_1 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \hat{m}_1 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \text{ and } \hat{h}_1 = \begin{pmatrix} 4 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}.$$

$\mathcal{B}_2$ indicates firing of τ_1, τ_8 which involves removing tokens from p_1 and p_4 at time t. Thus, the first column of $\hat{\mu}$ contains only zeros (tokens "reside" in the transitions). The duration of τ_8 is one time unit. Thus, at $t+1$ (second column) one token is released and put on p_3. The duration of τ_1 is four time units (longest firing duration of all T) and, therefore, at $t+4$ (fifth column) the remaining token appears on p_2.

For the elapsing of time, a specific matrix called *progress matrix* $\mathcal{Q}$ is used. The progress matrix $\mathcal{Q} \in M(d,d)$ is of the form

$$\mathcal{Q} = \begin{pmatrix} 1 & 0 & .. & 0 & 0 \\ 1 & 0 & .. & 0 & 0 \\ 0 & 1 & .. & 0 & 0 \\ : & & & : & \\ 0 & 0 & .. & 1 & 0 \end{pmatrix}$$

Definition 29 (Time Elapsing) *Given a TPN with a marking $\hat{m}$, a clock vector $\hat{h}$, and a time marking $\hat{\mu}$. When one time unit elapses, the values change in the following way:*

$$\tilde{m} = \hat{m} + \sum_{\tau \in T | \hat{h}(\tau)=1} \tau^{+} \tag{6.7}$$

$$\tilde{h} = max(\hat{h} - 1, 0) \tag{6.8}$$

$$\tilde{\mu} = \hat{\mu} \cdot \mathcal{Q} \tag{6.9}$$

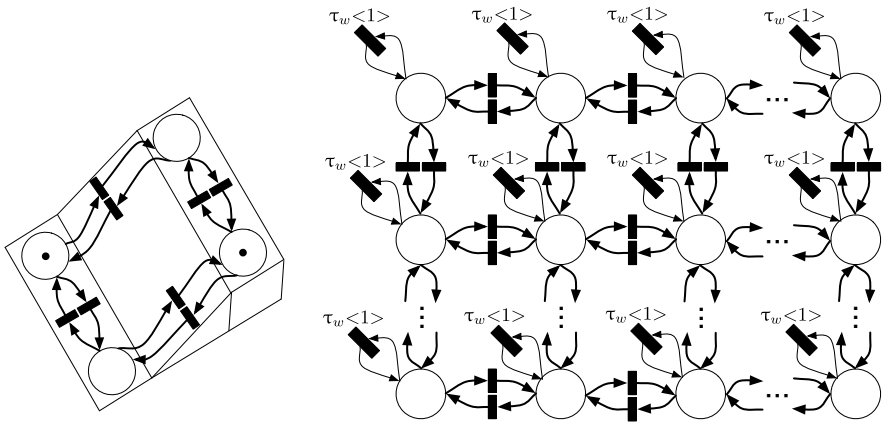
By applying this to the example, the following is obtained:

$$\tilde{\mu}_1 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \tilde{m}_1 = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \text{ and}$$

$$\tilde{h}_1 = \begin{pmatrix} 3 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

6.5 Translating Model into TPN

This section explains how spatio-temporal group scheduling problems using Timed Petri nets (Section 6.4) given the assumptions from Section 6.3 are modeled. The model



(a) Topography model. (b) Extended model with waiting transitions τ_w.

Figure 6.3: Simple and extended grid topology model.

consists of two parts: the first part shows the modeling of the topology of the world which includes movement abilities. The second part addresses the modeling of tasks. Finally, both parts are composed together.

The physical space of the world is discretized into cells. Each cell is modeled by a single place of the TPN. Possible movements between cells are modeled by timed transitions as shown in Figure 6.3. The firing time (duration in which the token is kept in the transition) states the time required in order to move from one cell to the next one. The movement time between two cells c_1 and c_2 must not be symmetric: as shown in Figure 6.3(a)[3], different topographical models can be supported. Due to this topography, moving from c_1 to c_2 might be faster based on terrain characteristics than taking the return path.

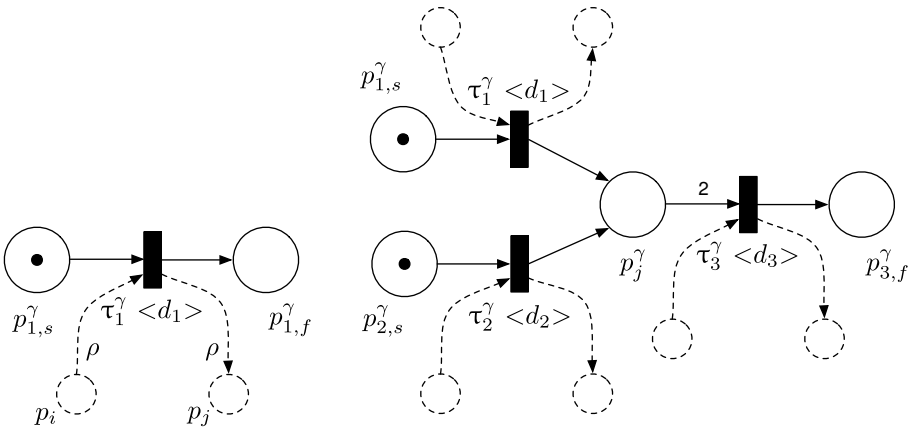
Robots are modeled as tokens in the net. The number of robots in a cell is translated into the same number of tokens and put into the respective places of the TPN. The result of translating the grid, shown in Figure 6.1 (Section 6.3), into a TPN is depicted in Figure 6.3(b). Inner cells[4] have four movement possibilities to adjacent cells indicated by the timed transitions.

A TPN is forced to progress, i.e., enabled transitions are required to fire (according to Definition 27 and Definition 28). According to this behavior, a robot would be forced to move. Therefore, additional waiting transitions τ_w are introduced and modeled at each place as shown in Figure 6.3(b). This allows to wait at a current place.

It is possible that multiple robots are positioned in the same place. Since a firing transition is blocked during firing, it is impossible that multiple robots (> 1) perform the same action (moving, waiting). Therefore, multiple waiting and moving transitions

[3]Example net shown in Figure 6.2 from Section 6.4 which has been mapped onto the topography.

[4]There are border and inner cells in the grid: the former are located on the border and have either two or three movement possibilities to adjacent cells while the latter have four movement possibilities. Each grid has exactly four corner (border) cells with a degree of two.



(a) A simple task. (b) Dependencies between tasks.

Figure 6.4: Task modeling.

have to be modeled. Since the overall net is conservative (number of robot tokens does not change over time), the needed number of transitions for each place and movement is equal to the total number of robots. Figure 6.3(b) shows only one waiting and one moving transition, the remaining ones are omitted for simplification.

A simple task is modeled by two places and one transition as shown in Figure 6.4(a). The duration of the task execution is denoted by d_1. Hence, d_1 is assigned to the timed transition τ^{γ}_1 between both places and states the execution time.

The two places are denoted by $p^{\gamma}_{1,s}$ and $p^{\gamma}_{1,f}$ which reflect the status of the task, i.e., a token on $p^{\gamma}_{1,s}$ marks the task as "ready to run" while $p^{\gamma}_{1,f}$ states that the task has been executed. In order to distinguish model elements (places and transitions) from the space model and the task model, the index $^{\gamma}$ is used for all task-oriented elements.

Location-independent tasks are modeled as a triple: $(p^{\gamma}_{i,s}, \tau^{\gamma}_i, p^{\gamma}_{i,f})$, with i as task id, s and f mark the start and final state of the task, respectively. The number of tokens on $p^{\gamma}_{1,s}$ states the multiplicity of task execution. If a task is location-dependent, i.e,. the task has to be executed at a certain location, then the task model has to be connected to the space model as indicated by the dashed lines in Figure 6.4(a). The task execution starts at place p_i and ends at place p_j. This modeling technique allows to move and execute the task at the same time if $i \neq j$. With the parameter ρ, the number of robots which are required for the joint execution can be specified.

Figure 6.4(b) shows the modeling of three dependent tasks $\tau^{\gamma}_1, \tau^{\gamma}_2$ and τ^{γ}_3 with durations d_1, d_2 and d_3. Task τ^{γ}_3 depends on task τ^{γ}_1 and τ^{γ}_2. In order to model predecessor / successor relations, *joint* places are introduced: a joint place is a common place of a predecessor task i and a successor task j by replacing $p^{\gamma}_{i,f}$ and $p^{\gamma}_{j,s}$ by p^{γ}_j. Only if both tasks (τ^{γ}_1 and τ^{γ}_2) have been executed and the place p^{γ}_j is marked with two tokens, the task τ^{γ}_3 can be executed. The multiplicity of the arc $(p^{\gamma}_j, \tau^{\gamma}_3)$ has to be marked with 2 or, in general, with the number of preceding tasks.

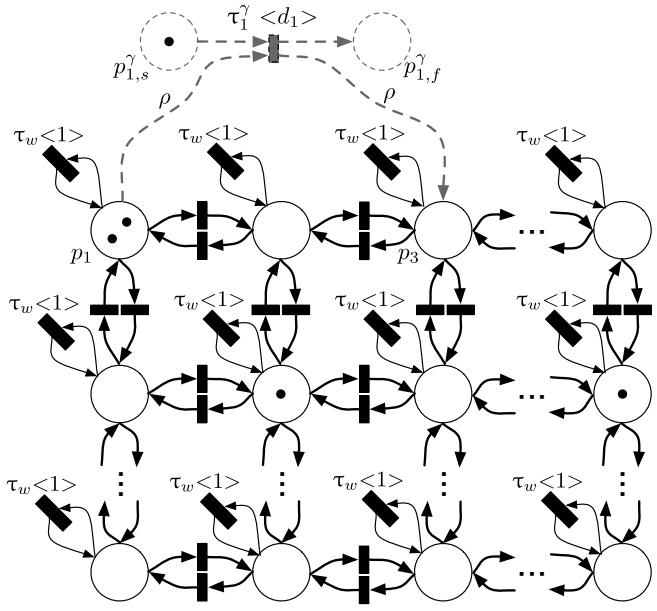


Figure 6.5: Modeling a complete scheduling problem.

After modeling the physical space as well as the tasks, both parts can be composed together in one TPN as shown in Figure 6.5. For clarity, multiple waiting and movement transitions have been omitted. The final model shows four tokens and, hence four robots. If $\rho = 2$, the task τ_1^γ can be executed immediately since the robots have already been positioned correctly (at place p_1). However, if $\rho = 4$ requires the two robots on place p_1 to use the waiting transitions until the other remaining ones have arrived. During that time, the other two robots have to take the shortest path in the grid, i.e, the city block distance, to place p_1. If at least ρ tokens are on place p_1, the transition p_1^γ becomes enabled and due to the maximal step semantic, p_1^γ fires accordingly and keeps ρ tokens for a duration of d_1 time units in the transition. Afterwards, the place $p_{1,f}^\gamma$ becomes marked with a token indicating that the task has been executed and ρ tokens are placed on p_3. Since the execution of τ_1^γ incorporates movement, d_1 must be large enough in order to move to place p_3, i.e., d_1 must reflect the movement as well as the task execution time. This has to be checked at modeling time.

The overall goal is to execute all real-space-time tasks according to their requirements. This is represented in this model by a marking of the net in which all places $p_{*,f}^\gamma$ of these tasks are marked. Additionally, the time required to reach such a marking, starting at some initial marking, should be minimized.

6.6 Schedule with Minimal Makespan

A schedule with minimal makespan is defined as an assignment of tasks to nodes that minimizes the overall time for executing all tasks (optimal schedule concerning time). After this schedule is found, there exists no other schedule with a smaller makespan. The schedule can be found by translating the problem into a shortest path problem on a graph:

For this, the state space graph is spawned. Every node in the graph represents a state and the edges indicate the possible transitions[5] from one state to another state. The edges are weighted according to the time required in order to obtain the new state. A particular state is called *goal state* which indicates a marking in which all places $p^{\gamma}_{*,f}$ are marked with a token. This state reflects that all tasks have been executed. Finding the shortest path (minimal time) from the initial state to the goal state, represents the schedule, i.e., the sequence of robot movement together with task execution given as a sequence of maximal step firing.

In Section 6.4, the TPN's state is defined as $s = (m, h)$. In the following an extended state $\xi = (\mu, h)$ is used which applies time markings instead of markings. Using ξ the state equation given in [75] can be applied in order to calculate future markings:

$$\mu_{i+\Delta t} = \mu_i \cdot \mathcal{Q}^{\Delta t} + C \cdot \Psi_\sigma \tag{6.10}$$

with

$$\Psi_\sigma = \sum_{j=1}^{\Delta t} B^{(j)} \cdot \mathcal{Q}^{\Delta t - j} \tag{6.11}$$

The parameters in Equation 6.10 and 6.11 have the following meaning:

- C is the (timed) incidence matrix $C \in M(|P|, |T| \cdot d)$[6]. An incidence matrix consists of submatrixes $C^{(k)} \in M(|P|, d)$ with $k \in \{1, .., |T|\}$. Given the example shown in Figure 6.2, the matrix is of the form: $C \in (4, 8 \cdot 5) = (4, 40)$. Each submatrix $C^{(k)}$ corresponds to one transition τ_k and indicates the impact of that transition, i.e., τ^-, τ^+ together with the timed behavior (how long the tokens are kept in the transition). Therefore, C contains all timed information of the impact of firing transitions.

- $B^{(j)}$ is the bag matrix for the jth step of the firing sequence σ. In Definition 27, the maximal step $\mathcal{B}$ has been introduced. A firing sequence is called $\sigma = (\mathcal{B}_1, \mathcal{B}_2, ..)$ which shows a firing sequence of maximal steps. B is the bag matrix $B \in (d \cdot |T|, d)$ of the maximal step $\mathcal{B}$. Similar to C, B also consists of $|T| = 8$ submatrixes. If $\mathcal{B}_1 = \{\tau_1, \tau_5\}$, then the 1st and the 5th submatrix consists of diagonal 1s. All other positions in these submatrixes (including the other submatrixes) remain 0.

[5] Transitions define here the possible state change.

[6] The parameter d is given by Definition 26.

- Ψ_σ is the (timed) Parikh matrix which includes the bag matrix B and the progress matrix $\mathcal{Q}$. Summarizing the equations, the bag matrix B indicates the maximal step $\mathcal{B}$ and, thus, states which transitions in the net shall fire. The firing sequence σ include the information when a maximal step shall fire (ordering of firing). C contains all information what happens to the net (impact) if a given maximal step fires (put tokens into transitions and release tokens from transitions). Last, but not least, the main characteristic of a TPN is its timed behavior. Hence, $\mathcal{Q}$ is used in order to state the time elapsing.

In [75], the correctness of Equations 6.10 and 6.11 are shown. With the help of (6.10) and (6.11), the extended state space is spawned. The shortest path to a goal state has to be found, i.e., a state with a time marking

$$\mu_x = \begin{pmatrix} * & * & * & \dots \\ * & * & * & \dots \\ * & * & * & \dots \\ * & * & * & \dots \\ * & * & * & \dots \\ 1 & * & * & \dots \end{pmatrix} \begin{matrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p^\gamma_{1,s} \\ p^\gamma_{1,f} \end{matrix} \tag{6.12}$$

Considering the example grid of Figure 6.2 again and attaching one simple task model (as given in Figure 6.4(a)), the resulting composed net has six places in total $(p_1, .., p_4, p^\gamma_{1,s}, p^\gamma_{1,f})$. Since, this model has only one task, the goal state is defined by a marking of the net in which exactly one token is placed on $p^\gamma_{1,f}$. As a consequence if $p^\gamma_{1,f}$ is marked with a token, then by definition $p^\gamma_{1,s}$ is marked with zero tokens. Tokens on other places $(p_1, .., p_4)$ do not matter. The marking μ_x shows the goal state.

Performing an analysis of the net, it might be possible that the goal state with time marking μ_x is not reachable. This might happen due to two reasons: there are insufficient resources for task execution or due to structural problems, i.e., circular task dependencies or tasks in disconnected areas of the topology.

In the former, non-reachability based on insufficient resources can be detected in advance: let be $\rho_{max} = \max(\rho_i)$ and n_γ the number of tasks. In particular, in case of dependent tasks, n_γ denotes the number of tokens required for modeling the tasks, i.e., dependent tasks do not count since they are not modeled using a separate token. If $|\mu_0| < n_\gamma + \rho_{max}$, then no goal state is reachable. Considering the example net of Figure 6.2 and an attached task as shown in Figure 6.4(a) with $\rho_{max} = 1$, then the number of robot tokens is two and $n_\gamma = 1$. The inequality $(|\mu_0| < n_\gamma + \rho_{max} \Leftrightarrow 3 < 2)$ is not true and, therefore, a goal state exists which is true in this case. If $\rho_{max} > 2$, then no goal state is reachable. If there are at least ρ_{max} tokens in the topology (space) model, then a goal state is always reachable. This holds for the timeless as well as for the timed net since waiting transitions τ_w enable to stay at a certain position.

In the latter, structural problems, i.e., circular task dependencies as well as tasks in disconnected areas of the topology have to be avoided at modeling time. This can be done manually while creating the model or afterwards based on model checking.

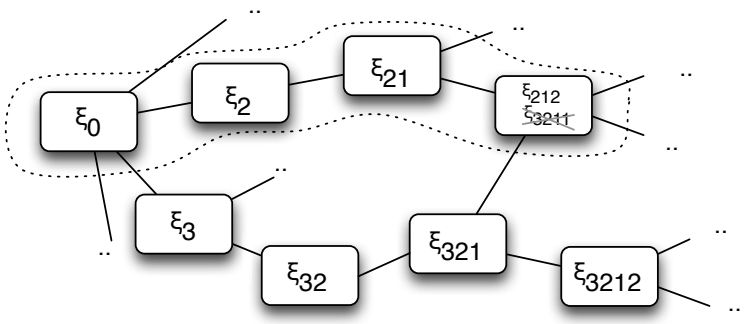


Figure 6.6: Shortest path in reachability graph.

Next, the conservativeness is analyzed. The marking m of the net is not conservative. Considering only the marking, tokens are kept arbitrary times (as stated by the timed transition τ) in the transition. Due to τ^- and τ^+ tokens are removed from and added to the marking. This might lead to non-conservativeness ($m_i \neq m_j$) for a marking m_i and m_j with $i \neq j$. However, the net is conservative with respect to the time marking μ:

$$\forall \mu, |\mu| = \sum_{i=1}^{|P|} \sum_{j=1}^{d} \mu^{i,j} = \text{const} \tag{6.13}$$

The state space of time markings is finite. Each clock vector $h(\tau)$ can only have the values $0, \ldots, D(\tau)$, i.e., the clock vector state space is finite, too. Since both elements are finite, the overall state space which consists of the combined (extended) state $\xi = (\mu, h)$ is also finite.

A schedule with minimal makespan can be found by finding the shortest path in the state space to a goal state by applying the usual algorithms, e.g., Dijkstra's algorithm [15]. Figure 6.6 depicts the shortest path in the reachability graph. The graph consists of all states that are reachable from a given initial state, i.e., given by the initial marking of the net. Imagine, there are two firing sequences of transitions σ_1 and σ_2. The initial state is given by ξ_0. The sequence σ_1 generates, amongst others, the following states: ξ_2, ξ_{21} and ξ_{212}. The sequence σ_2 generates: $\xi_3, \xi_{32}, \xi_{321}$ and ξ_{3211}. The firing sequence σ_1 and σ_2 lead to the same state (ξ_{212} and ξ_{3211}). For simplicity, it is assumed that each state change costs 1 time unit[7]. Therefore, the path created by σ_1 is shorter in terms of time than the path created by σ_2. In this case, the shortest path algorithm uses relaxation in order to update the distance information (time distance).

The complexity depends on two aspects: the spawning of the state space, which has an exponential complexity, and the shortest path algorithm, which has a polynomial (or better, depending on the actual algorithm) complexity in the number of states. The overall complexity is dominated by the spawning of the state space.

[7]Based on the modeling, the duration of state change corresponds to the duration of the timed transition.

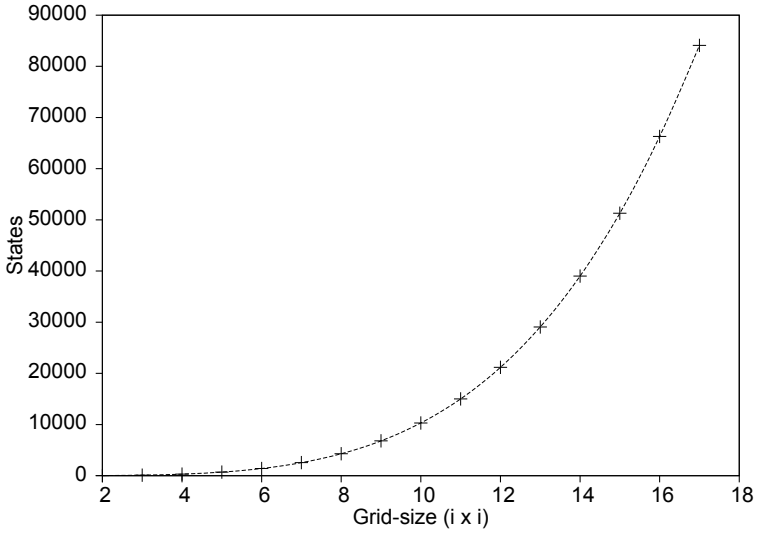


Figure 6.7: States depending on grid-size with 2 robots and 1 task.

6.7 Case Study

For the evaluation, a case study has been performed. As stated in Section 6.6, the computation of the schedule with minimal makespan is comprised of two stages: first, the reachability graph which consists of the states of the net is spawned and second, the shortest path from the initial state to the goal state of the graph has to be found. The computation of the reachability graph has an exponential complexity in the number of robot tokens and places of the topology (space) model in the net. Petri nets are used to model concurrent behavior. In this approach, tokens "move" around in the net independent of other tokens. Spawning the state space has to take this into account resulting in the high complexity. Modeling tasks requires to add "task-specific" tokens that indicate the state of the task (ready, executed). These dedicated tokens have to stay in the task-specific model and, hence, are not comparable with robot tokens that stay in the topology (space) model. Due to the complexity, the case study is limited to rather small test cases.

The topology (space) model is discretized into a 12×12 grid. Two robots and five tasks are modeled. Three tasks are coupled by a precedence, the other tasks are independent. Four tasks need one robot for execution, the remaining one requires two robots. Based on the task specification, three tokens are required in order to model the tasks (independent as well as dependent ones). Two additional tokens are required in order to model the robots resulting in a total of five tokens: the initial marking is $|\mu_0| = 5$, $n_\gamma = 3$ and $\rho_{max} = 2$. The inequality $|\mu_0| < n_\gamma + \rho_{max}$ is not true and, therefore, a goal state is reachable.

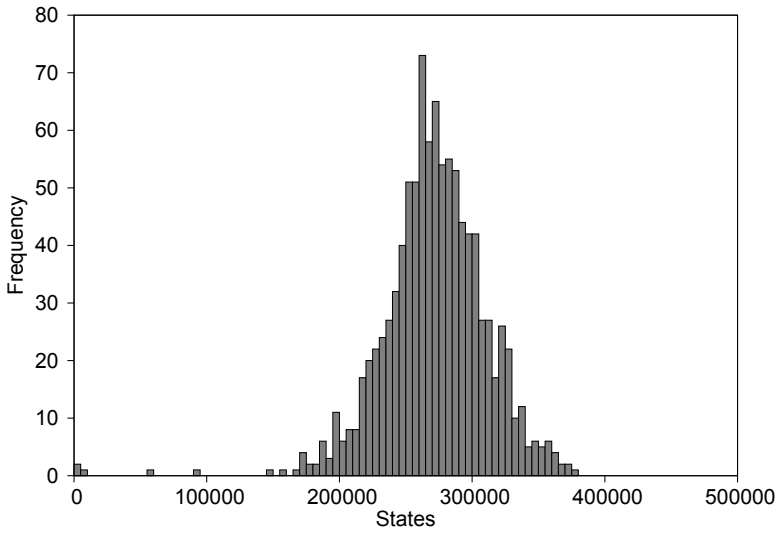


Figure 6.8: State distribution on a 12x12 grid with 2 robots and 5 tasks.

The execution locations of the tasks as well as the initial positions of the robots are generated randomly. A random terrain is assumed defined by the following distribution for movement transitions: 50 % of the transitions have 1 time unit, 12.5 % have 2 and another 12.5 % have 3 time units. The remaining 25 % simulates obstacles indicating no possible movement.

The simulations have been performed with INA (Integrated Net Analyzer) [72]. In total, 1000 simulations were conducted. The computer which have been used for the simulations were equivalent to a standard workstation with Intel Core i7 (Nehalem) CPU with 2.66 GHz and 6 GB RAM. The execution time of a single run on the machine on a single core took up to several hours depending on the grid-size and the number of robot tokens in the net.

Figure 6.7 shows the number of states as a function of the grid-size ($i \times i$). The number of places is given by $|P| = i^2$. As depicted, the number of states grows polynomially in the grid-size by keeping the number of robots and tasks constant. In Figure 6.8 the distribution of generated states for the reachability graph is shown for the full configuration.

6.8 Conclusion

This chapter has presented an approach for offline scheduling by Timed Petri nets. The approach consists of two model elements: the topology (space) model reflects topographical characteristics and the task model states task-specific characteristics. It has been

shown how tasks, robots and the topology of the physical world are modeled and translated into a TPN. The model enables to bind task execution to physical locations and specify the multiplicity of robots required to execute the task jointly. Furthermore, it has been shown how dependencies between tasks are modeled.

In order to execute the tasks, an assignment and a coordination of robots in space and time are required. The overall goal is to find a schedule with minimal makespan that incorporates all tasks. The presented solution is to analyze the TPN, spawn the state space and create the reachability graph in order to find the shortest path to a goal state. The goal state defines a dedicated marking of the net that reflects that all tasks have been executed. The shortest path represents the minimal order of states and, hence, the minimal order of transition firing in order to reach the goal state. This represents the schedule with minimal makespan.

The case study showed a long runtime of the simulations. Therefore, the approach fits as an offline scheduler. Although, the approach finds the schedule with minimal makespan, an online scheduler has to perform significantly faster. Therefore, suitable algorithms are required in order to establish fast online scheduling decisions.

Chapter 7

Swarm Space-Time Scheduling (Online)

The core-component of the runtime system is the space-time scheduler that is responsible for resource management. Given a set of spatio-temporal actions and a set of mobile robots that move through physical space, a main objective remains in finding a mapping of actions to robots. This chapter addresses the scheduling problem and provides a solution that is based on the path-velocity decomposition. The outcome of the scheduler is a schedule that assigns actions to robots under consideration of the spatio-temporal constraints. If movement is necessary for the execution of the respective action, then the scheduler computes collision-free spatio-temporal trajectories for the robots in order to arrive at the desired locations in time. The scheduler is an online scheduler that schedules sets of actions at runtime and merges the computed schedule into the global system schedule. The collision avoidance addresses both static and dynamic obstacles. The scheduler consists of two components: a *job scheduler* that uses a heuristic and performs a coarse-grained scheduling and a *trajectory planner* that takes the output of the job scheduler and computes spatio-temporal trajectories.

7.1 Introduction

The scheduling which includes the computation of collision-free, spatio-temporal trajectories remains a major challenge which requires an *online-scheduler* that schedules the actions in space and time under consideration of the spatio-temporal constraints.

This chapter is structured as follows: Section 7.2 shows related work in this field. In Section 7.3, assumptions and the model of the world and the scheduler is shown. The scheduler is composed of two components: The *job scheduler* is presented in Section 7.4 and the *trajectory planner* is described in Section 7.5. An evaluation of the scheduler together with the produced results are presented in Section 7.6 and include a complexity analysis as well as benchmark results. Finally, Section 7.7 summarizes this chapter.

7.2 Related Work

In [24], a fast scheduling algorithm called *Tercio* is presented that assigns tasks with spatio-temporal constraints to agents. A task sequencer is used that computes a schedule in polynomial time for multi-agent systems. The authors are able to satisfy upper and lower bound temporal deadlines as well as spatial restrictions.

In [2], the *Law Enforcement Problem* (LEP) is presented. This problem shows policemen on their patrol that have to react to incidents (tasks) that also have to be addressed by multiple policemen. They present the *FMC_ TA* algorithm that is based on a heuristic which best utilizes the capacity of the team for the LEP.

Both approaches (Tercio as well as FMC_TA) also schedule spatio-temporal tasks in space and time. The main difference to the approach presented in the following is that Tercio as well as FMC_TA are based on a task model that clearly states the execution location. Furthermore, none of the approaches computes spatio-temporal trajectories.

In general, one has to distinguish between path coordination and path planning. In the latter a path is not *a priori* computed. The resulting path planning problem (PPP) addresses the problem of computing collision-free paths in Cartesian space. Path coordination, in contrast, addresses the problem of coordinating robots while they move along a predefined path in order not to collide.

The path coordination problem was considered in [67], where a coordination diagram is presented that avoids collisions and deadlocks of two robot manipulators. Later, [91] extended this approach by addressing the path coordination problem in order to avoid collisions with several robots.

The approach presented in this thesis is based on the Path-Velocity-Decomposition as presented in [40] that decomposes the problem into the PPP and the velocity planning problem (VPP). The main reason for performing the decomposition is to face the complexity of the problem. In general, "the complexity of the PPP is exponential in the number of degrees of freedom, which for a point robot is the same as the dimension of the space in which the robot is embedded. For instance, in 3D space, a point robot has three degrees of freedom." [40]. Decomposing the problem from 3D space-time ($x \times y \times t$) which means $\mathcal{O}(n^3)$ to two times 2D reduces the complexity to $\mathcal{O}(2n^2)$.

There are several revisions of the original approach presented in [40]: In [19], a prioritized planning scheme is proposed that introduces robot priorities. In [102], priorities are combined with the potential field method for conflict resolution. A decentralized approach that also uses priorities is presented in [1].

In [22], the approach is extended by introducing alternative routes from a start point to a target point that are adjacent paths. The switching from one path to an adjacent path also allows spatial collision avoidance of dynamic obstacles and not solely temporal collision avoidance by velocity profile adjustment. It considers multiple spatial paths and, thus, avoids the problem of the single spatial path approach.

7.3 Assumptions and Model

This section states assumptions and presents the model of the world and the scheduler.

7.3.1 The Model of the World

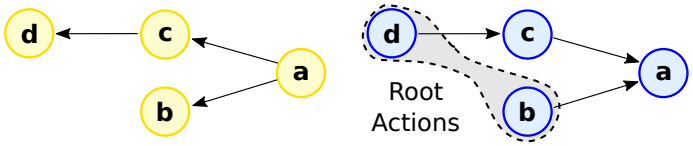
The world is the geometry in which all entities reside. The world is mapped to a 2-dimensional surface[1] which represents the physical space. A third dimension which indicates the time is added. All entities have their own geometry which is represented as a 2D-polygon and a description of their temporal behavior. An entity can be either static or dynamic. The first one is time-invariant and, thus, the temporal behavior has no influence on the physical location. Dynamic entities are able to change their physical location over time. A function $f : T \rightarrow \mathbb{R}^2$ describes the spatio-temporal relation.

R denotes the set of mobile robots (dynamic entities) and O^s denotes the set of static obstacles (static entities). From the "perspective" of a robot r_i, all other robots $R \backslash \{r_i\}$ are considered as dynamic obstacles O^m.

7.3.2 Actions and ActionSuites

An action $a \in A$ is defined as a tuple (g, t_{min}, t_{max}, d). The first element states the geometry g that consists of 2D points that mark the space window on a plain surface. The time-window is denoted by t_{min} and t_{max}. Together (g, t_{min}, t_{max}) describe the space-time-window in which a shall be executed while d denotes the worst-case execution time.

If an action a_i depends on another action a_j, then this is denoted as: $a_i \rightarrow a_j$; if $a_j \mid\mid a_i$, then a_j, a_i are independent, with $i \neq j$. In the following, a dependency graph[2] (Figure 7.1(a))—showing inter-dependencies among actions—and an execution graph[3] (Figure 7.1(b))—showing the order of execution of actions—are considered. The two functions *succ*(..) and *reach*(..) that are introduced in the following are operations on the execution graph.



(a) Dependency graph. (b) Execution graph.

Figure 7.1: Dependent jobs.

[1] For simplicity and due to the hardware equipment of the testbed (Appendix A), the model is based on two space dimensions. However, the approach presented in this thesis does also work with three dimensions.

[2] "$\rightarrow$" means "depends on": $a_2 \rightarrow a_1$ means a_2 depends on a_1.

[3] "$\rightarrow$" means "proceeds": $a_1 \rightarrow a_2$ means a_1 proceeds a_2 or a_2 succeeds a_1 (used to model data flow from a_1 to a_2).

Action	$succ(..)$	$reach(..)$
a	–	–
b	a	a
c	a	a
d	c	c, a

Table 7.1: Successors and reachability-set.

A function $succ(a)$ delivers all successors of a (based on the topological sorting of the execution order of actions). If $a_i \rightarrow a_j$, then $succ(a_j) = a_i$. A function $reach(a)$ delivers all actions that are reachable from a (reachability-set of a). An action a_i is reachable from a_j if $a_i \in succ(a_j)$ or transitive if $a_i \in succ(a_k) \wedge a_k \in succ(a_j)$. Table 7.1 shows the direct successors and the reachability-set of the actions shown in Figure 7.1.

An ActionSuite $as \in AS$ is defined as a container for actions. An action a is assigned to exactly one suite: if $a \in as_i$, then $\forall as_j \in AS : a \notin as_j$, with $i \neq j$. Each action has to belong to a suite: $\forall a \in A : a \in AS$. All depending actions have to reside in the same suite. However, it is also possible to put non-depending actions in the same suite.

If $a_i \rightarrow a_j$ or $a_j \rightarrow a_i$, with $i \neq j$, then a_i, a_j have to reside in the same suite as. If $a_i \,||\, a_j$, then a_i, a_j can either reside in the same suite or in different suites. There shall be no inter-suite dependencies between two suites as_i and as_j: $\forall a_k^{as_i} \in as_i, \forall a_l^{as_j} \in as_j : a_k^{as_i} \,||\, a_l^{as_j}$. Since the dependency graph shown in Figure 7.1(a) is connected, all contained actions (a, b, c, d) have to reside in the same suite.

An action a^{r_i} is considered a root action if there is no other action a^{r_j} such that $a^{r_i} \in reach(a^{r_j})$, with $i \neq j$. Let A_{as}^r be the set of root actions of suite as. Each action that is reachable from A_{as}^r has to be in the same suite, i.e., $\forall a^r \in A_{as}^r : reach(a^r) \subseteq as$. The execution graph (Figure 7.1(b)) has two root nodes (b, d) since they never appear in any reachability-set (Table 7.1).

The elements of a suite as together with their dependencies form a directed, acyclic graph. The graph can be either connected or disconnected. A topological sorting on the graph is performed which generates the order in which the actions are scheduled (starting with root actions).

As explained in Section 4.6.3 (page 45) and Section 4.6.4 (page 46), a contract is created for every suite between the system and the application if *sched()* and *exec()* are successfully checked. A contract provides guaranteed resource allocation.

7.3.3 Transaction-based Scheduling

As introduced in Section 4.6.2 (page 44), an ActionSuite as has three operations in order to interact with the swarm runtime system: `schedule()`, `reschedule()` and `unschedule()`.

A *transaction* is a set of actions that logically belong together and, thus, are grouped together. Spatio-temporal constraints and logical dependencies describe the relations

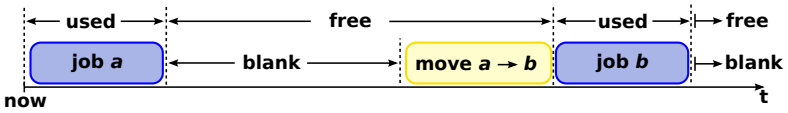


Figure 7.2: Local schedule $\mathcal{S}_i$ of a robot.

among the actions as well as relations to space and time. Each of the system operations starts a new distributed transaction Tx. Tx consists of all actions that belong to as: $Tx = \{a_1, a_2, .., a_n\}$. A contract is only created if all actions of Tx are successfully scheduled (*sched()* and *exec()* holds): $\forall a \in Tx \mid sched(a) \wedge exec(a)$. If $\exists a \in Tx \mid \neg sched(a) \vee \neg exec(a)$, then Tx is aborted and no contract is issued. In case a partial assignment of actions to nodes has already been performed, the system attempts to unschedule them.

However, the objective of the scheduler is to compute a schedule for Tx that maps the actions $a_1, a_2, .., a_n$ to the robots under consideration of the spatio-temporal constraints of $a_1, a_2, .., a_n$. Actions are scheduled sequentially and in the order of the topological sorting. The topological sorting depends on the set of root actions A^r_{as}, the inter-action dependencies and the spatio-temporal constraints.

The scheduler computes for each action a a job (p, t, r): $p \in g$ is the execution location, $t \in [t_{min}, t_{max} - d]$ is the start time and $r \in R$ is the executing robot. If r is not already at position p, then the scheduler computes a collision-free, spatio-temporal trajectory in order to move r to p. The trajectory consists of linear segments and is given by a list of spatio-temporal points $[(x_1, y_1, t_1), (x_2, y_2, t_2), .., (x_m, y_m, t_m)]$, where a segment is defined by a pair of points $[(x_i, y_i, t_i), (x_{i+1}, y_{i+1}, t_{i+1})]$. Let $\vec{x} = (x, y)$, then $\Delta\vec{x}_i = \vec{x}_{i+1} - \vec{x}_i$ and $\Delta t_i = t_{i+1} - t_i$, $t_{i+1} > t_i$ holds. The robot moves along the trajectory with velocity $v = \frac{|\Delta\vec{x}|}{\Delta t}$; the velocity is constant for a segment, but different segments may have different velocities: $[v_1, v_2, v_{m-1}]$.

Let $\mathcal{S}^g = \{\mathcal{S}_1, \mathcal{S}_2, .., \mathcal{S}_k\}$ be the global schedule, then $\mathcal{S}_i$ is the local schedule for robot r_i; $k = |R|$. The initial schedule is the empty schedule: $\mathcal{S} = \{\}$. A schedule $\mathcal{S}$ consists of jobs: $\mathcal{S}_i = \{j_1, j_2, .., j_l\}$. A job can be either a spatio-temporal action (j^a) or a spatio-temporal trajectory (j^t). Figure 7.2 shows a simple schedule that consists of two actions and one movement job.

If a spatio-temporal action (j^a) has been successfully scheduled and committed, then j^a is included in the schedule $\mathcal{S}_i$. No modification of j^a is allowed afterwards, unless explicitly requested by *unschedule(..)* or *reschedule(..)*. In contrast, all spatio-temporal trajectories are replaceable if they do not violate the following two conditions:

A spatio-temporal trajectory (j^t) is not replaceable if j^t or a part of it is already in the *frozen horizon* (definition is given in the following). Second, j^t is not replaceable if it interferes with spatio-temporal actions (j^a) such that the execution of j^a cannot be guaranteed. Hence, the movement job $a \rightarrow b$ in Figure 7.2 is arbitrarily replaceable if it has no influence on the execution of job a and b.

Free slots of a schedule $\mathcal{S}$ are defined as the gaps before, between or after spatio-temporal actions by omitting all movement jobs. *Blank slots* indicate blank gaps in

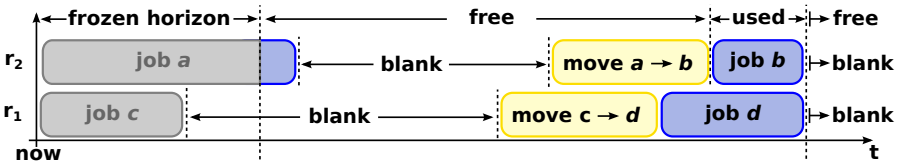


Figure 7.3: Global schedule $\mathcal{S}^g$ with frozen horizon fh.

between action or movement jobs. If there are no movement jobs, then the set of free slots and the set of blank slots are equal. Otherwise, the size of a blank slot is less or equals to the size of a free slot. Figure 7.2 has two free slots: one between a and b and one after b. It is assumed that a is scheduled for *now* such that there is no gap before a.

If a new action z with time constraints shall be scheduled, the time window must be large enough in order to execute z with duration d not violating the time constraints: $t_{curr} + d \leq t_{max}$; t_{curr} defines the current point in time. The computation of the schedule as well as sending the message with the schedule information (j^a, j^t) to robot r requires time. Let t^{sched} be the duration for the computation of the schedule and t^{msg} be the duration for sending the message to r. Since time progresses constantly, the *frozen horizon* (fh) is a time interval in which no modification of the global schedule $\mathcal{S}^g$ is allowed as depicted in Figure 7.3. The duration of fh is defined by t^{fh} and the interval is dynamically adjusted to $[t_{curr}, t_{curr} + t^{fh}]$, with $t^{fh} \geq t^{sched} + t^{msg}$. Due to fh, the condition above $(t_{curr} + d \leq t_{max})$ must be adjusted to $t^{fh} + d \leq t_{max}$.

Determining t^{sched} and t^{msg} accurately depend on context data; these include the number of dynamic and static obstacles, their shape detail-level, the number of spatio-temporal trajectory segments and the current workload of jobs. This could be done approximately. However, t^{sched} and t^{msg} are estimated during system execution. In order to determine t^{sched}, the duration of each invocation of the scheduler is measured and an estimate based on the time the scheduler takes for computation is computed. Determining t^{msg} requires to estimate the average message delay. Since the message that contains the schedule information (j^a, j^t) has to be acknowledged by robot r, t^{msg} is estimated by measuring the roundtrip time and dividing it by 2 in order to obtain the simple message delay. This is performed iteratively each time the scheduler is invoked.

In the following, the size of a robot is shrunk to a point in 2D while all obstacles are buffered according to the robot's original size. This approach (*Cspace approach*) has been presented in [51].

7.4 Job Scheduling

The task of the scheduler is to schedule actions in space and time and assign them to robots. Let $Tx = \{a\}$ be a transaction that consists of only a single action a. Algorithm 1 shows the main steps the scheduler performs in order to schedule a. The function takes a as input and computes a spatio-temporal action job (j^a) and a spatio-temporal trajectory (j^t).

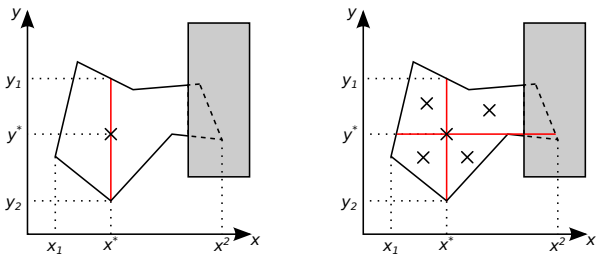
Algorithm 1 Job scheduling: Schedule single job

Input: Action $a = (g, t_{min}, t_{max}, d)$
Output: Job $j^a = (p, t, r)$, job $j^t = [(x_1, y_1, t_1), ..]$

```
function SCHEDULE_SINGLE(a)
    while loc ← next_location(a.g) do
        slots ← find_slots(loc, a.t_min, a.t_max, a.d)
        while slots ≠ [] do
            slot ← min_detour(slots)
            (j^a, j^t) = plan_job(loc, slot, a)
            if j^a ≠ null then
                return (j^a, j^t)
            end if
            slots = slots\{slot}
        end while
    end while
end function
```

7.4.1 Location Sampling

In order to find a location $p \in g$, the function $next\_location(..)$ recursively samples a new location candidate $p^* \in g \setminus \{O^s\}$. A sample is an interior point of $g \setminus \{O^s\}$ as depicted in Figure 7.4(a) and, thus, must not lie in an obstacle ($p^* \notin O^s$). If the scheduler was unable to schedule a using p^*, then g is sub-divided into up to four sub-areas $g_1, .., g_4$ by drawing a vertical and horizontal line through p^* as depicted in Figure 7.4(b). Afterwards, the same procedure is repeated on $g_1, .., g_4$. The algorithm terminates if either a valid sample has been found or a maximum sample-depth sd has been reached. Since each area has an infinite amount of points, sd prevents the algorithm from going into an endless loop. If sd is reached, the location sampling fails and, thus, the computation of the schedule fails. If a valid sample has been found, then $next\_location(..)$ returns that sample candidate.



(a) Interior point. (b) Sub-dividing area.

Figure 7.4: Location sampling.

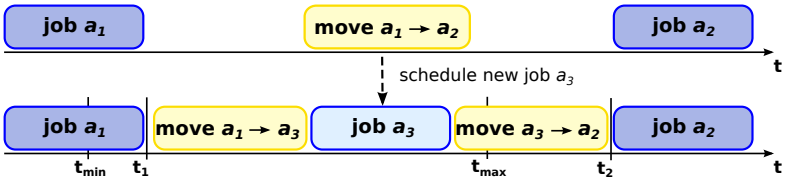


Figure 7.5: Job scheduling.

7.4.2 Determine Slot Candidates

Initially, the local schedule of each robot is the empty schedule: $\mathcal{S} = \{\}$. During system execution, the scheduler assigns new actions to robots such that $\mathcal{S} \neq \{\}$. Gaps in-between action jobs mark available (free) slots (movement jobs omitted).

If the scheduler assigns an action to a robot, the path that the robot has to move in addition is defined as detour since it produces additional effort for the robot. Figure 7.5 shows the transition of modifying the schedule. The scheduler checks if a new job a_3 can be assigned to a robot that already has scheduled actions (a_1, a_2). The free slot in-between is defined by the interval $[t_1, t_2]$. The execution location of job a_i is defined by $\vec{x}_{a_i}$. The current trajectory must be adapted such that the robot does not move straight from $\vec{x}_{a_1} \rightarrow \vec{x}_{a_2}$. The new trajectory which considers a_3 is then from $\vec{x}_{a_1} \rightarrow \vec{x}_{a_3} \rightarrow \vec{x}_{a_2}$. For performance reasons, the job scheduler is based on a heuristic, i.e., all obstacles are neglected here (static and dynamic) and it is assumed that the robot is able to move directly from $\vec{x}_{a_i}$ to $\vec{x}_{a_j}$. Thus, the distance between two execution location $\vec{x}_{a_i}$ and $\vec{x}_{a_j}$ is calculated by the Euclidean distance: $s = ||\vec{x}_{a_i} - \vec{x}_{a_j}||$.

In order to schedule the new action a_3, the scheduler computes all possible slots ($find\_slots(..)$, Algorithm 1) from all robots that match with the temporal constraints of the action and uses loc as execution location of a_3: $\vec{x}_{a_3} = loc$.

The scheduler marks a slot as a candidate if the following conditions hold: The robot is able to move from $\vec{x}_{a_1}$ to $\vec{x}_{a_3}$ (s_1) and execute a_3 not violating the deadline given by t_{max}:

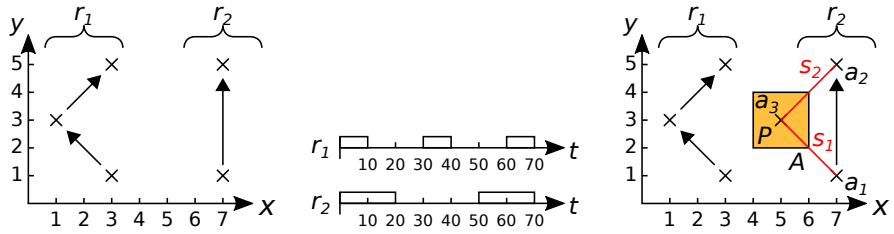
$$t_{max} - t_1 \geq \frac{s_1}{v_{max}} + d \tag{7.1}$$

The job a_3 can be executed and the robot is able to move from $\vec{x}_{a_3}$ to $\vec{x}_{a_2}$ (s_2) such that the execution start time of a_3 is after t_{min}:

$$t_2 - t_{min} \geq \frac{s_2}{v_{max}} + d \tag{7.2}$$

The length of the free slot is sufficient in order to move to $\vec{x}_{a_3}$, execute a_3 and move to $\vec{x}_{a_2}$:

$$t_2 - t_1 \geq \frac{s_1 + s_2}{v_{max}} + d \tag{7.3}$$



(a) Initial situation. (b) Local schedule $\mathcal{S}_1$ and $\mathcal{S}_2$ of r_1 and r_2. (c) Schedule a_3.

Figure 7.6: Schedule new action.

Finally, $find\_slots(..)$ returns a set of valid slots. In order to select the slot which causes minimal additional movement, $min\_detour(..)$ returns the slot with the minimal detour. The detour is defined as the way that the robot has to move in addition:

$$\Delta s := (s_1 + s_2) - s_{12} \tag{7.4}$$

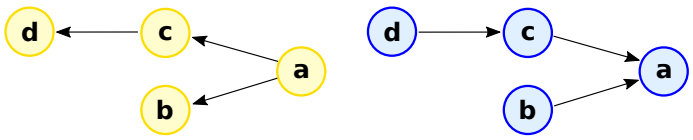
The original distance between $\vec{x}_{a_1}$ and $\vec{x}_{a_2}$ is given by s_{12} and Δs states the additional way. In case of an empty schedule, s_{12} and s_2 are both zero and, thus, $\Delta s = s_1$ holds.

Figure 7.6 shows an example with two robots r_1 and r_2. The initial situation is depicted in Figure 7.6(a): r_1 already has three scheduled jobs and r_2 has two scheduled jobs (a_1 and a_2). Figure 7.6(b) shows the local schedule of both robots; for simplicity, all movements jobs have been omitted here and, instead, only free slots are shown.

Now, a new action a_3 shall be scheduled (Figure 7.6(c)). The spatial constraints of a_3 are $x = [4, 6]$, $y = [2, 4]$ and the temporal constraint is given by $t = [10, 60]$. The duration d of a_3 is 10 and the maximum velocity of both robots is set to: $v_{max} = 0.5$. The location candidate is set to $loc = (5, 3)$[4]. For r_2 the detour over a_3 involves the two segments $s_1 = \sqrt{8}$ and $s_2 = \sqrt{8}$. The total detour is then $s_1 + s_2 \approx 5.66$ and takes 11.32 time units. The original distance is $s_{12} = 4$ and the additional way is then $\Delta s := 1.66$. The time required for moving along s_1 and s_2 takes 11.32 time units. The time for executing a_3 takes 10 time units which requires in total 21.32 time units. The only free slot that satisfies the conditions given in Equation 7.1 to 7.3 is the one provided by r_2 and, thus, the scheduler assigns a_3 to r_2. In the interval $[10, 60]$, r_1 has only free slots of 20 time units length.

Since the job scheduler is based on a heuristic, the schedule might not be feasible. Thus, the trajectory planner uses the proposed slot and checks feasibility by considering static as well as dynamic obstacles and computes a collision-free spatio-temporal trajectory ($plan\_job(..)$). If a_3 could not be scheduled under the current variable assignment, the scheduler discards the current slot and checks the next one. If none of the slots could be taken, the scheduler picks another location point and repeats the procedure.

[4]Based on the location sampling, this is the first candidate which is checked. If no schedule can be found, a new location is sampled.



(a) Dependency graph. (b) Execution graph.

Figure 7.7: Scheduling of dependent jobs.

7.4.3 Dependent Jobs

If a transaction consists of multiple dependent jobs, then the scheduler schedules the actions in Tx according to their dependencies. For this a topological sorting of the actions is performed. A dependency could be data that one action generates while a second action requires this data as input. Let $Tx = \{a, b, c, d\}$ and its elements have the following dependencies: $a \rightarrow b$ and $a \rightarrow c \rightarrow d$. Table 7.2 shows their specifications. Figure 7.7 shows the resulting dependency and execution graph respectively.

Action	t_{min}	t_{max}	d
a	20	100	5
b	0	100	5
c	0	50	5
d	10	100	10

Table 7.2: Action specifications.

Due to the specifications, certain variable assignments are not possible and, thus, the specifications have to be further pruned. The pruned action specifications are shown in Table 7.3. Equation 7.7 and Equation 7.8 state how the new values for t_{min} (t'_{min}) and t_{max} (t'_{max}) are recursively computed. For this, two auxiliary functions ($\alpha(..)$ and $\beta(..)$) are defined that determine the earliest possible finishing time and the latest possible start time of action s. Two other functions $dep(..)$ and $dep^{-1}(..)$ are introduced: $dep(s)$ returns all other actions that s depends on and $dep^{-1}(s)$ returns all actions that depend on s. The operation $\alpha(dep(s))$ is defined by applying the function $\alpha(..)$ on each element of $dep(s)$. The same holds for $\beta(..)$ and $dep^{-1}(s)$.

$$\alpha(s) := max\Big[\alpha(dep(s)) \cup \{s.t_{min}\}\Big] + s.d \tag{7.5}$$

$$\beta(s) := min\Big[\beta(dep^{-1}(s)) \cup \{s.t_{max}\}\Big] - s.d \tag{7.6}$$

$$s.t'_{min} := \alpha(s) - s.d \tag{7.7}$$

$$s.t'_{max} := \beta(s) + s.d \tag{7.8}$$

In order to determine $a.t'_{min}$, $\alpha(a) - a.d$ has to be computed. For this, all actions that a depends on have to be considered: $dep(a) = \{b, c\}$ and, then, the same has to be done for b and c ($dep(b) = \{\}$ and $dep(c) = \{d\}$). Finally, $a.t'_{min} := 25$ is obtained as listed in Table 7.3.

Next, it is shown how $d.t'_{max}$ is computed: For this, $\beta(d)$ has to be computed and all actions that depend on d: $dep^{-1}(d) = \{c\}$ have to be considered. Due to the recursion, $\beta(c)$ and, finally, $\beta(a)$ have to be computed. After computing the respective values, $\beta(d) = 35$ is obtained. After adding the duration of d, $d.t'_{max} := 45$ is obtained as listed in Table 7.3.

Action	t'_{min}	t'_{max}	d
a	25	100	5
b	0	95	5
c	20	50	5
d	10	45	10

Table 7.3: Implicit action specifications.

Based on the topological sorting of the jobs, there are different orders in which the jobs can be scheduled: d, c, b, a or d, b, c, a. In order to obtain the highest probability that the given jobs are schedulable, a higher priority is given to jobs with an earlier deadline. Thus, the jobs will be scheduled in this order: d, c, b, a. Each job is, thereby, scheduled as a single job by the function `Schedule_Single(..)`.

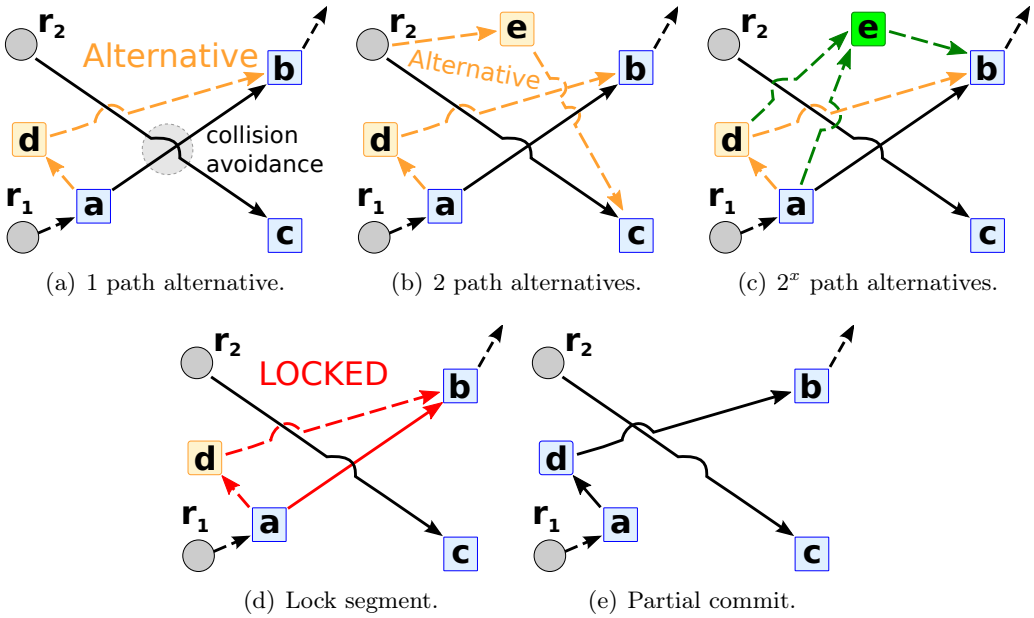
7.4.4 Periodic Jobs

Jobs can also be scheduled periodically, but with a fixed amount of repetitions. Periodic jobs are given as a tuple (g, t_0, d_p, d). Similar to the scheduling of single jobs, periodic jobs also have a duration d and a geometry g in which the job shall be executed. The temporal restrictions are changed to an initial start time given by t_0 and a period given by d_p, with $d_p > d$. During each period, one job has to be executed. Each job j_i ($i \in \{1, .., n\}$) has a time window given by $[t_{i-1}, t_i - d]$, with $t_i := t_0 + i \cdot d_p$.

The spatial constraint of a periodic job is fixed and given by g. However, as a feature, it is possible to set a flag that requests the scheduler to reuse the first location that is found in g for all instances of the periodic job. If the flag is not set, the scheduler is forced to find a new location for each instance of the periodic job in g.

7.4.5 Transactions

As already mentioned, a transaction-semantic is supported. If one action out of a set of actions in a transaction is not schedulable, then none of the actions will be scheduled. If a transaction Tx shall be scheduled, then the scheduler tries to find a suitable schedule that assigns all actions in Tx to robots. In case the scheduler found a suitable schedule $\mathcal{S}^{Tx}$ for Tx, then this schedule is marked as *pending* in the global schedule $\mathcal{S}^g$. Pending is defined as a state in which system resources are already allocated, but have not finally



(a) 1 path alternative. (b) 2 path alternatives. (c) 2^x path alternatives.

(d) Lock segment. (e) Partial commit.

Figure 7.8: The transaction-semantic causes path alternatives while scheduling new jobs. During the *uncertainty period* alternatives are locked.

been committed. After the schedule is computed, all nodes that are involved in the distributed transaction are notified. Each node that receives a message decides locally if it accepts or rejects the job. The job can only be rejected if the node is unable to execute the job. In case the node accepts the job, then it is merged into its local schedule $\mathcal{S}_i$.

If, finally, all nodes participating in the distributed transaction have commonly voted for commit, then the scheduler *commits* Tx. This transforms the state of $\mathcal{S}^{Tx}$ from pending to committed. In case at least one node has voted for abort, then $\mathcal{S}^{Tx}$ is aborted, i.e., all jobs in Tx are unscheduled and $\mathcal{S}^{Tx}$ is marked as aborted.

The period that starts right after $\mathcal{S}^{Tx}$ is computed and messages are sent in order to notify the involved nodes and a commit or an abort is issued, is called *uncertainty period* since $\mathcal{S}^{Tx}$ can neither be unilateral committed nor aborted. The notification phase is asynchronous, in order to not block the scheduler and instead enable the scheduling of new jobs. As already mentioned, the scheduling of a job includes the computation of a spatio-temporal trajectory.

During the *uncertainty period* a trajectory can be committed or aborted. If new jobs are scheduled concurrently, the scheduler can not modify a schedule that is still *pending* due to possible inconsistencies. Figure 7.8(a) shows the situation in which two robots (r_1, r_2) already have committed jobs: Each robot has a local schedule $(\mathcal{S}_1, \mathcal{S}_2)$, with $\mathcal{S}_1 = \{a, b\}$ and $\mathcal{S}_2 = \{c\}$. A new job d is scheduled, assigned to r_1 and $\mathcal{S}^{Tx}$ is marked as *pending*. Let $\mathcal{S}_1^* = \{a, d, b\}$ be the new schedule that shall replace $\mathcal{S}_1$. During the *uncertainty period* both schedules have to be maintained. This is due to the following reason: If, while $\mathcal{S}^{Tx}$ is still pending, a new job e shall be scheduled (which opens a

second transaction $\mathcal{S}^{Tx2}$), both schedules, $\mathcal{S}_1$ as well as $\mathcal{S}_1^*$ have to be taken into account in order to avoid robot collision as depicted in Figure 7.8(b). Therefore, $\mathcal{S}_1^*$ is defined as an alternative path for r_1. The alternative remains valid as long as no commit or abort has been issued. Here it is assumed that e is assigned to r_2. So, the alternative schedule for r_2 is $\mathcal{S}_2^* = \{e, c\}$. Since, $\mathcal{S}^{Tx}$ is still pending, the computation of the new spatio-temporal trajectory for the alternative schedule $\mathcal{S}_2^*$ has to take the other schedule $\mathcal{S}_1$ as well as its alternative $\mathcal{S}_1^*$ into account. As shown in Figure 7.8(b), a collision avoidance is performed for the two trajectories $a \to b$ and $a \to d \to b$ of r_1.

If e shall also be assigned to r_1 and shall be executed after a, but before b, this would require to compute two additional path alternatives as shown in Figure 7.8(c): One for $\mathcal{S}_1$ denoted by $\mathcal{S}_1^{**}$ and one for $\mathcal{S}_1^*$ denoted by $\mathcal{S}_1^{***}$:

$$\begin{aligned}\mathcal{S}_1 &= \{a \to b\}\\ \mathcal{S}_1^* &= \{a \to d \to b\}\\ \mathcal{S}_1^{**} &= \{a \to e \to b\}\\ \mathcal{S}_1^{***} &= \{a \to d \to e \to b\}\end{aligned}$$

The amount of path alternatives grows exponentially in the number of new jobs that arrive during the *uncertainty period* (2^x). In order to avoid that, the trajectory segment that is part of $\mathcal{S}^{Tx}$ is locked until a decision is made as depicted in Figure 7.8(d). If, finally, a commit or abort is issued, one of the path alternatives is removed as depicted in Figure 7.8(e). This also includes the removal of the trajectory lock.

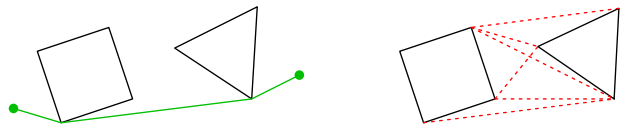
7.5 Trajectory Planning

The trajectory planning is based on the path-velocity-decomposition which has been presented in [40]. The resulting problem is a trajectory planning problem (TPP). The TPP is split into the static path planning problem (PPP) and the dynamic velocity planning problem (VPP) (cf. [40]). This approach is adjusted in order to check and guarantee schedulability of a job by calculating a collision-free spatio-temporal trajectory.

The objective of the trajectory planning is to move a robot from an initial point $(\vec{x}_I, t_I) \in \mathbb{R}^2 \times T$ to a target point $(\vec{x}_F, t_F) \in \mathbb{R}^2 \times T$ without collision with other obstacles O. Obstacles are represented as time dependent polygons that are implemented as time dependent point sets $O : T \to \mathcal{P}(\mathbb{R}^2)$. All obstacles are buffered according to the robots' geometry (in case of a circle the radius is chosen for buffering) while all robots are shrunk and represented as time dependent points $\vec{x}_\pi : T \to \mathbb{R}^2$.

The spatial path planning computes a continuous path $\pi \subset \mathbb{R}^2$ that starts at $\vec{x}_I$ and ends at $\vec{x}_F$, where π is the spatial trajectory that the robot has to follow; π is represented as point set.

The temporal path planning computes a temporal trajectory that states at what point in time the robot has to move along the segments of the spatial trajectory. Therefore, π is parametrized in dependence of the radian measure s of the path: $\vec{x}_\pi(s)$. The location



(a) Shortest path. (b) Lines of sight.

Figure 7.9: Spatial path planning.

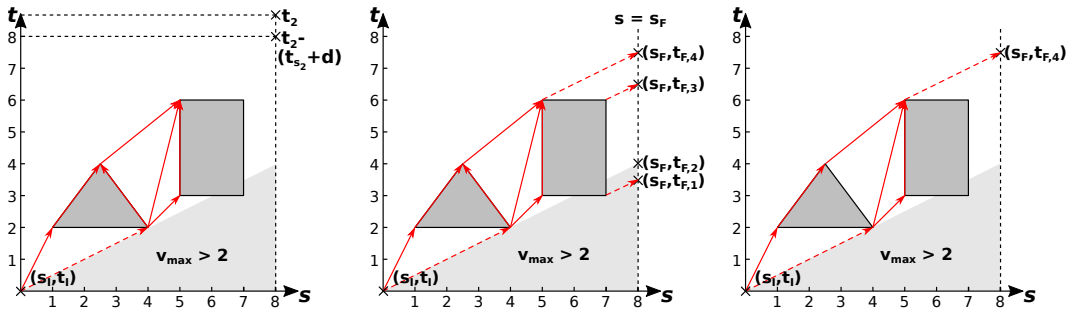
function $\vec{x}_\pi : S \to \vec{x}$ maps s to a point in (x, y)-space that is on the path π. The task is to find a time-based mapping $s : T \to S$ for s with $S = [s_I, s_F]$ being the radian interval of the path π. The spatio-temporal trajectory is then defined by $\vec{x}_\pi(t) = (\vec{x}_\pi \circ s)(t)$. The goal of the temporal planning is to find a mapping $\vec{x}_\pi : T \to \pi$ between time t and the points of the path π such that collisions with dynamic obstacles are avoided.

7.5.1 Spatial Path Planning

The PPP targets the problem of finding the shortest path between two points. Time is neglected in this step. The result of the spatial path planning is the path π that connects $\vec{x}_I$ and $\vec{x}_F$ by avoiding collisions with static obstacles O^s. In 2D space, the shortest path between two points which does not intersect any polygons, is a composition of segments, which connect a subset of vertices of the obstacles. As depicted in Figure 7.9(a), the path is along the vertices of the obstacles. In order to find the shortest path, a weighted graph is constructed that contains $\vec{x}_I$, $\vec{x}_F$ and the vertices V of the obstacles. The set of edges is defined by all pairs of vertices that are *visible* to each other. Vertex v_1 is visible by vertex v_2 if the line segment $[v_1 v_2]$ does not intersect with an obstacle (Figure 7.9(b)). The weight of the edge is defined by the Euclidean distance. The shortest path in the graph that connects $\vec{x}_I$ and $\vec{x}_F$ is the path π.

7.5.2 Temporal Path Planning

After π is calculated, the temporal path planning addresses the VPP and is, thus, responsible for computing a temporal path in order not to collide with dynamic obstacles O^m. Dynamic obstacles are represented as polygons that move over time. In order to calculate the velocity profile, the concept of forbidden regions is used—spatio-temporal regions in $s \times t$-space which are occupied by a dynamic obstacle. A forbidden region is always associated with a particular $x \times y$-path π and defines at which time intervals π is blocked. The location function $\vec{x}_\pi(s)$ uses s as arc length parameter. A computation of a velocity profile $s : T \to S$ that has no intersection with forbidden regions in $s \times t$-space avoids collisions with all dynamic obstacles by adjusting the velocity. The result of the temporal planning is a new path $\sigma = \{(s(t), t) | t \in T)\}$ that is computed similar to the PPP. The computation of σ has to take the following two conditions into account:



(a) Graph construction. (b) Finishing $s \times t$ points. (c) Pruning of the graph.

Figure 7.10: Temporal path planning with $v_{max} = 2$ represented by dashed lines.

$$t_i < t_j, with\ i < j \tag{7.9}$$

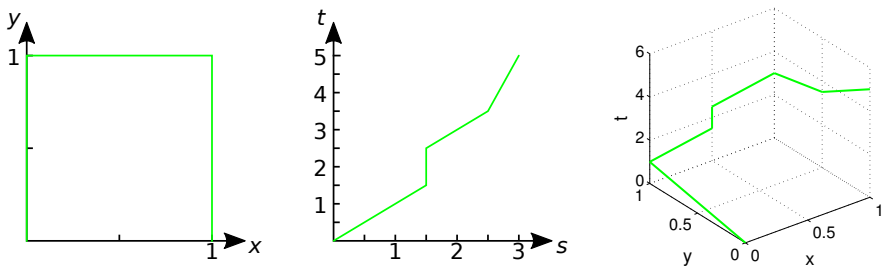
According to physical laws, time progresses constantly and, thus, the robot is not able to move back in time. Second, the robot has a maximum velocity which it can not exceed; (s_i, t_i) is a vertex of the graph that the robot visits before moving on to (s_j, t_j).

$$\left| \frac{s_j - s_i}{t_j - t_i} \right| \leq v_{max} \tag{7.10}$$

Similar to the PPP, a graph is constructed, as shown in Figure 7.10(a), that connects the vertices of the forbidden regions in $s \times t$ space. All vertices of the graph that are *visible* to each other are interconnected by edges. Each edge has a weight that indicates the time required to move from one endpoint of the edge to the other endpoint. There are upper bounds in time for calculating the temporal path since a slot has a given length. When calculating the temporal path for the path from $\vec{x}_{a_1}$ to $\vec{x}_{a_3}(s_1)$, then the upper time boundary is set to $t_2 - (t_{s_2} + d)$ with t_{s_2} being the time for moving back from $\vec{x}_{a_3}$ to $\vec{x}_{a_2}(s_2)$. However, when computing the temporal path for the path represented by s_2, then the time boundary is set to t_2 which indicates the end of the free slot. If the computed temporal path exceeds the boundary, then it is not possible to find a valid temporal path for the spatial path π that avoids collisions with dynamic obstacles and satisfies the spatio-temporal constraints of the action.

For the graph construction in Figure 7.10(a), a maximum velocity of $v_{max} = 2$ is assumed. All edges that indicate v_{max} are dashed. In order to compute a temporal path that states the earliest arrival time at a given destination, additional vertices have to be added to the graph: The first vertex (s_I, t_I) represents the initial situation and is set to $(0, 0)$. Next, up to multiple vertices $s_F, t_{F,*}$, representing final situations, are added.

The path is chosen that has the earliest arrival time, i.e., the minimum of all $t_{F,*}$. The vertices $(s_F, t_{F,*})$ are on the line $s = s_F$ as depicted in Figure 7.10(b). A vertex on the line $s = s_F$ is called final vertex $(s_F, t_{F,*})$ and is included in the graph if it is possible to connect a vertex of the graph with $s_F, t_{F,*}$ by a v_{max} edge. All pairs of vertices that are connected by an edge must be *visible* to each other. Hence, three additional



(a) $x \times y$ component π (b) $s \times t$ component σ (c) $x \times y \times t$ trajectory τ

Figure 7.11: Example: trajectory planning.

vertices $(s_F, t_{F,1})$, $(s_F, t_{F,3})$ and $(s_F, t_{F,4})$ have been included. The vertex $(s_F, t_{F,2})$ would be considered the optimal vertex since it would be reachable in the minimum possible amount of time (robot moves constantly with v_{max}).

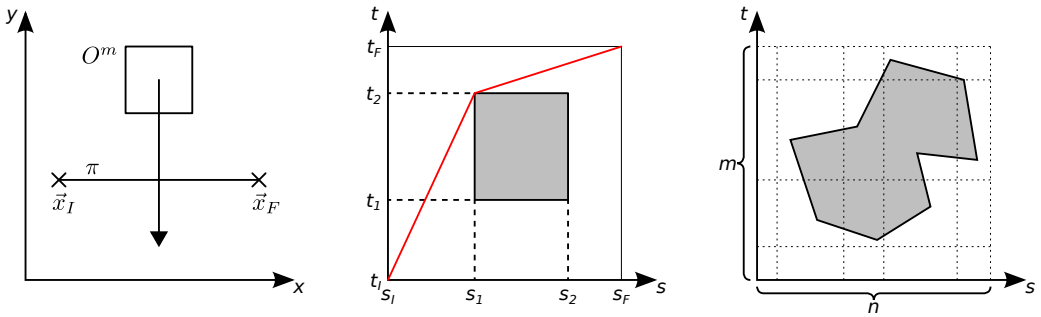
However, since the second dynamic obstacle crosses the path π, the vertex $(s_F, t_{F,2})$ is not reachable and, thus, has to be discarded. The vertex $(s_F, t_{F,1})$ is not reachable since this would require a velocity of $v_{max} > 2$. Finally, $(s_F, t_{F,3})$ is not reachable since the second dynamic obstacle blocks the path π. The vertices $(5, 3)$ and $(5, 6)$ are reachable. However, the vertices $(7, 3)$ and $(7, 6)$ are not reachable since this would require to progress on the path π in zero time which is not possible. Therefore, the only reachable vertex is $(s_F, t_{F,4})$.

As depicted in Figure 7.10(b), the resulting graph is a directed, weighted and potentially disconnected one. In order to find the shortest path, all sub-graphs of the graph that are not reachable by the initial vertex (s_I, t_I) are removed. The resulting pruned graph is shown in Figure 7.10(c). In order to determine σ, the shortest path in the pruned graph that connects the initial vertex (s_I, t_I) with one of the final vertices (s_F, t_F) is computed. This path results in the earliest arrival time of the robot at the destination. Using the spatial path π and the temporal path σ, the spatio-temporal trajectory $\tau = \{(x, y, t) | t \in T \wedge \vec{x}_\pi(t) = (x, y)\}$ in $x \times y \times t$ space is computed.

Figure 7.11 shows the computation of the different trajectory components: First, Figure 7.11(a) shows the computation of the spatial path π in $x \times y$ space. The path starts in $(0, 0)$ and ends in $(1, 0)$. The detour over $(0, 1)$ and $(1, 1)$ is due to an obstacle that blocks the direct path. The visualization of obstacles is neglected here. After π is computed, the velocity profile is adjusted by computing the temporal path σ in $s \times t$ space as depicted in Figure 7.11(b). Finally, the resulting trajectory τ in $x \times y \times t$ space is composed as shown in Figure 7.11(c).

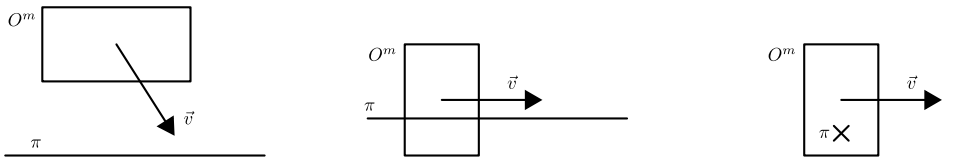
7.5.3 Forbidden Regions

In order to compute the temporal path σ, forbidden regions have to be computed first. Forbidden regions are caused by dynamic obstacles O^m that cross the spatial path π as depicted in Figure 7.12(a). The crossing of π by O^m causes a spatio-temporal blocking of



(a) O^m crosses path π (b) Forbidden region (c) $n \times m$ tiles

Figure 7.12: Computing forbidden regions.



(a) Regular case. (b) Parallel case. (c) Stationary case.

Figure 7.13: Cases for computing forbidden regions.

π that is represented by the forbidden region as shown in Figure 7.12(b). The blocking occurs in the interval given by $[s_1, s_2]$ and $[t_1, t_2]$. The solid line shows the temporal path σ. A forbidden region is iteratively computed for each dynamic obstacle and finally combined. Similar to the robot, each dynamic obstacle also follows a trajectory that consists of linear segments. The computation of the region is done segmentally. Assuming n spatial path segments of the robot and m trajectory segments of the obstacle creates $n \times m$ tiles as depicted in Figure 7.12(c) where each tile shows a part of the entire region.

Let $\vec{x}_i^r := (x_i^r, y_i^r)^T$ be the i-th vertex of the robot path π and (x_j^m, y_j^m, t_j^m) be the j-th vertex of the obstacle's trajectory. The i-th robot segment defines the radian interval: $S_i := [s_i^r, s_{i+1}^r] \ni s$. The j-th obstacle segment defines the time interval: $T_j := [t_j^m, t_{j+1}^m] \ni t$. Then $\vec{v}_j^m := \frac{(\Delta x_j^m, \Delta y_j^m)^T}{\Delta t_j^m}$ is defined as the velocity of the obstacle during the j-th trajectory segment[5].

In order to compute a tile of the forbidden regions, the following three cases as depicted in Figure 7.13 have to be distinguished:

- In the regular case, neither $\Delta\vec{x}_i^r$ nor $\Delta\vec{v}_j$ are zero and they are linear independent: $\det\begin{bmatrix}\Delta\vec{x}_i^r & \Delta\vec{v}_j\end{bmatrix} \neq \vec{0}$ as shown in Figure 7.13(a).

[5] Δ is defined as the difference of two adjacent values: $\Delta(\cdot)_i := (\cdot)_{i+1} - (\cdot)_i$

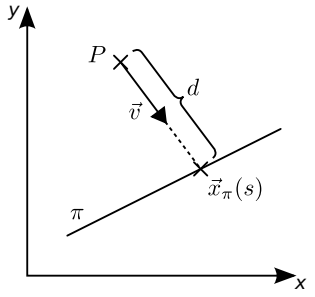


Figure 7.14: Extended velocity vector $\vec{v}$ intersects π.

- In the parallel case, the velocity vector $\vec{v}_j^m$ is parallel to the segment $\vec{x}_i^r$: $\det\left[\Delta\vec{x}_i^r \quad \Delta\vec{v}_j\right] = \vec{0} \ \wedge \vec{x}_i^r \neq \vec{0}$ as shown in Figure 7.13(b).
- In the stationary case, the robot segment has no length: $\vec{x}_i^R = \vec{0}$ as shown in Figure 7.13(c).

In the following, the regular case is described since it is the most common case. In the regular case, the forbidden region is created by transforming the vertices of the polygon $O^m(t_j^m)$. As depicted in Figure 7.14, one obstacle point P is considered which has the velocity $\vec{v}$. There will be an intersection of the extended velocity vector and the path π at a certain point (in space) s and at a certain point in time t. The value of t is computed by dividing the distance d by the absolute velocity $\vec{v}$.

Alternatively, the segment and the negative velocity vector can be considered as a base of the $s \times t$-space as depicted in Figure 7.15(a). Using this base, (s,t)-coordinates can be transformed to (x,y) coordinates:

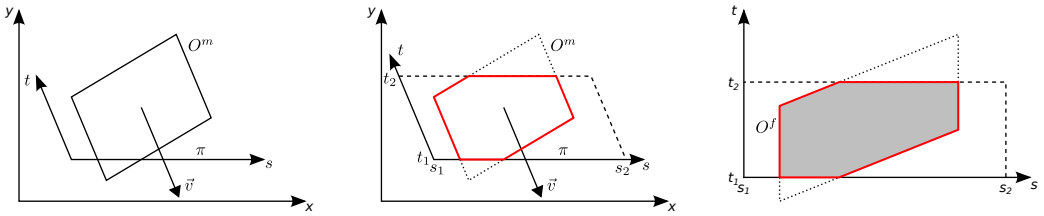
$$\begin{bmatrix} x - x_i^r \\ y - y_i^r \end{bmatrix} = (s - s_i^r) \cdot \vec{e}_s - (t - t_j^m) \cdot \vec{v}_j^m \tag{7.11}$$

$\vec{e}_s := \frac{\Delta\vec{x}_i^r}{||\Delta\vec{x}_i^r||}$ is defined as the unit vector of the robot segment. This equation can be transformed to

$$\begin{bmatrix} x - x_i^r \\ y - y_i^r \end{bmatrix} = \left[\vec{e}_s - \vec{v}_j^m\right] \begin{bmatrix} s - s_i^r \\ t - t_j^m \end{bmatrix} \tag{7.12}$$

in order to transform polygons to the $s \times t$-space. Since the polygon does not always entirely cross the robot segment as shown in Figure 7.15(b), it has to be cropped (Figure 7.15(c)). The crop boundary is defined by the robot segment S_i and the time interval T_j. The crop boundary sets the geometry for the current tile.

Figures 7.16 and 7.17 show an example of computing forbidden regions with multiple segments: two robot segments and three trajectory segments of the obstacle O^m. Figure 7.16 shows the movement of the robot and the movement of the obstacle O^m. The



(a) $s \times t$-base. (b) $x \times y$-space. (c) $s \times t$-space.

Figure 7.15: Cropping forbidden region.

segments are considered separately: the $0th$ robot segment defines the radian interval $S_0 := [s_0, s_1]$ and the $1st$ robot segment defines the radian interval $S_1 := [s_1, s_2]$. The values are calculated as follows:

$$s_i := \sum_{k=0}^{i-1} ||\vec{x}^r_{k+1} - \vec{x}^r_k|| \tag{7.13}$$

According to Equation 7.13, the values for s_0, s_1 and s_2 are calculated as follows:

$$\begin{aligned} s_0 &= 0 \\ s_1 &= ||\vec{x}^r_1 - \vec{x}^r_0|| = \sqrt{72} \\ s_2 &= ||\vec{x}^r_1 - \vec{x}^r_0|| + ||\vec{x}^r_2 - \vec{x}^r_1|| = \sqrt{72} + 9. \end{aligned}$$

The three trajectory segments of the obstacle O^m define the time interval T_j: $T_0 := [t_0, t_3]$, $T_1 := [t_3, t_3 + t^*_5]$ and $T_2 := [t_3 + t^*_5, t_3 + t^*_5 + t^*_7]$.

The obstacle O^m starts its movement at time t_0. At time t_1 it intersects the path π in the (radian) interval $[s^0_x, s^0_y]$. The intersection can be calculated using the extended velocity vector of the first trajectory segment of O^m. At time t_2, the obstacle has fully crossed the path π. At time t_3, O^m has reached the end of its first segment. O^m continues directly moving along its second segment which crosses π again in the (radian) interval $[s^1_x, s^1_z]$. At time $t_3 + t^*_4$, the first intersection with π (in S_1) is at s^1_y. At time $t_3 + t^*_5$, the obstacle O^m has reached the end of its second segment and has not fully crossed π yet (it is positioned on π). The intersection with π is in the interval $[s^1_x, s^1_z]$. From this position, O^m continues directly (at time $t_3 + t^*_5$) by moving along the third segment which starts on π. At time $(t_3 + t^*_5 + t^*_6)$, O^m has fully left π and at time $(t_3 + t^*_5 + t^*_7)$, O^m has reached its final destination. The resulting two forbidden regions are shown in Figure 7.17. Since there are two robot and three trajectory (of O^m) segments, the $s \times t$ diagram has 2×3 tiles with respective crop boundaries.

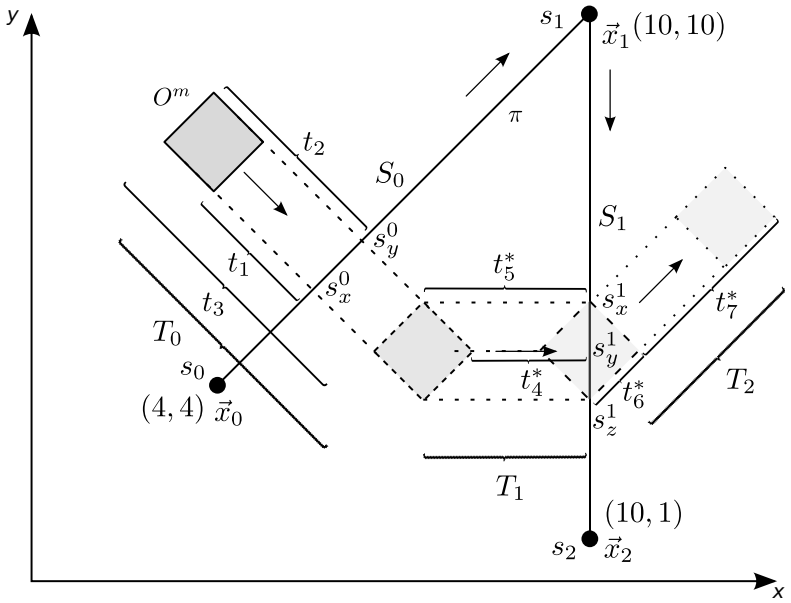


Figure 7.16: O^m has three trajectory segments (T_0, T_1 and T_2) and crosses the path π twice. The robot path has two segments (S_0 and S_1). The robot starts initially at location $\vec{x}_0 = (4,4)$ and proceeds towards $\vec{x}_2 = (10,1)$ by taking the detour over $\vec{x}_1 = (10,10)$.

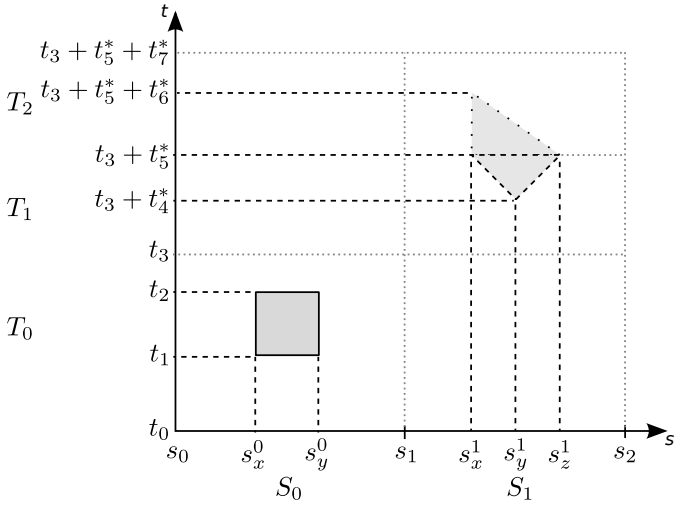


Figure 7.17: Space-time diagram with 2×3 tiles (two space segments and three time segments). The two crossings of O^m with path π causes two forbidden regions. Due to the different relative orientations of O^m to the robot, the shape of the resulting forbidden regions appear different.

7.5.4 Trajectory Planning

For the explanation of the trajectory planning, the example provided in Section 7.4.2 (page 104) is used. Algorithm 2 shows the trajectory planning of a single job. The algorithm takes the location *loc*, a free slot *slot* and the single action a as input from the job scheduler and computes (j^a, j^t). First, the algorithm computes two spatial paths π_1 and π_2 where π_1 is the path from $\vec{x}_{a_1}$ to $\vec{x}_{a_3}$ (approximated by the line segment s_1) and π_2 is the path from $\vec{x}_{a_3}$ to $\vec{x}_{a_2}$ (approximated by the line segment s_2) as shown in Figure 7.6(c).

Second, the temporal paths for π_1 and π_2 are calculated and composed to the trajectories τ_1 and τ_2, respectively. The computation of the temporal path σ_1 along the path π_1 and the composition to trajectory τ_1 are done by $calc\_trajectory(..)$. There are two $calc\_trajectory(..)$ functions with different method signatures which are explained in the following.

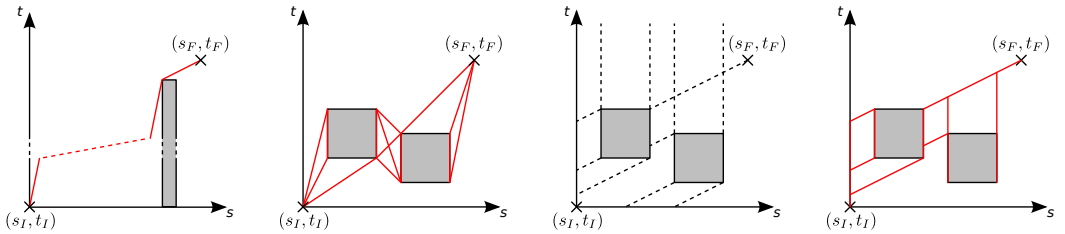
First, $calc\_trajectory(\pi_1, slot.t_1, a)$ computes σ_1 for π_1 and composes both to τ_1. The parameter $slot.t_1$ states the earliest possible start time (of the current free slot) (Figure 7.5) for σ_1 and, therefore, for τ_1. The third parameter is the job specification a which is required to obtain t_{min}, t_{max} and d. This is necessary in order to arrive in time at the respective location where the job shall be executed, i.e., not before t_{min} and not after $t_{max} - d$. After computing τ_1, the variable *job* is created which stores the job information: the *node* which executes the job, the execution location (*loc*), the execution start time ($\tau_1.t_{finish}$) which is the same point in time as the (timely) end of trajectory τ_1 and the duration d.

Second, $calc\_trajectory(\pi_2, job.t_{finish}, slot.t_2)$ computes σ_2 for π_2 and composes both to τ_2. The parameter $job.t_{finish} = \tau_1.t_{finish} + a.d$ states the earliest possible start time for σ_2 and, therefore, for τ_2. The third parameter $slot.t_2$ states the end of the free slot (Figure 7.5). Finally, $update\_trajectory(..)$ combines τ_1 and τ_2 to one trajectory τ.

Algorithm 2 Trajectory planning of a single job

Input: $loc, slot, a$
Output: $job = (p, t, r)$, $\tau = [(x_1, y_1, t_1), ..]$

```
function PLAN_JOB(loc, slot, a)
    node ← slot.node
    π1 ← calc_spatial_path(slot.x⃗start, loc)
    π2 ← calc_spatial_path(loc, slot.x⃗finish)
    τ1 ← calc_trajectory(π1, slot.t1, a)
    job ← (node, loc, τ1.tfinish, a.d)
    τ2 ← calc_trajectory((π2, job.tfinish, slot.t2))
    τ ← update_trajectory(node, τ1 ∪ τ2)
    return (job, τ)
end function
```



(a) Scenario 1. (b) Scenario 2. (c) Ray projection. (d) Graph construction.

Figure 7.18: Improvement of the original version by preferring high velocities and, thus, minimizing overall movement time. The result is a higher utilization since robots are able to perform other tasks while they wait for their next movement job.

7.5.5 Waiting Times

The trajectory planner chooses the earliest possible start time—that is the beginning of a free slot—as starting point for the temporal trajectory planning before continuing with the graph construction as described in Section 7.5.2. Dynamic obstacles that cross the path π of a robot may cause a delay of the arrival time. Figure 7.18(a) shows a scenario in which a dynamic obstacle with very low velocity crosses π. Analogously, this results in an very long blocking of π. The trajectory planner computes a velocity profile that indicates a linear correlation between the first robot segment that starts in s_I and the time that starts in t_I. The resulting profile is a constant movement with very low velocity. Figure 7.18(b) shows a scenario with two obstacles that also have slow velocities. There are three main issues when assigning very low velocities to a robot:

The first problem is that not all arbitrary low velocities are physically possible. Second, assigning a very low velocity to a robot creates exactly the same problem for other robots since the currently observed robot appears as a dynamic obstacle to others and blocks their path for a longer time. For explaining the third problem, the situation that has been shown in Figure 7.5 is considered: A new job a_3 shall be scheduled. If the conditions given in Equations 7.1 - 7.3 are satisfied, the movement job $a_1 \rightarrow a_2$ is discarded and is replaced with the three jobs $a_1 \rightarrow a_3$, a_3, $a_3 \rightarrow a_2$ which show the detour over a_3. Remembering the frozen horizon which was explained on the basis of Figure 7.3, the scheduler is only able to modify the schedule if the jobs are beyond the frozen horizon. In case a robot moves along path π with very low velocity, the following situation could occur: the new job (a_3) could be theoretically scheduled, but is now not schedulable since the movement job $a_1 \rightarrow a_2$ is already a part of the frozen horizon and, thus, $a_1 \rightarrow a_2$ is locked (Section 7.3.3). In this case, the robot has to follow the current trajectory segment. By minimizing the duration of the movement job by operating the robot with approximately maximum velocity, the probability that a new job will be rejected based on non-schedulability is decreased.

The advantage of scheduling the movement job as late as possible is that the probability that the robot is able to execute more tasks is higher then when scheduling movement

jobs as soon as possible. This is due to the fact that movement jobs can be arbitrarily replaced as long as they are not part of the frozen horizon and do not violate other committed action jobs. A robot that moves from $\vec{x}_{a_1}$ to $\vec{x}_{a_2}$ in order to execute a_2 tomorrow can be arbitrarily rerouted before reaching $\vec{x}_{a_2}$. A robot that already moved in the past to $\vec{x}_{a_2}$ in order to execute a_2 tomorrow can, indeed, also execute other jobs in-between, but already wasted energy by moving to $\vec{x}_{a_2}$. If, before executing a_2, new jobs shall be executed that are in physical proximity to $\vec{x}_{a_1}$, then the robot has to move back to the area of $\vec{x}_{a_1}$ and finally back to $\vec{x}_{a_2}$ which would be a large detour.

Figure 7.18(c) shows the ray projection phase that is necessary in order to perform the graph construction that prefers high velocities. The process is reversed by starting at the target point (s_F, t_F) and going backwards to (s_I, t_I). Rays are projected that indicate maximum velocity edges from the vertices that represent the forbidden regions backwards in negative t- and s-direction. The same is done for (s_F, t_F). In addition, vertical lines are projected in positive t-direction that indicate waiting times for the robot since they only make progress in t-direction, but not in s-direction. After the ray projection, the navigation graph is constructed by starting in (s_F, t_F) and interpreting the rays as edges and following them in order to reach (s_I, t_I). Since the construction is reversed, all edges have to go in negative t-direction. All edges that go in positive t-direction, those that are not reachable from (s_F, t_F) or those over which (s_I, t_I) can not be reached are pruned as depicted in Figure 7.18(d). Finally, in order to determine σ, the shortest path in the graph is computed that connects (s_I, t_I) and (s_F, t_F). The resulting trajectory shows that the robot waits as long as possible at its current location and then moves with maximum velocity to the next location.

Higher velocities indeed cause higher energy consumption. At this point, it is important to state that—although energy consumption is an issue for mobile robots—energy optimization is not addressed here.

7.6 Evaluation

The approach is evaluated by first performing a complexity analysis and, second, showing benchmark results.

7.6.1 Complexity Analysis

First, the trajectory planner is addressed and, thus, the TPP. Afterwards, the complexity of the job scheduler is analyzed. As this point, it is assumed to have no modifications of static obstacles over time. The graph construction in order to solve the PPP needs only to be done once and is not considered here. Let $\underline{O^s}$ be the number of vertices that represent static obstacles O^s and, thus, also the number of vertices in the graph. In order to compute the path π, Dijkstra's algorithm is applied. Depending on the actual implementation the algorithm has a polynomial (squared) or better (logarithmic) complexity. Here, squared complexity is assumed: $\underline{O^s}^2$. The path π of a robot consists of segments. Let $\underline{\pi}$ indicate the number of segments.

When computing forbidden regions, the robot path π and the trajectories of the dynamic obstacles O^m have to be taken into account. The complexity of O^m depends linear on the polygon size[6] of an obstacle, the number of trajectory segments of an obstacle and the total number of obstacles. Since there is a linear correlation, $\underline{O^m}$ is defined as product and, in particular, the number of 3-dimensional vertices that represent O^m:

Assuming that O^m consists only of one obstacle—a square—that has one trajectory segment, then $O^m_k(t)$ defines the k-th obstacle of O^m at time t. In this example, only $O^m_0(0)$ and $O^m_0(1)$ are constructed. Both point sets consist of four vertices. The total number of 3-dimensional vertices $\underline{O^m}$ is, thus, eight.

The worst-case complexity when computing forbidden regions is: $\mathcal{O}(\underline{\pi}\cdot\underline{O^m})$. However, since a check of envelope intersection is performed before computing a frame of the forbidden regions, in most cases, a decision can be made in advance if the current frame will contain a forbidden region. Although this does not change the complexity, this check significantly decreases the average computation time of calculating forbidden regions.

After the computation of the forbidden regions in $s \times t$ space, the navigation graph is constructed in order to compute the temporal path σ and finally compose π and σ together to the robot trajectory τ. Let $\underline{O^f}$ be the total number of vertices of the forbidden regions and, thus, also the number of vertices in the navigation graph. Constructing the navigation graph requires pair-wise iterating over the vertices of the forbidden regions which causes $\mathcal{O}(\underline{O^f}^2)$ complexity. Furthermore, visibility needs to be checked which causes linear complexity. The combined complexity is: $\mathcal{O}(\underline{O^f}^3)$. Afterwards, the shortest path is computed. For this Dijkstra's algorithm is applied in order to find the shortest path. Squared complexity is assumed again. The combined complexity for the TPP is:

$$\mathcal{O}(\underline{O^s}^2 + \underline{O^m} \cdot \underline{\pi} + \underline{O^f}^3) \tag{7.14}$$

At first glance, $\underline{O^f}^3$ seems to be a very high complexity, but this complexity holds only for vertices of the forbidden regions that are used in order to construct the navigation graph. However, a forbidden region occurs only if obstacles cross the very particular path of this one robot and at the same point in time. If no obstacles cross the path of the robot, then there are no forbidden regions and, thus, no navigation graph has to be computed. An important factor while operating a robot swarm is to be aware of the amount of robots in a given area or, in particular, the density of robots. A too high density increases the probability of potential collisions and as a consequence velocity profiles have to be adapted which cause navigation graph constructions of larger size.

Next, the complexity of the job scheduler is addressed: First, the job scheduler picks a location ($next\_location(..)$) from the specified geometry of the action specification ($a.g$). This requires to compute an area that is not covered by static obstacles. The complexity is $\mathcal{O}(L \cdot \underline{O^s})$, with L being the number of vertices that define the geometry g. Furthermore, the computation of an interior point of the geometry causes an additional

[6]With polygon size, the minimal number of vertices that describe the polygon is stated.

overhead of at maximum $L \cdot l$ which results in $\mathcal{O}(L \cdot \underline{O^s} + L \cdot l)$; with l being the number of location samples.

Next, $find\_slots(..)$ computes all free slots from all robots in the time window given by t_{min} and t_{max} and checks the conditions given by Equations 7.1 - 7.3 which causes $\mathcal{O}(|R| \cdot k)$ complexity, with $|R|$ being the number of robots and k the average amount of free slots per robot. This has to be done l times and, thus, the complexity is given by $\mathcal{O}(l \cdot |R| \cdot k)$. While checking the equations, $find\_slots(..)$ also computes Δs.

K is defined as the number of slots that are computed by $find\_slots(..)$—satisfying Equations 7.1 - 7.3—with $K \leq |R| \cdot k$. All Δs of all K slots are inserted in a red-black tree which causes $\mathcal{O}(\log(K))$ complexity.

In order to start the trajectory planning, $min\_detour(..)$ gets and removes the slot with the minimal detour from the red-black tree in $\mathcal{O}(\log(K))$. The current slot is discarded and $min\_detour(..)$ gets and removes the next slot with minimal detour, if no trajectory can be computed. This is repeated at most $l \cdot K$ times resulting in $\mathcal{O}(l \cdot K \cdot \log(K))$. Composing all parts together results in a total complexity of:

$$\mathcal{O}\Big(L \cdot \underline{O^s} + l \cdot L + l \cdot |R| \cdot k + l \cdot K \cdot \log(K) + l \cdot K \cdot \log(K) \cdot \Big[\underline{O^{s^2}} + \underline{O^m} \cdot \underline{\pi} + \underline{O^{f^3}}\Big]\Big) \quad (7.15)$$

Factoring out l results in

$$\mathcal{O}\Big(L \cdot \underline{O^s} + l \cdot \Big(L + |R| \cdot k + K \cdot \log(K) + K \cdot \log(K) \cdot \Big[\underline{O^{s^2}} + \underline{O^m} \cdot \underline{\pi} + \underline{O^{f^3}}\Big]\Big)\Big) \quad (7.16)$$

This can be transformed to

$$\mathcal{O}\Big(L \cdot \underline{O^s} + l \cdot \Big(L + |R| \cdot k + K \cdot \log(K) \cdot \Big[\underline{O^{s^2}} + \underline{O^m} \cdot \underline{\pi} + \underline{O^{f^3}} + 1\Big]\Big)\Big) \quad (7.17)$$

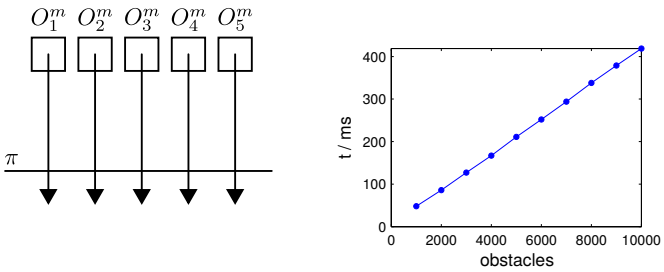
This shows that the trajectory planner has a larger complexity than the job scheduler which is due to the heuristic that is used in the job scheduler. Next, benchmarks are presented that also state that the job scheduler has only a small influence on the complexity.

7.6.2 Benchmarks

Several benchmarks of the space-time scheduler have been performed. For the simulations, a standard workstation with Intel Core i5 (Ivy Bridge) CPU with 3.2 GHz was used.

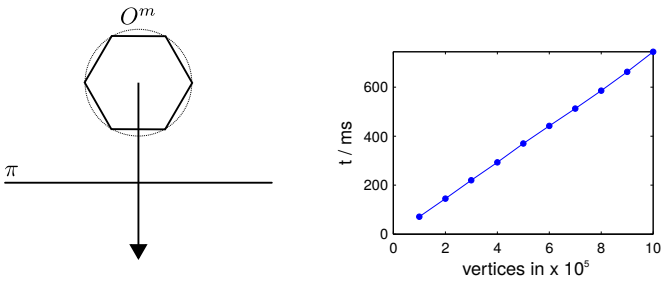
Figure 7.19(a) shows a scenario in which multiple dynamic obstacles are crossing the path π such that plenty of forbidden regions have to be computed. Figure 7.19(b) shows the linear correlation between the number of dynamic obstacles and the time for the computation of the collision-free trajectory τ. This was simulated with a maximum number of 10,000 dynamic obstacles. The computation took around 400 ms.

In Figure 7.20(a), only one dynamic obstacle was simulated that was crossing the path π. In this set-up the impact of the shape (number of vertices that describe the shape) on the computation time is observed. The number of vertices is iteratively increased,



(a) Initial setup. (b) Computation time.

Figure 7.19: Multiple obstacles crossing path π.



(a) Detailed shape. (b) Computation time.

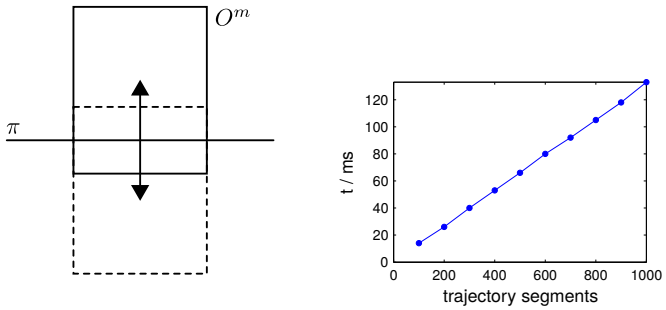
Figure 7.20: Impact of detail-level of obstacle on computation time.

i.e., the shape approximates a circle. Figure 7.20(b) shows the linear correlation between the number of vertices that represents the obstacle and the computation time of the scheduler. The most-detailed polygon was represented by 10^6 vertices. The computation took around 700 ms.

In the scenario depicted in Figure 7.21(a), one dynamic obstacle was created that was crossing the path π multiple times. Figure 7.21(b) shows the linear correlation between the number of trajectory segments of the obstacle (which is equal to the number of crossings) and the time for computing the schedule. This scenario was simulated with a maximum of 1,000 crossings. The computation took around 130 ms.

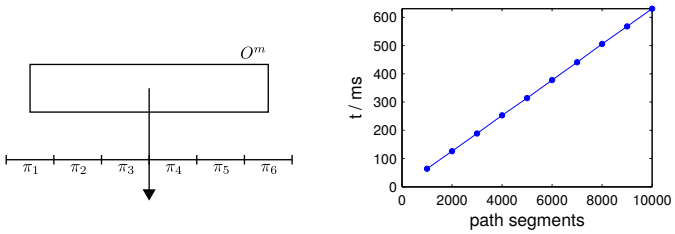
It was simulated that the robot path π consists of multiple segments π_i as shown in Figure 7.22(a). A large dynamic obstacle crosses all path segments. Figure 7.22(b) shows the linear correlation between the number of path segments and the time for computing the schedule. This has been simulated with a maximum number of 10,000 segments which took around 600 ms.

In Figure 7.23(a), four forbidden regions are simulated and successively approximated the shape of the regions to a circle. Figure 7.23(b) shows the influence of increasing the



(a) Multi-crossing. (b) Computation time.

Figure 7.21: Influence of multiple trajectory segments on computation time (obstacle crosses π multiple times).



(a) Multi-path. (b) Computation time.

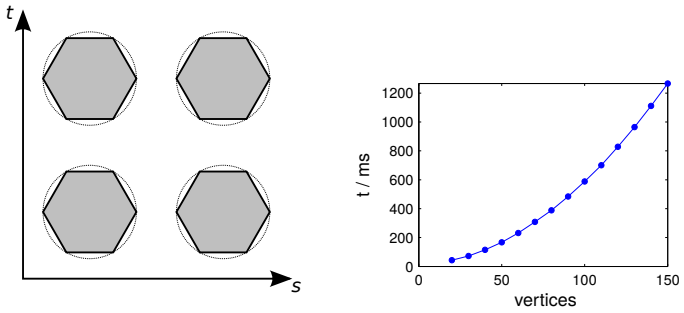
Figure 7.22: Influence of multiple path segments π_i that are intersected by a large dynamic obstacle on computation time.

number of polygon vertices on the computation time. The maximum number of polygon vertices per shape was 150 for each region. The computation took around 1200 ms. The benchmark makes clear that there is a disproportional large relation between both dimensions.

A scenario was created with n robots that have been arranged in a circle (Figure 7.24). Then an action was scheduled that should be executed in the center of the circle. Figure 7.25(a) shows the computation time as a function of the number of involved nodes. In this case, the first selected robot candidate was able to execute the job. The linear correlation is due to the fact that during the trajectory planning all other remaining robots are considered as dynamic obstacles and the planner iterates linearly over them. The same simulation was performed again, but placing an external dynamic obstacle in the center which did not move over time.[7]

Using a static obstacle instead, the job scheduler would have already been rejected the jobs since no location candidate could be found which would result in a rather fast

[7]Or did not move during the time period while the simulation was running. The intention of that behavior was chosen in order to examine the runtime impact of failing scheduling attempts since the target destination was blocked.



(a) Detail-level of forbidden regions.

(b) Impact on computation time.

Figure 7.23: Influence of the detail-level of forbidden regions (which is equivalent to the detail-level of the robot shape) on the computation time which shows a significant impact.

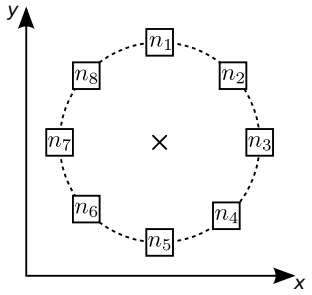
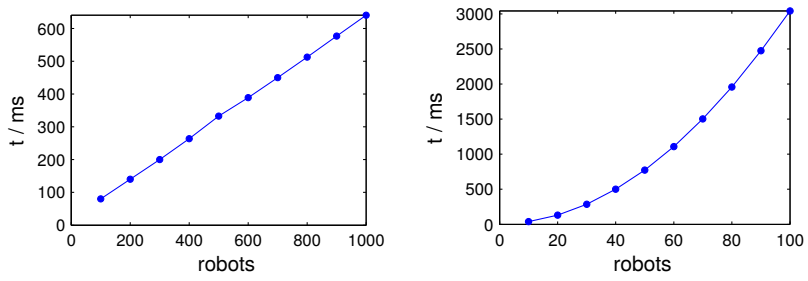


Figure 7.24: Circular arrangement of robots.

scheduling decision. However, this simulation should show the worst case performance, i.e., the scheduler has to iterate over all location places, over all robots and all free slots and compute spatio-temporal trajectories (performed by the trajectory planner) including the calculation of forbidden regions. The trajectory planner has the dominating complexity.

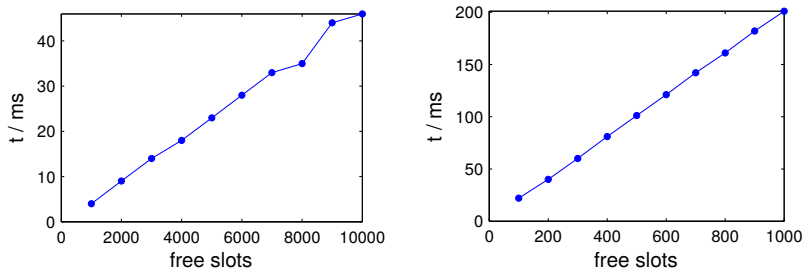
Each schedule attempt resulted in a fail since the external dynamic obstacle occupied the place the entire time. Since the scheduler attempted to schedule every robot and during each round all other robots have been considered as dynamic obstacles, the complexity is squared in the number of robots as depicted in Figure 7.25(b). Non-schedulable actions lead to higher computation times.

The scenario was modified again in order to measure the influence of free slots on the scheduling. Figure 7.26(a) shows the linear correlation between increasing the number of free slots and the time for computation. 10,000 was chosen as maximum number of free slots and took around 45 ms of computation time. In order to find the minimal detour, the job scheduler had to check all free slots. However, since this is done using a heuristic,



(a) First attempt was successful. (b) All attempts fail.

Figure 7.25: Influence of the number of nodes on the computation time while scheduling a new action using a circular arrangement setting.



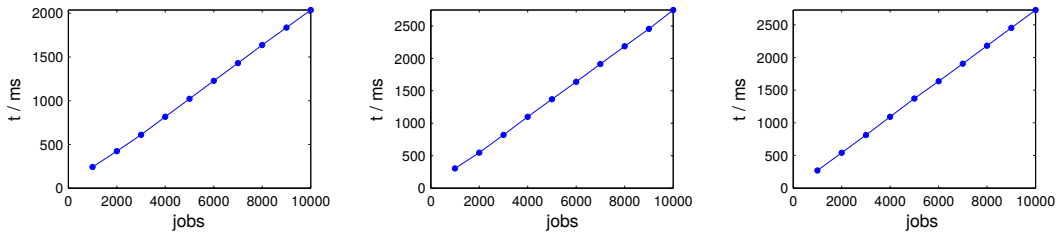
(a) First attempt was successful. (b) All attempts fail.

Figure 7.26: Influence of the number of free slots on the computation time while scheduling a new action using the same circular arrangement setting.

this computation performs quite fast. In contrast, the slower trajectory computation has only to be done once.

In the following, the scenario was modified again such that the scheduling of all free slots will fail. In this case, the scheduler iteratively determined the minimal detour and performed the entire trajectory planning with the new slot. Figure 7.26(b) shows the result that also states a linear correlation although the trajectory planning had to be repeated n times with n being the number of free slots. A decrease in the overall computation time could be measured which is still linear. This is due to the fact that a low detailed polygon shape is used which, again, has a strong influence on the runtime behavior of the scheduler.

Finally, a simulation was performed in which the correlation between the number of dependent jobs that shall be scheduled and the computation time was measured. In this simulation, only one robot was used and the number of jobs was iteratively increased by 1,000. The dependencies of the jobs were given by a singly linked list. Figure 7.27(a) shows the results which state a linear correlation. The computation time was 2 seconds for scheduling 10,000 jobs. Due to the linear correlation, the scheduling of one job took about 0.2 ms.



(a) Dependent jobs. (b) Periodic: same *loc*. (c) Periodic: different *loc*.

Figure 7.27: Scheduling of dependent / periodic jobs.

The same scenario was used in order to measure the computation time of the scheduler by scheduling periodic jobs as shown in Figure 7.27. Again, the number of periodic jobs was iteratively increased by 1,000. In Figure 7.27(b), the flag is set in order to use the same location for each instance of the periodic job. In Figure 7.27(c), a new location was computed for each instance of the periodic job. Comparing both results shows that there is no noticeable difference in the computation time.

According to the complexity and due to the observations of the benchmark it is important to keep the details of shapes of forbidden regions low. It is recommended to use low detail shapes (bounding boxes for instance). This holds for all geometries.

7.7 Conclusion

In this chapter, an online-scheduler was presented to schedule spatio-temporal actions in space and time. The algorithm follows a two-stage approach:

First, the job scheduler performs a coarse-grained planning that is based on a heuristic and neglects all obstacles. Decisions are made based on proximity between the robot and the action's execution location by using the Euclidean distance. The job scheduler proposes a free slot of a robot that is large enough in terms of time in order to move the robot to the designated execution area and execute the action without violating the spatio-temporal constraints. If necessary, the robot must be able to reach the execution location of the next action without violating the spatio-temporal constraints of that action.

Second, the trajectory planner takes the slot of the robot candidate as input and computes a collision-free spatio-temporal trajectory to avoid collisions with static and dynamic obstacles.

Due to the heuristic, the computation of the job scheduler is fast compared to the trajectory planner. As shown in the evaluation, the computation of the forbidden regions is expensive. A forbidden region will only appear if a dynamic obstacle crosses the path of the robot. In order to check if a collision is possible, a first check of envelope intersection of a robot segment and a trajectory segment of an obstacle is performed. If no envelope intersection is possible, then the entire computation of the forbidden regions can already be dropped for that particular segment. Another important factor that has a strong

impact on the computation time of the forbidden regions is the detail-level of the polygons that represent the obstacles. If possible, it is recommended to approximate obstacles by a rectangle (bounding box). In this case, only four vertices have to be projected on the robot's path along the extended velocity vector. Using coarse-grained rectangular approximations reduces overall computation overhead, but shrinks the solution space.

Chapter 8

Evaluation

This chapter presents the evaluation of this thesis. In the evaluation, different scenarios are constructed and their outcome is analyzed and explained.

8.1 Introduction

In this chapter the evaluation results are shown. The entire system is evaluated which is comprised of the following aspects:

- The *programming model* (Chapter 4)
- The *swarm runtime system* (Chapter 5)
- The *space-time scheduler* (Chapter 7)

All scenarios in the evaluation have been performed by programming applications consisting of suites and actions according to the programming model. The scheduler scheduled all these actions in space and time according to the spatio-temporal constraints and computed path alternatives. This chapter is split into three parts:

- A simulation (Section 8.2) which has evaluated the *space-time scheduler* and parts of the *swarm runtime system*.
- A hybrid approach (Section 8.3) which has evaluated the entire stack ranging from the *programming model* over the *swarm runtime system* to the *space-time scheduler* while movement itself is simulated. The runtime system performed the adapted two-phase commit protocol by communicating with the involved nodes using message passing and, finally, committed or aborted the (distributed) transaction and, hence, removed one of the path alternatives. The execution of the scheduled and committed applications have been performed distributed on several machines.
- An experimental part which has been performed on the testbed (Section 8.4) which has evaluated the entire stack ranging from the *programming model* over the *swarm*

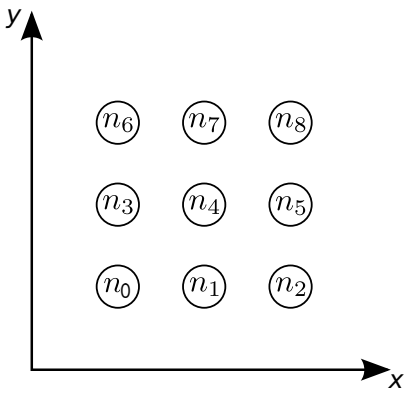


Figure 8.1: Grid-oriented, spatially uniform distributed arrangement of nodes.

runtime system and the *space-time scheduler* to localization (Appendix B) and motion control (Appendix C) of the robots (Appendix A). In addition to the evaluation presented in Section 8.3 (hybrid approach), the scenarios in this part include real robot movement on the testbed.

8.2 Simulation

This section presents the results of the simulations which have been performed by evaluating the *space-time scheduler* and parts of the *swarm runtime system*. In Section 8.2.1, virtual and physical movement are examined. The relation (fraction) between physical and virtual movement is shown as a function of the number of nodes. According to the approach presented in this thesis, the total physical movement can be strongly reduced by scaling up the number of nodes. This leads to less movement and, therefore, less resource consumption, e.g., allows more jobs to be scheduled. Energy consumption is not considered, but reducing the amount of physical movement, results in less energy consumption, in general. In Section 8.2.2, the system utilization (Section 5.4) is examined. The correlation between the job acceptance rate, the job load, the motion load and the combined system load are shown as a function of the number of jobs and the number of nodes. Furthermore, the ratio between schedulable jobs and required amount of nodes is presented. Concrete spatial units (mm, m, ..) are neglected in the simulation. The units are kept abstract.

8.2.1 Virtual Movement vs. Physical Movement

In this section, the ratio between virtual and physical movement is examined. In general, it is hard to predict the behavior of applications since the application's intention is unknown. A random appearance of actions is assumed.

The experiments[1] have been performed on a world with the dimensions $x = 100, y = 100$. Nodes are shaped as a circle with a diameter of 1 and a speed of $v_{max} = 1$. The experiments show the impact on virtual and physical movement as a function of the amount of nodes in the world. During the set-up, the nodes are arranged using a spatial uniform distribution as depicted in Figure 8.1.

One application is simulated that spawns a number of actions, each of which has a duration of $d = 1$. The spatial constraint of each action is randomly generated inside the world's borders. Every action is put into a separate suite. The length of the virtual movement is defined as

$$s_{virt} = \sum_{j=1}^{n-1} ||as_j.\vec{x}_{a_1} - as_{j+1}.\vec{x}_{a_1}|| \tag{8.1}$$

with $n = 10, 50, 80$. The length of the physical movement s_{phys} is defined as the sum over all robot trajectories.

All experiments have been performed 100 times. The generated plots show the following values as a function of the number of nodes:

- $avg(virt) := \sum_{i=1}^{100} s_{virt} \cdot 100^{-1}$
- $avg(phys) := \sum_{i=1}^{100} s_{phys} \cdot 100^{-1}$
- $avg(fraction) := \sum_{i=1}^{100} \frac{s_{phys}}{s_{virt}} \cdot 100^{-1}$
- $fraction$ shows the quartiles, outliers and the median value of the relation between physical movement (s_{phys}) and virtual movement (s_{virt}).

The experiments have been performed in two ways: *only space constraints* and with *space and time constraints*.

Figure 8.2 shows experiments with only space constraints. The first plot schedules 10 jobs as a function of the number of nodes. The second one 50 and the third one 80 jobs. The value of $avg(virt)$ scales approximately linear with the amount of actions ($n = 10, 50, 80$): if $n = 10$ then $avg(virt)$ is approximately 550. It is approximately 2750 (for $n = 50$) and 4400 (for $n = 80$). Therefore, each action "produces" a virtual movement of approximately 55 space units which indicates the average Euclidean distance between two adjacent actions. Since there are no time constraints, the scheduler assumes a fictive time window: $[0, \infty]$. The actions are scheduled as soon as possible which results in equal length of s_{virt} and s_{phys} if only one node is present, i.e., the robot physically moves to every $\vec{x}_{a_i}$ in the order the actions are scheduled. This results in a fraction of 1. Increasing the number of nodes results in a decreased physical movement as the actions are mapped such that s_{phys} is reduced. As a consequence, a decrease of s_{phys} results in a decrease of the fraction. Increasing the number of jobs results in a smaller variance as indicated by the box plots.

[1]The term "experiment" is used here in the context of a certain simulation set-up.

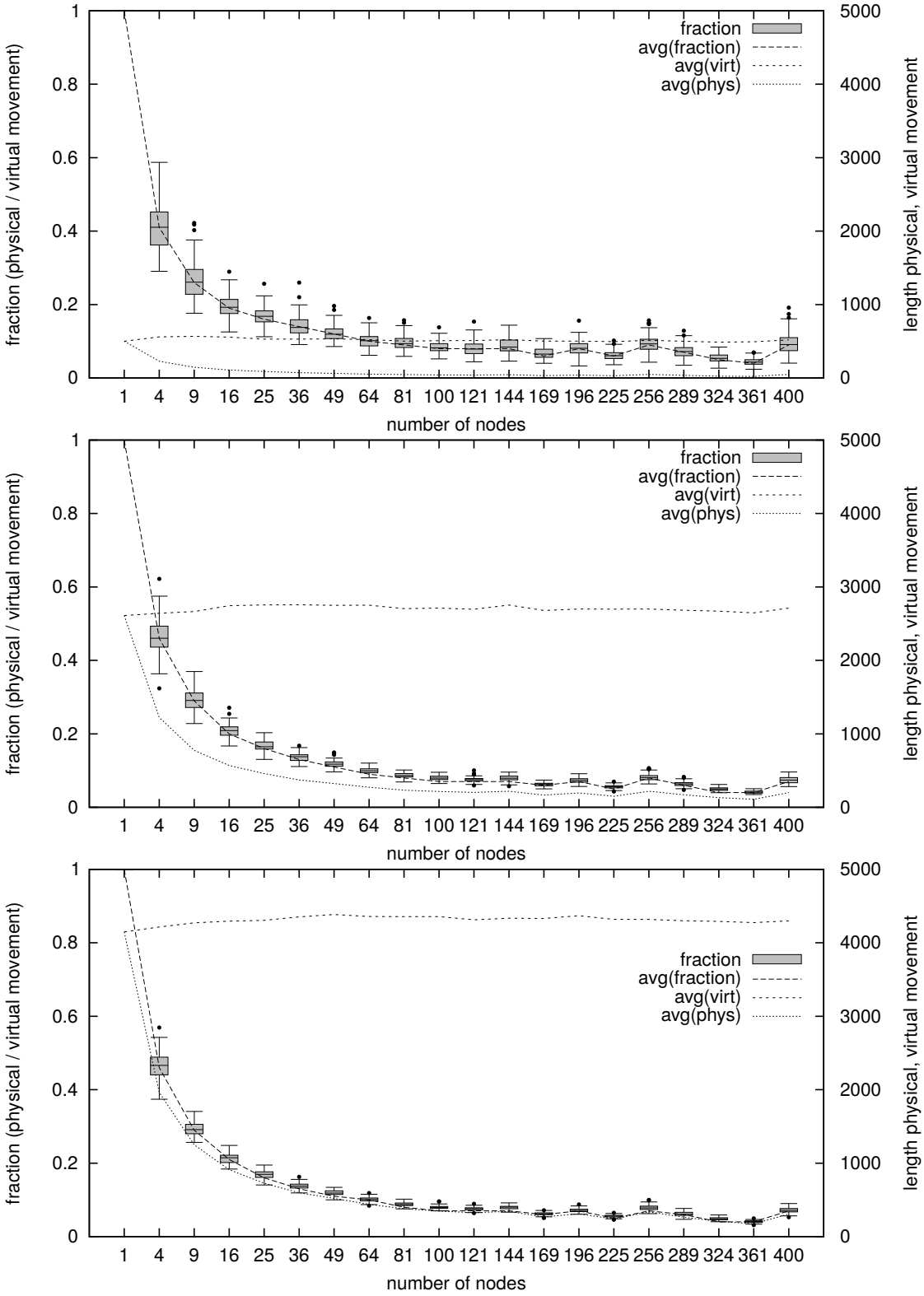


Figure 8.2: Simulation with 10 (50, 80) jobs per iteration, space constraints have been randomly generated in $x \in [0, 100]$, $y \in [0, 100]$ while assuming a fictive time window $t = [0, \infty]$.

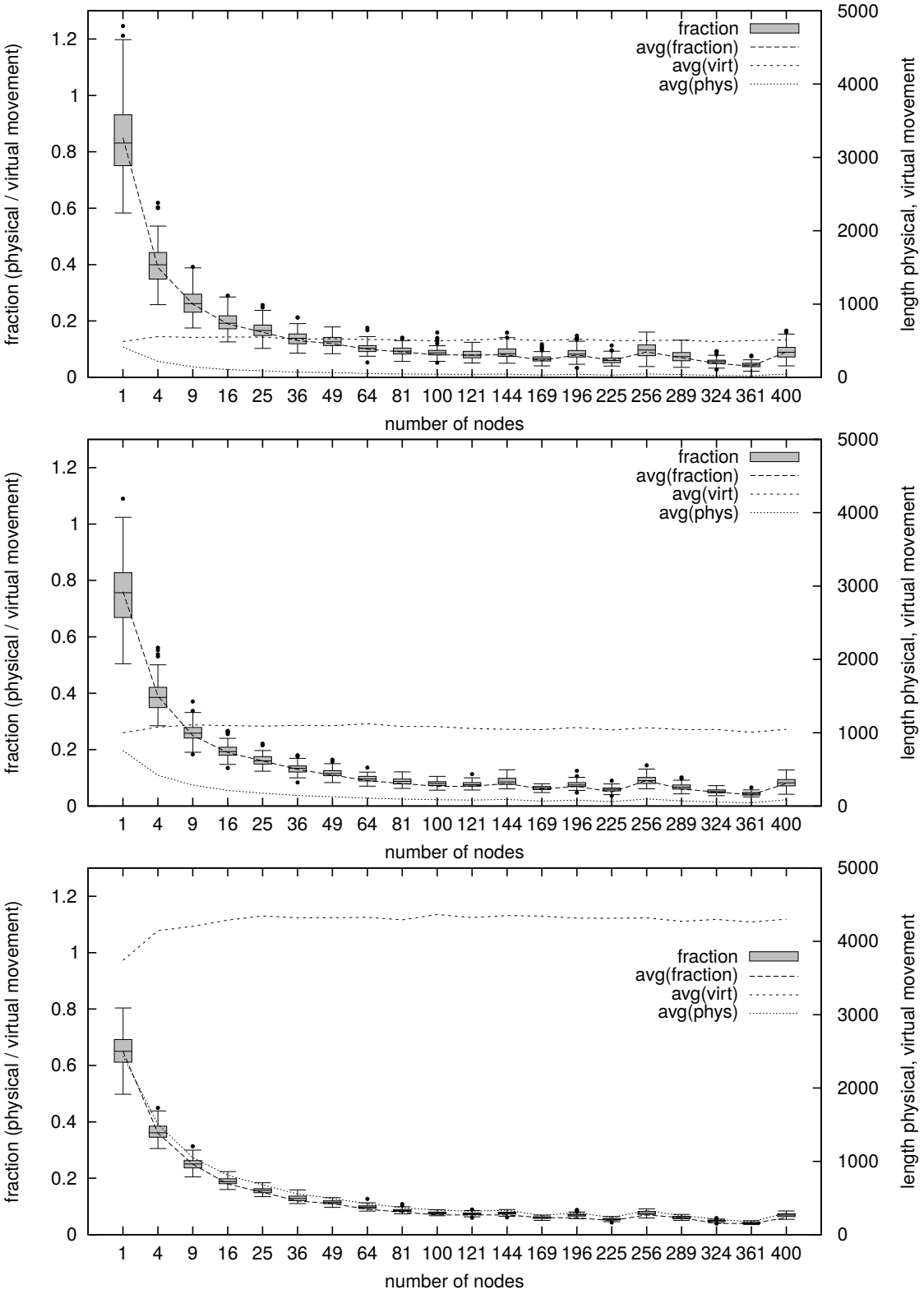


Figure 8.3: Simulation with 10 (20, 80) jobs per iteration, space constraints have been randomly generated in $x \in [0, 100]$, $y \in [0, 100]$; time constraints have been generated in $t \in [0, 3600]$.

Figure 8.3 shows the simulations with space and time constraints. The temporal constraints are randomly generated in the interval $t \in [0s, 3600s]$, i.e., t_{min}, t_{max} are both randomly generated. Therefore, also the length of the time window varies. In general, the behavior looks similar, but there are some details that are different: In the beginning (around up to 16 nodes) the fraction (*physical / virtual movement*) is smaller when scheduling actions only with space constraints. Using spatio-temporal constraints results in a larger fraction and, therefore, the amount of physical movement is larger. Using only 1 node, the fraction might be even larger than 1. Increasing the number of nodes (starting from approximately 25 nodes), the fraction and also the variance is even smaller using spatio-temporal constraints.

The behavior is explained in the following starting with the one node scenario: when scheduling a set of actions (a, b, c) using only space constraints in the order a, b, c and assuming that no dynamic obstacle "delays" the execution start of one of the actions as described in Section 8.3.1, the only possible schedule is (a, b, c) which always results in $s_{virt} = s_{phys}$.

Incorporating temporal constraints enables all possible permutations (depending on the constraint specification). If all temporal constraints are equal ($t^a_{min} = t^b_{min} = t^c_{min} \wedge t^a_{max} = t^b_{max} = t^c_{max}$) as shown in Section 8.3.1, and assuming again that no dynamic obstacle is present, the case can be mapped to the one where no temporal constraints are present. Then the only possible schedule is given by (a, b, c). If all temporal constraints are different, then any arbitrary permutation is possible: (a, b, c), (a, c, b), (b, a, c), (b, c, a), (c, a, b), (c, b, a). However, the length of the virtual movement s_{virt} is given as defined in Equation 8.1. The length of the movement is, thus, $s_{virt} := ||\vec{x}_a - \vec{x}_b|| + ||\vec{x}_b - \vec{x}_c||$ given the assumption that the *schedule* operation is invoked in the order (a, b, c). Depending on the temporal constraints, the computed schedule may look like this: a, c, b. In this case (still assuming one node), the node's trajectory is given by $a \to c \to b$ and $s_{phys} := ||\vec{x}_a - \vec{x}_c|| + ||\vec{x}_c - \vec{x}_b||$. Assigning the following execution locations to the actions $\vec{x}_a = (10\ 10)^T$, $\vec{x}_b = (20\ 20)^T$ and $\vec{x}_c = (30\ 30)^T$ results in the following length values: $s_{virt} := 2 \cdot \sqrt{200}$, $s_{phys} := \sqrt{800} + \sqrt{200}$ resulting in a fraction > 1.

However, scheduling a set of spatio-temporal actions will probably generate gaps in the schedule. These gaps can be filled with other actions that are "on the way". The more actions are scheduled, the more the probability increases that gaps are filled which result in an even better relation between s_{virt} and s_{phys}. In the following, it is assumed that the actions have been scheduled at the following times: $t^a = 0, t^b = 100, t^c = 50$.

Now, a second application is executed that performs two schedule requests with actions d and e. The actions are scheduled at $\vec{x}_d = (12\ 12)^T$, $\vec{x}_e = (28\ 28)^T$ at time $t^d = 25, t^e = 75$. The virtual movement is given by $s^*_{virt} := \sqrt{512}$ and the nodes trajectory is updated as follows: $a \to d \to b \to e \to c$. Since d and e are scheduled "on the way", the physical movement s_{phys} remains unchanged. The combined virtual movement is given as $s_{virt} + s^*_{virt} := 2 \cdot \sqrt{200} + \sqrt{512}$. The resulting fraction is now < 1.

8.2.2 System Utilization

In this section, the system utilization (Section 5.4) is examined. The following evaluation has been performed on the one hand to examine the system utilization as a function of the amount of jobs and on the other hand as a function of the amount of nodes. All nodes are set up as in the previous scenarios: maximum speed of $v_{max} = 1$, shaped as a circle with diameter of 1 using a spatially uniform distributed arrangement on a grid. The world's size is again set up with $x = 100, y = 100$. All actions have a duration of $d = 1$. The spatial constraints are randomly generated inside the world's border. The temporal constraints are either set to the interval of t_1 and t_2 which is given in the following or randomly generated in $[t_1, t_2]$. Actions are spatially uniformly distributed. The plots show the following values:

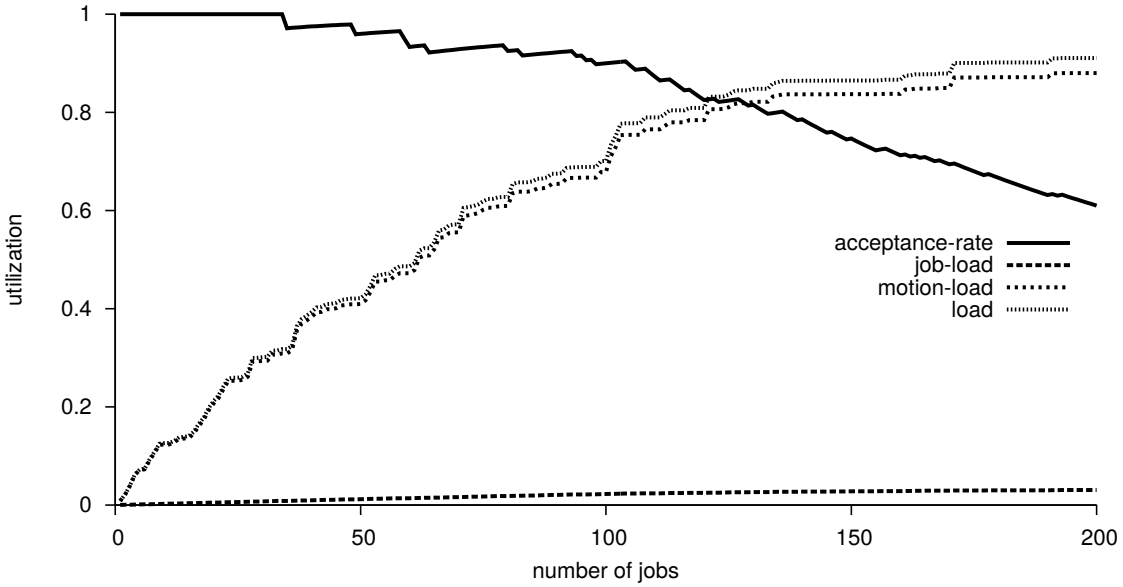
- *acceptance-rate*: the rate indicates the fraction of accepted jobs in relation to the total amount of jobs ($\frac{accepted}{accepted+rejected}$).
- *job-load*: $u^j(t_1, t_2)$ indicates the job load during that interval.
- *motion-load*: $u^m(t_1, t_2)$ indicates the motion load during that interval.
- *load*: $u(t_1, t_2)$ indicates the total load during that interval.

It is assumed that no time progresses during the simulation, i.e., the frozen horizon as well as the present time are kept static. This does not influence the result or behavior of the simulations since the simulations can be mapped to a scenario in which time progresses and the values of t_1 and t_2 are set to be sufficient far into the future and all generated jobs are also in the interval $[t_1, t_2]$. Doing so, it must be guaranteed that the simulations finish before the present time of the scheduler reaches t_1. Setting $t_1 = 0$ and allowing time to progress forces the scheduler to adjust t_1 by setting it to its internal present time reducing the interval length of t_1 and t_2 which leads to a modification of the system utilization values.

In scenario 1 (Figure 8.4), the system utilization is measured in the interval $t_1 = 0, t_2 = 4000s$ using only 1 node and iteratively increasing the number of jobs until 200, each of which has a duration of $d = 1$. The result is plotted in Figure 8.4(a) based on scenario 1(a) in which time constraints are kept static ($t_{min} = 0s, t_{max} = 4000s$). Two phases can be noticed: in phase 1 ranging from 0 - 74 jobs the acceptance rate is constantly 1 which states that all jobs have been successfully scheduled. As a result, all loads are strictly increasing. After scheduling the 74. job, the load u reaches approximately 1 and, therefore, the system is fully utilized. At this point in time all utilizations have reached their maximum. The job load is $u^j = \frac{74s}{4000s} = 0.0185$ while the motion load is $u^m \approx 0.9810$ resulting in a total utilization of $u \approx 0.9995$.

The second phase starts while job 75 is scheduled. From here on, the acceptance rate is strictly decreasing since the system is not able to accept new jobs. As a consequence, all utilization values maintain their values. It is important to add that the system is only fully utilized in the interval of $[0s, 4000s]$. Extending the interval to $[0s, 8000s]$ would cut all system utilization values into halves.

(a) Scenario 1(a): Uniformly distributed space constraints; time constraints are kept static (randomly generated in $x \in [0, 100]$, $y \in [0, 100]$; $t_{min} = 0s, t_{max} = 4000s$).



(b) Scenario 1(b): Space and time constraints follow a uniform distribution (randomly generated in $x \in [0, 100]$, $y \in [0, 100]$; t_{min} and $t_{max} \in [0s, 4000s]$).

Figure 8.4: Scenario 1: System utilization u, u^j, u^m in interval $[0s, 4000s]$ and acceptance rate with 200 jobs and 1 node.

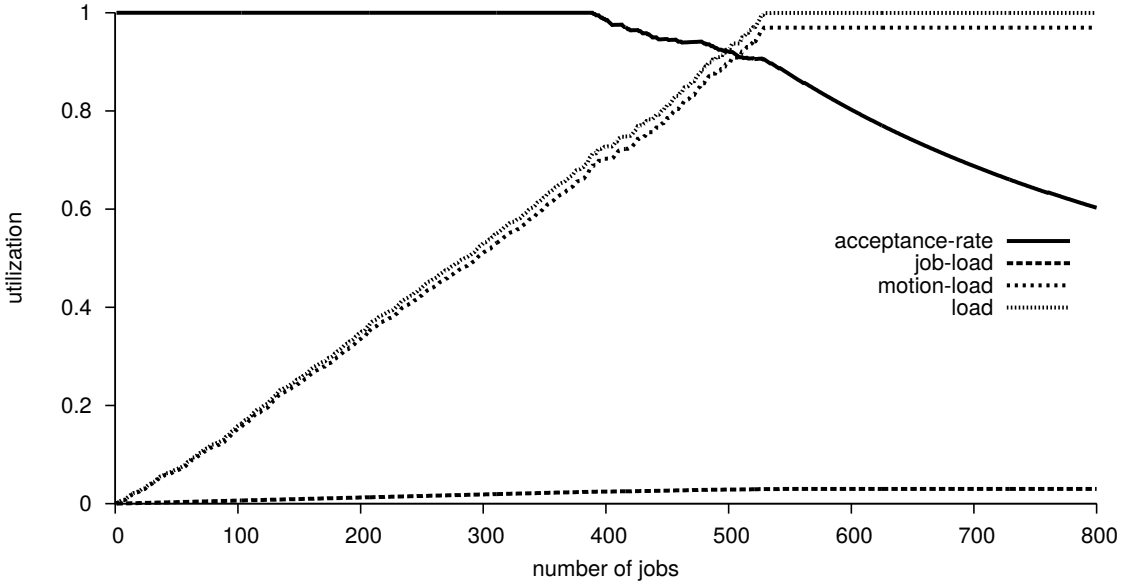
Scenario	scheduled jobs	acceptance rate	u	#nodes
scenario 1(a)	74 / 200	0.37	1	1
scenario 1(b)	122 / 200	0.61	0.91	1
scenario 2(a)	482 / 800	0.60	1	4
scenario 2(b)	733 / 800	0.92	0.83	4

Table 8.1: Simulation results.

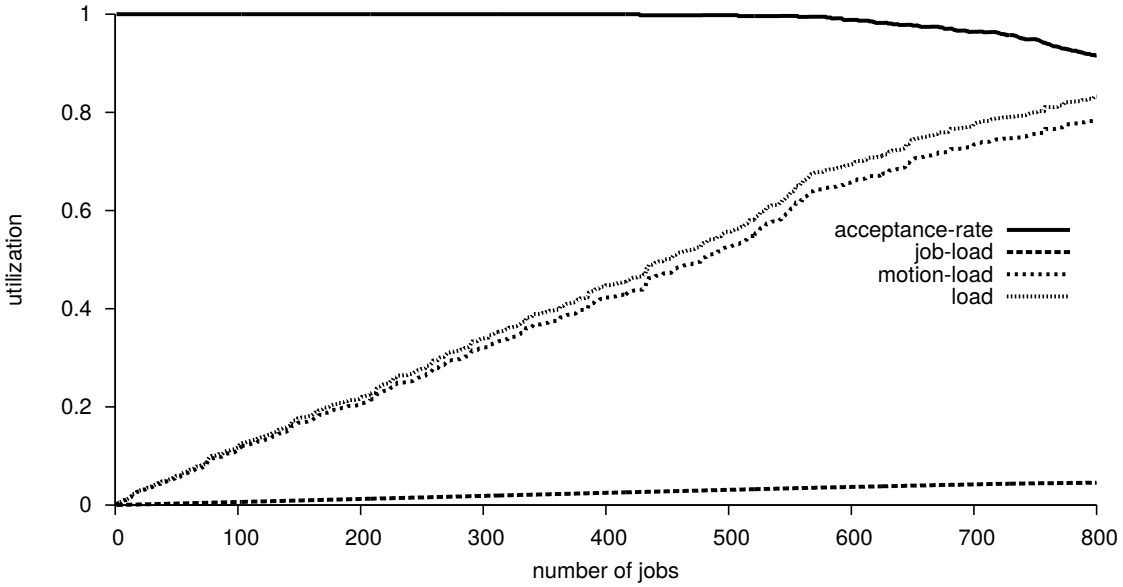
In scenario 1(b), the temporal constraints are randomly generated in $[0s, 4000s]$. The measured values are plotted in Figure 8.4(b). The acceptance rate already starts decreasing at the 35th job ($u^j \approx 0.0085, u^m \approx 0.3091, u \approx 0.3176$) while the system utilization increases less strong compared to Figure 8.4(a). Higher numbers of scheduled jobs correlate with higher utilization values which indicate that the system has less free capacity. Hence, the probability that a new job gets scheduled decreases with the number of already scheduled jobs. After the 200th job has been scheduled (last job according to scenario 1), the utilization is $u^j \approx 0.0305, u^m \approx 0.8799, u \approx 0.9104$. The system is yet not fully utilized and is able to accept more jobs. The acceptance rate is 0.61 which indicates that in total 122 jobs have been successfully scheduled. In comparison, the acceptance rate in scenario 1(a) (Figure 8.4(a)) is 0.37 resulting in 74 scheduled jobs. In scenario 1(b) shown in Figure 8.4(b), approximately 1.65 times more jobs could be scheduled. The rejection of some jobs may have different causes, e.g., since the constraints are generated, it is possible that multiple jobs have similar time constraints while the space constraints are set in different regions such that the Euclidean distance exceeds a certain threshold making it impossible to reach the next location using only a "one-node" set-up.

In the next scenario (Figure 8.5), the number of nodes is set to 4 and the amount of jobs is set to 800. The remaining parameters are maintained. In Figure 8.5(a) the measured values are presented in which the temporal constraints are set to: $t_{min} = 0s, t_{max} = 4000s$. In the beginning (until job 389), the system utilization values are strictly increasing while the acceptance rate constantly remains 1. Scheduling job 390 starts the decreasing of the acceptance rate. The utilization at this point is given by $u^j \approx 0.0242$, $u^m \approx 0.6885$ and $u \approx 0.7127$. Between job 390 and 542 the system utilization is constantly increasing while the acceptance rate tends to go down, i.e., though the system accepts more jobs, the probability of acceptance goes down. After scheduling job 542, the utilization is given as follows: $u^j \approx 0.0298$, $u^m \approx 0.9697$ and $u \approx 0.9995$, the acceptance rate is approximately 0.8870. After that point, the probability of accepting new jobs is approximately zero (acceptance rate decreases strongly). In the end, the values are given by: $u^j \approx 0.02995$, $u^m \approx 0.9697$ and $u \approx 0.9996$. The acceptance rate is approximately 0.6027 which states that 482 jobs have been scheduled.

Comparing scenario 1 and scenario 2 (see Table 8.1 for simulation results), it becomes obvious that quadrupling the amount of nodes from 1 to 4 does not simply lead to accept four times more jobs. In fact, it allows to accept even more since there is a non-linear correlation. In scenario 1(a), 74 jobs have been scheduled successfully. In scenario 2(a), 482 jobs have been scheduled. Comparing those numbers, the amount of jobs that have



(a) Scenario 2(a): Uniformly distributed space constraints; time constraints are kept static (randomly generated in $x \in [0, 100]$, $y \in [0, 100]$; $t_{min} = 0s, t_{max} = 4000s$).



(b) Scenario 2(b): Space and time constraints follow a uniform distribution (randomly generated in $x \in [0, 100]$, $y \in [0, 100]$; t_{min} and $t_{max} \in [0s, 4000s]$).

Figure 8.5: Scenario 2: System utilization u, u^j, u^m in interval $[0s, 4000s]$ and acceptance rate with 800 jobs and 4 nodes.

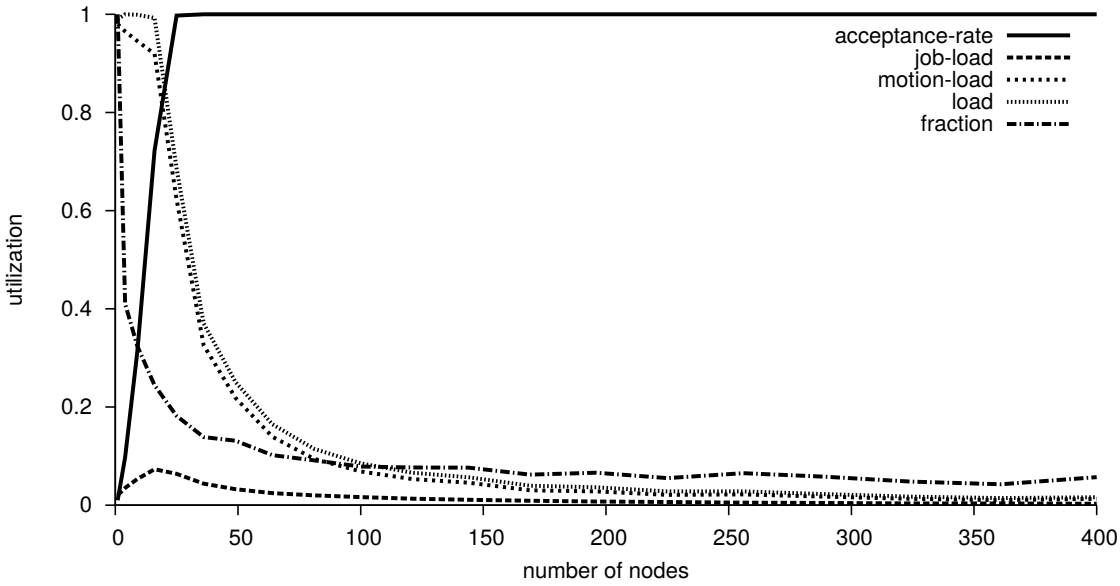


Figure 8.6: Scenario 3: System utilization u, u^j, u^m together with job acceptance-rate and fraction of reduced physical movement with 400 jobs as a function of the amount of nodes (1-400), using space and time constraints $(t_{min}, t_{max} \in [0s, 500s], g \in (x \in [0, 100], y \in [0, 100])$.

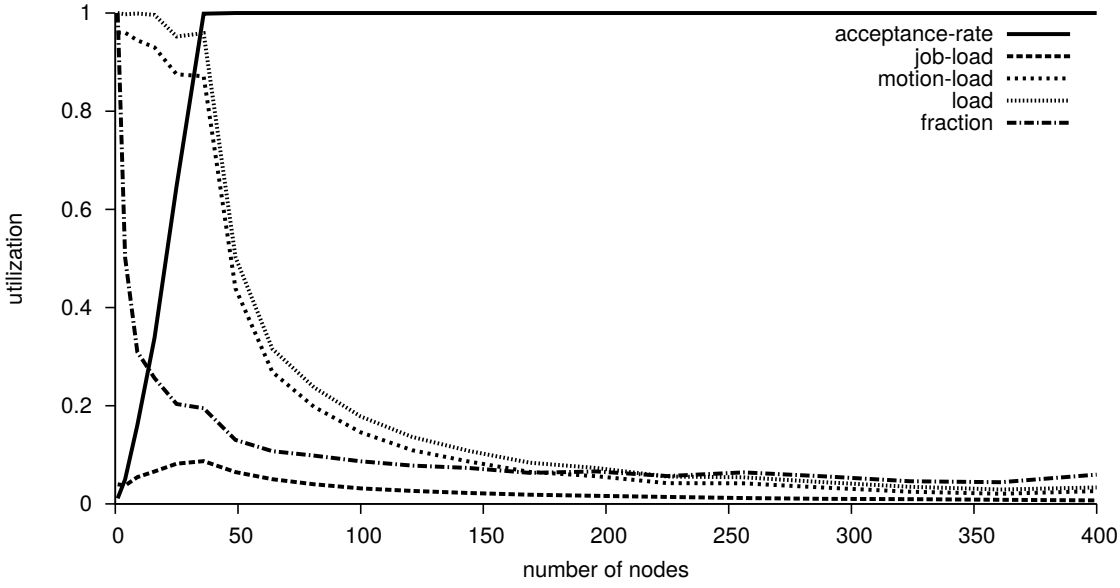


Figure 8.7: Scenario 4: System utilization u, u^j, u^m together with job acceptance-rate and fraction of reduced physical movement with 800 jobs as a function of the amount of nodes (1-400), using space and time constraints $(t_{min}, t_{max} \in [0s, 500s], g \in (x \in [0, 100], y \in [0, 100])$.

been scheduled is approximately 6.5 times higher using 4 times more nodes. In scenario 2(b), the temporal constraints are randomly generated in $[0s, 4000s]$ (Figure 8.5(b)). Here, the acceptance rate remains 1 until job 427 and then goes down slightly while the utilization increases. At this point the utilization is given by: $u^j \approx 0.02662$, $u^m \approx 0.4388$ and $u \approx 0.4654$. After job 800, the final values are: $u^j \approx 0.0456$, $u^m \approx 0.7863$ and $u \approx 0.8320$ while the acceptance rate is 0.9160 which states that 733 jobs have been scheduled. This is a gain of 1.52 times more jobs using scattered temporal constraints.

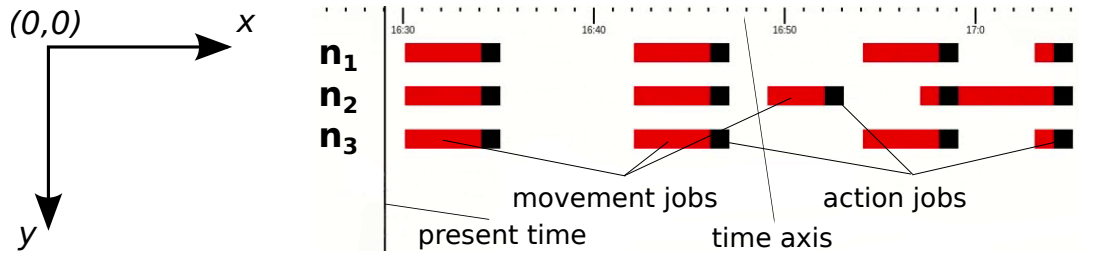
In the following two scenarios (scenario 3 and 4), the effect of increasing the number of nodes on the system utilizations is examined. For this scenario 2 is modified as follows: the number of nodes is iteratively increased starting with one up to 400 nodes. The amount of jobs is set to 400 (scenario 3). The temporal constraints are modified and randomly chosen in the interval $[0s, 500s]$. The system utilization is measured in the same interval. Figure 8.6 shows the plotted values. The plot shows that up to 16 nodes the system utilization u stays approximately constant at 1 while u^m together with the fraction $(\frac{s_{phys}}{s_{virt}})$ decreases and u_j together with the acceptance rate increases. Increasing the number of nodes leads to shorter node movement phases and, hence, more jobs can be scheduled. There is a correlation between u, u^m, u^j and the acceptance rate: until the acceptance rate is less than 1 and the overall load u is approximately 1 (system is fully utilized but does not have sufficient capacity to accept all jobs), u^j increases while u^m decreases when the number of nodes increases. This is due to shorter physical movement resulting in a smaller fraction, i.e., the efficiency increases. Until 49 nodes, there is a strong decrease in u, u^m as well as the fraction noticeable. Afterwards, an asymptotic behavior of u, u^m, u^j and the fraction regarding the node axis is observable. This means that given the assumption of this scenario, an amount of 25 nodes is sufficient in order to reach an acceptance rate of approximately 1.

In scenario 4, the set up is modified by increasing the number of jobs to 800. All other parameters remain unaffected. Figure 8.7 shows the results. The behavior of the values appear similar. In the beginning (up to around 25 nodes), the system utilization u, u^m are shifted to the right while u^j is stretched (larger increasing phase). The decreasing phase is in both scenarios similar. The maximum acceptance rate of 1 is reached with 36 nodes (doubling the number of jobs requires 1.5 times more nodes). Independently on the amount of jobs, is the behavior of the fraction curve.

In order to determine the suitable number of nodes, an estimation of the number of actions together with an assumption of the spatio-temporal constraints has to be performed. In particular, the frequency has to be analyzed (number of actions per time frame).

8.3 Hybrid Approach

This section presents different scenarios with and without obstacles and examines the system behavior. The experiments evaluate the system on a holistic level, excluding the actual movement. The movement is simulated by simply following the spatio-temporal trajectories as they have been computed by the scheduler. The runtime system performed



(a) World with space dimensions x and y.

(b) Visualization of the schedule of nodes n_1, n_2 and n_3 with movement and action jobs.

Figure 8.8: Explanation of world set-up and schedule visualization.

World set-up	Description
O_0^m	
	shape: $(-0.5, 0), (-0.5, 3), (0.5, 3), (0.5, 0), (-0.5, 0)$
	path: $(20, 1), (20, 5)$
	times: $2s, 45s$
n_1	
	shape: circle$((0, -0.5), (-0.5, 0), (0, 0.5), (0, 5, 0))$
	position: $(2, 4)$
	speed: $v_{max} = 1$

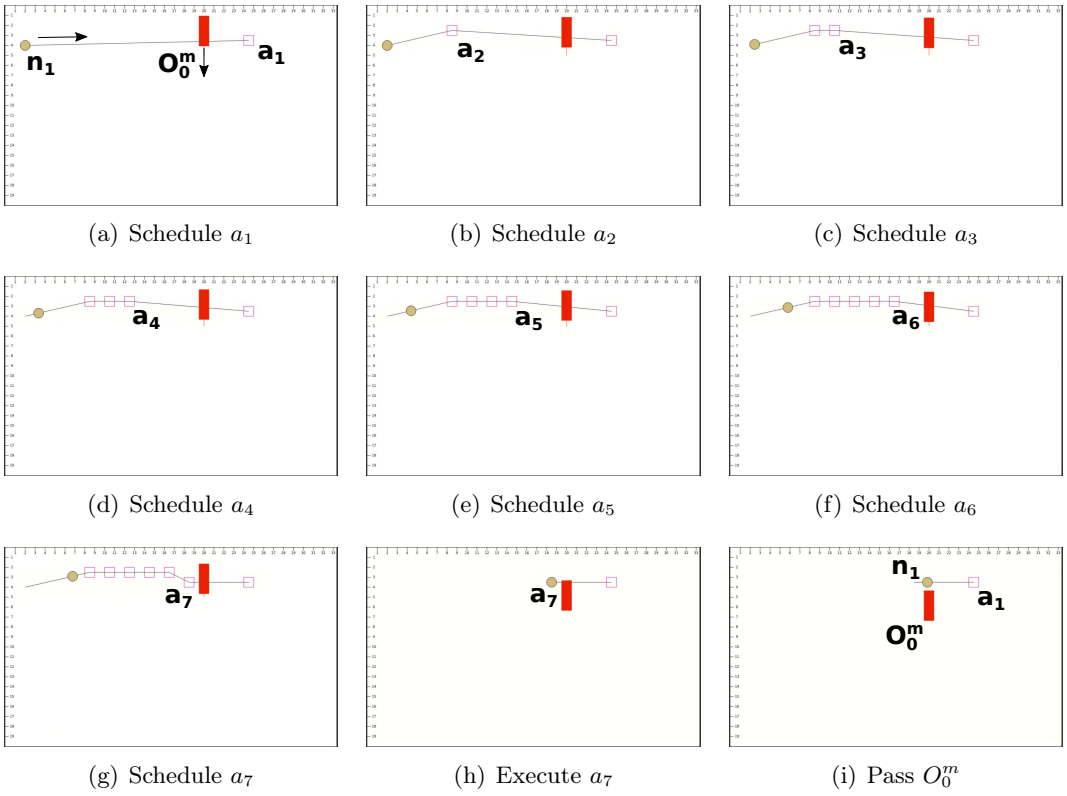
Table 8.2: World set-up: 1 node and 1 dynamic obstacle O^m.

the adapted two-phase commit protocol by communicating with the involved nodes using message passing and, finally, committed or aborted the (distributed) transaction and, hence, removed one of the path alternatives. The execution of the scheduled and committed applications have been performed distributed on several machines. Section 8.3.1 shows the system behavior when scheduling a set of actions while a slow dynamic obstacle crosses the node's path. Section 8.3.2 presents two applications that form two triangles consisting of six nodes that "approach" each other based on the spatio-temporal constraints of the actions. Using the approach presented in this thesis, both applications could be successfully scheduled and executed by using virtual movement. Section 8.3.3 shows that, using the approach presented in this thesis, nodes can be navigated through a tight labyrinth consisting of multiple static obstacles. Two actions are created and two nodes are present in the world. The shortest paths for the nodes are computed to the respective execution location of the actions which leads through the labyrinth.

All experiments that have been performed in this section are set up in a world with an arrangement of the x and y axis as depicted in Figure 8.8(a). The schedule of the nodes, which includes movement and action jobs, is visualized in Figure 8.8(b). Concrete spatial units (mm, m, ..) are neglected again. The units are kept abstract.

Action	d [in s]	t_{min} [in s]	t_{max} [in s]	g [rect((x_1, y_1), (x_2, y_2))]
a_1	1	5	10^4	$(24, 3), (25, 4)$
a_2	1	5	10^4	$(8, 2), (9, 3)$
a_3	1	5	10^4	$(10, 2), (11, 3)$
a_4	1	5	10^4	$(12, 2), (13, 3)$
a_5	1	5	10^4	$(14, 2), (15, 3)$
a_6	1	5	10^4	$(16, 2), (17, 3)$
a_7	1	5	10^4	$(18, 3), (19, 4)$

Table 8.3: Action specifications: 7 actions *before* and *behind* O^m.



(a) Schedule a_1 (b) Schedule a_2 (c) Schedule a_3

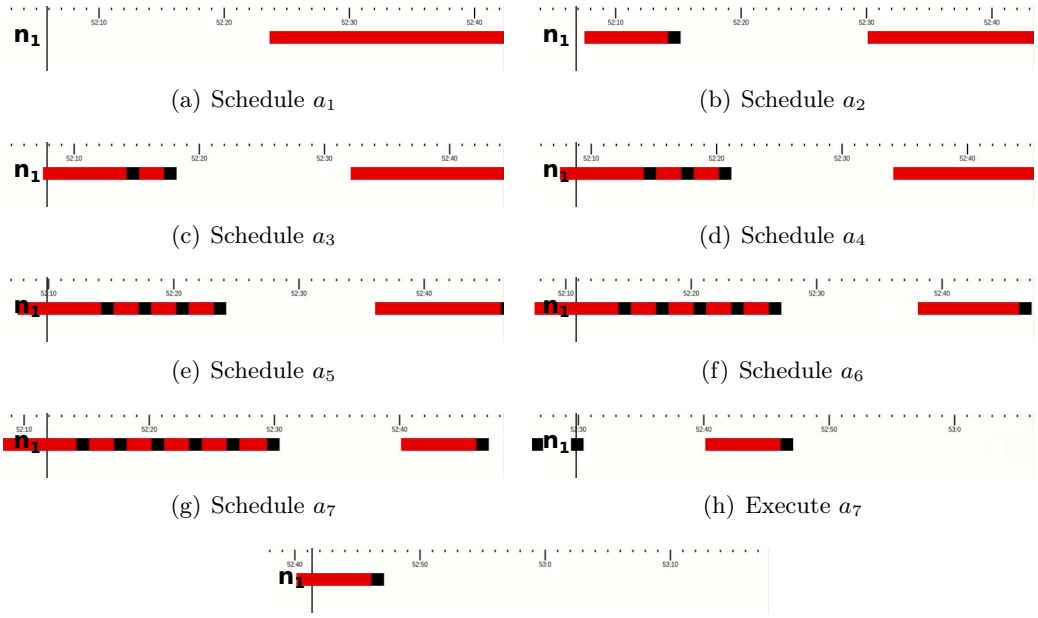
(d) Schedule a_4 (e) Schedule a_5 (f) Schedule a_6

(g) Schedule a_7 (h) Execute a_7 (i) Pass O_0^m

Figure 8.9: Visualization of scenario progress.

8.3.1 Slow Dynamic Obstacles and Waiting Times

As described in Section 7.5.5 (page 118), low velocities result in a bad system utilization. In the following the improvement of using waiting times and preferring higher velocities is demonstrated. Table 8.2 shows the world set-up given one dynamic obstacle O_0^m and one node n_1. The dynamic obstacle has a rectangular shape and appears at the coordinate $(20, 1)$ at time 2. It then continues moving with a constant speed of $v_{O_0^m} \approx 0.093$ to



(a) Schedule a_1

(b) Schedule a_2

(c) Schedule a_3

(d) Schedule a_4

(e) Schedule a_5

(f) Schedule a_6

(g) Schedule a_7

(h) Execute a_7

(i) Pass O_0^m

Figure 8.10: Alternating schedule of n_1 while time progresses.

coordinate $(20, 5)$. It arrives at $t = 45$. In this scenario, there are in total 7 actions that will be scheduled. Table 8.3 states the actions' specifications. The constraint g spans a rectangular space window given by (x_1, y_1) and (x_2, y_2). The overall progress of the scenario is visualized in Figure 8.9 while Figure 8.10 shows the adapted schedule of node n_1 while time progresses. Each action is scheduled separately, i.e., in a different action suite. The first action a_1 has to be executed spatially behind[2] the dynamic obstacle O_0^m as depicted in Figure 8.9(a). Without using the waiting times, the robot would move with constant low speed towards the execution location of a_1. In this case, the robot would not be able to accept new jobs in the meanwhile resulting in a bad job utilization u_r^j (Section 5.4.1).

Using the waiting time, the robot will move with a high velocity and waits as long as possible at its current location in order to possibly accept new jobs. While time progresses, the actions $a_2, a_3, .., a_6$ are scheduled iteratively, resulting in an adaption of the schedule of n_1 as shown in Figure 8.9(b) - Figure 8.9(g) and Figure 8.10(b) - Figure 8.10(g), respectively. As a result, the system is able to accept all remaining 6 actions and schedules them before a_1 since n_1 has enough capacity in order to execute them. This results in a modification of the original trajectory. As shown in Figure 8.9(g), the robot passes O_0^m without a collision and still in time.

[2]In this case, it means spatially *behind* the dynamic obstacle as seen from the robot's perspective.

nodes	initial position	speed
n_1	position: $(2, 1.5)$	$v_{max} = 1$
n_2	position: $(5, 10.5)$	$v_{max} = 1$
n_3	position: $(2, 19.5)$	$v_{max} = 1$
n_4	position: $(31, 1.5)$	$v_{max} = 1$
n_5	position: $(26, 10.5)$	$v_{max} = 1$
n_6	position: $(31, 19.5)$	$v_{max} = 1$

Table 8.4: World set-up: 6 nodes forming triangles.

8.3.2 Triangle Formations

This scenario illustrates the difference between virtual and physical movement. In order to demonstrate this, two applications are requested to be scheduled. Table 8.4 describes the arrangement of nodes. Nodes are shaped as circle again. Table 8.5 (page 145) states the respective specifications of suites and actions together with their spatio-temporal constraints. The duration is kept constant: $d = 1$.

Both applications (app_1, app_2) consist of 7 action suites denoted by as_i^l (for app_1) and as_i^r (for app_2), $i \in [1, .., 7]$. Each suite contains exactly 3 actions a_1, a_2, a_3 resulting in a total amount of $2 \cdot 7 \cdot 3 = 42$ actions. All actions have different spatio-temporal constraints. Due to the specification of the spatio-temporal constraints, all actions in the same suite lead to the formation of a triangle. For all actions in the suites as_i^l with $i > 1$, the temporal constraints are increased by $(i-1) \cdot 12s$ and the spatial x constraints are increased by $(i-1) \cdot 4$.

For all actions in the suites as_i^r with $i > 1$, the temporal constraints are increased by $(i-1) \cdot 12s$ and the spatial x constraints are decreased by $(i-1) \cdot 4$. Due to the spatial constraints, the suite as_1^l starts on the left side and "moves" to the right. Therefore, the index l is used. The suite as_1^r starts on the right side and "moves" to the left side. Therefore, the index r is used.

The action suites are scheduled (and executed) in the following order: $for\ i \in \{1, 2, .., 7\} : schedule(as_i^l), schedule(as_i^r)$. Figure 8.11(a) - Figure 8.11(i) (page 146) visualize the execution including the current formations of the nodes while Figure 8.12(a) - Figure 8.12(i) (page 147) show the corresponding schedule. For simplicity, a formation caused by an action suite as_y^x is denoted by x_y as depicted in Figure 8.11. The two applications app_1 and app_2 are visualized by the formations l_i and r_i, respectively.

Figure 8.11(a) visualizes the execution of as_1^r. The suite as_1^l has already been executed previously. At the current point in time the formation is still maintained. Figure 8.11(b) shows the execution of as_2^l. This continues in Figure 8.11(c) by previously executing as_2^r and then executing as_3^l. The two triangles represented by the suites of as^l and as^r "approach" each other by moving the 3 nodes on the left (n_1, n_2, n_3) as a group to the right while the 3 nodes on the right side (n_4, n_5, n_6) move as a group to the left. This behavior changes in Figure 8.11(d) since the left most point of the triangle represented by as_3^r is now switched over from n_5 to the right most node of the left group (n_2). In Figure 8.11(e),

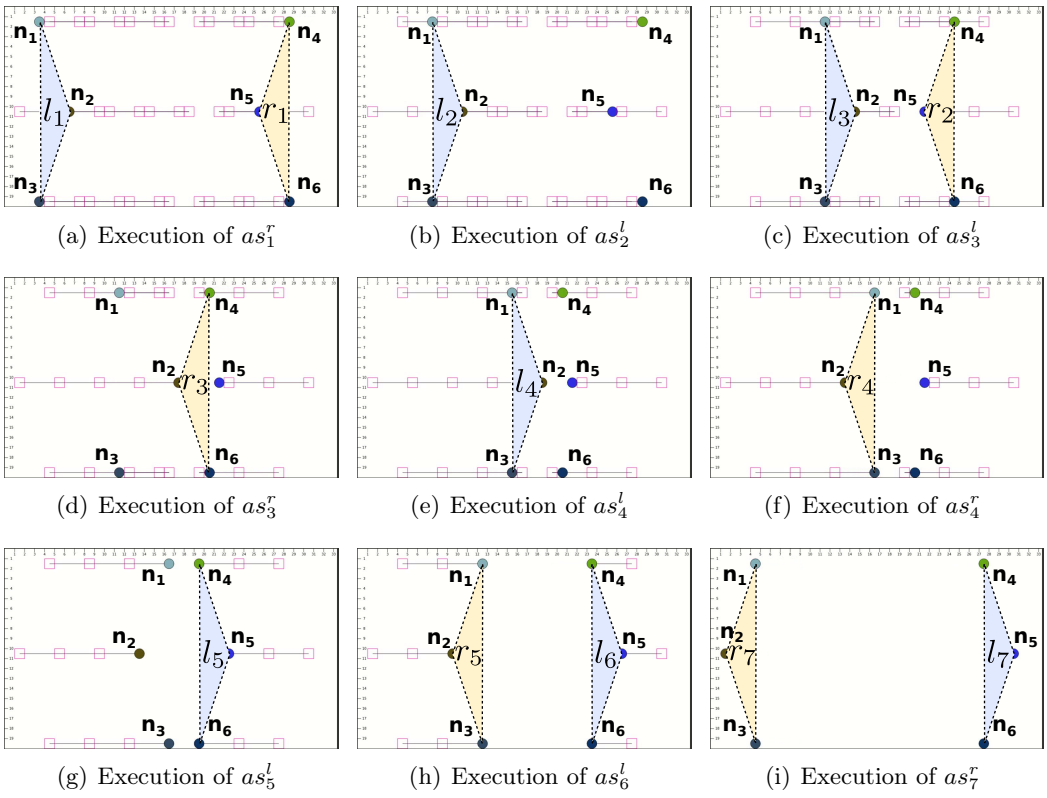
App	Suite	Action	t_{min} [in s]	t_{max} [in s]	g [rect(..)] (x_1, y_1), (x_2, y_2)
app_1	as_1^l	a_1	5	6	$(3,1),(4,2)$
		a_2	5	6	$(6,10),(7,11)$
		a_3	5	6	$(3,19),(4,20)$
	as_2^l	a_1^*	$a_1.t_{min}+12$	$a_1.t_{max}+12$	$a_1.g.x+4$
		a_2^*	$a_2.t_{min}+12$	$a_2.t_{max}+12$	$a_2.g.x+4$
		a_3^*	$a_3.t_{min}+12$	$a_3.t_{max}+12$	$a_3.g.x+4$
	$\vdots$				
	as_7^l	...			
app_2	as_1^r	a_1	11	12	$(28,1),(29,2)$
		a_2	11	12	$(25,10),(26,11)$
		a_3	11	12	$(28,19),(29,20)$
	as_2^r	a_1^*	$a_1.t_{min}+12$	$a_1.t_{max}+12$	$a_1.g.x-4$
		a_2^*	$a_2.t_{min}+12$	$a_2.t_{max}+12$	$a_2.g.x-4$
		a_3^*	$a_3.t_{min}+12$	$a_3.t_{max}+12$	$a_3.g.x-4$
	$\vdots$				
	as_7^r	...			

Table 8.5: Action specifications: 42 actions forcing triangle formations.

the triangle represented by as_4^l is still formed by n_1, n_2, n_3. In Figure 8.11(f), the triangle represented by as_4^r is now completely switched over to n_1, n_2, n_3 while in Figure 8.11(g) the formation (as_5^l) is carried out by n_4, n_5, n_6. At this point in time both applications have completely switched their executing devices which is completely transparent for the application. How the robots are moving back towards their origin while executing the remaining actions, is visualized in Figure 8.11(h) and Figure 8.11(i).

Figure 8.12(a) - Figure 8.12(i) depict the respective schedule at the specific points in time, that is, when the respective action suite is executed. The 3 contained actions are marked. The color is used to distinguish between both applications. Figure 8.12(a) shows the execution of as_1^r of app_2 by the nodes n_4, n_5 and n_6. Figures 8.12(b) and 8.12(c) indicate the execution of as_2^l and as_3^l of app_1 by the nodes n_1, n_2 and n_3. Figure 8.12(d) indicates the switching of nodes. Previously as_1^r and as_2^r have been executed by the nodes n_4, n_5 and n_6. Here, as_3^r is executed by the nodes n_2, n_4 and n_6. The last suite of app_1 that is executed by the nodes n_1, n_2 and n_3 is as_4^l (Figure 8.12(e)). Afterwards, all remaining suites off app_1 ($as_5^l - as_7^l$) are executed by the nodes n_4, n_5 and n_6 while all remaining suites off app_2 ($as_4^r - as_7^r$) are executed by the nodes n_1, n_2 and n_3.

This scenario illustrates the difference between virtual and physical movement. While app_1 strictly "moves" from the left side to the right and app_2 "moves" strictly from the right side to the left, the executing nodes move physically to approximately the center of the map and then move back. So the physical movement is different from the virtual movement.



(a) Execution of as_1^r (b) Execution of as_2^l (c) Execution of as_3^l

(d) Execution of as_3^r (e) Execution of as_4^l (f) Execution of as_4^r

(g) Execution of as_5^l (h) Execution of as_6^l (i) Execution of as_7^r

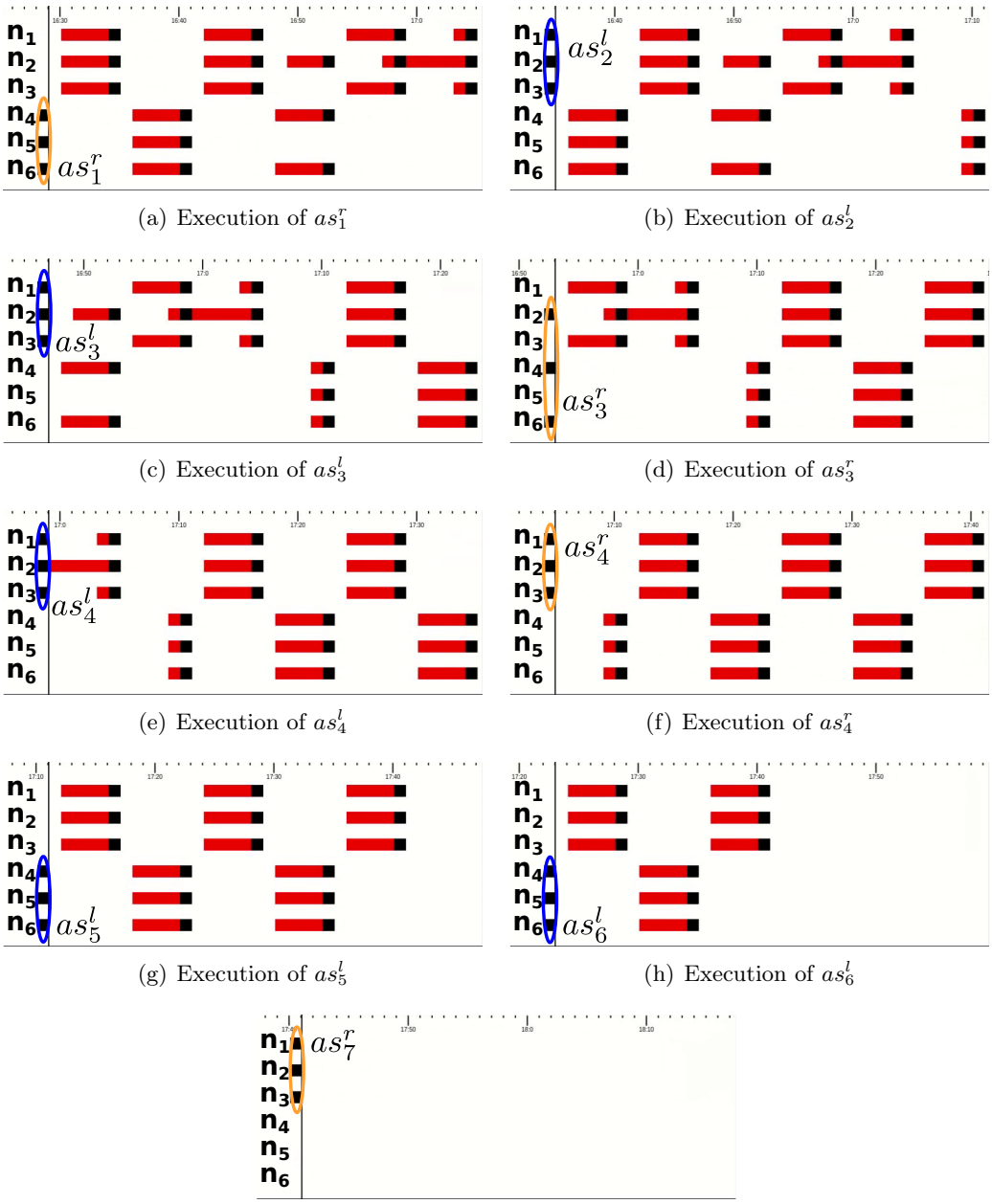
Figure 8.11: Visualization of scenario progress.

This happens due to the following reasons: first, the robots are on the same y-ordinate which would result in a crash if they would continue moving. Initiating a detour around the approaching ones would result in additional movement and might cause a temporal constraint violation. Second, simply switching the mapping of actions to nodes results in less physical movement. Third, there might be a large static obstacle that makes a detour impossible. The "length" of the virtual movement states the "distance" that one application travels and is defined as:

$$s_{virt} = \sum_{i=1}^{m} \sum_{j=1}^{n-1} ||as_j.\vec{x}_{a_i} - as_{j+1}.\vec{x}_{a_i}|| \tag{8.2}$$

Here, m is defined as the number of actions in a suite and n is defined as the number of suites per application. The physical movement s_{phys} is defined as movement that nodes perform. In this scenario, the total virtual movement of app_1 is $s_{virt} = 3 \cdot 24 = 72$. The same holds for app_2: $s_{virt} = 72$ which results in a total virtual movement of 144.

The physical movement is given by: $s_{phys}^{n_1} = 25$, $s_{phys}^{n_2} = 29$, $s_{phys}^{n_3} = 25$, $s_{phys}^{n_4} = 17$, $s_{phys}^{n_5} = 13$ and $s_{phys}^{n_6} = 17$. This results in a total physical movement of 126 which is a reduction of movement by 12.5 % or the absolute physical movement is 87.5 % of the



(a) Execution of as_1^r

(b) Execution of as_2^l

(c) Execution of as_3^l

(d) Execution of as_3^r

(e) Execution of as_4^l

(f) Execution of as_4^r

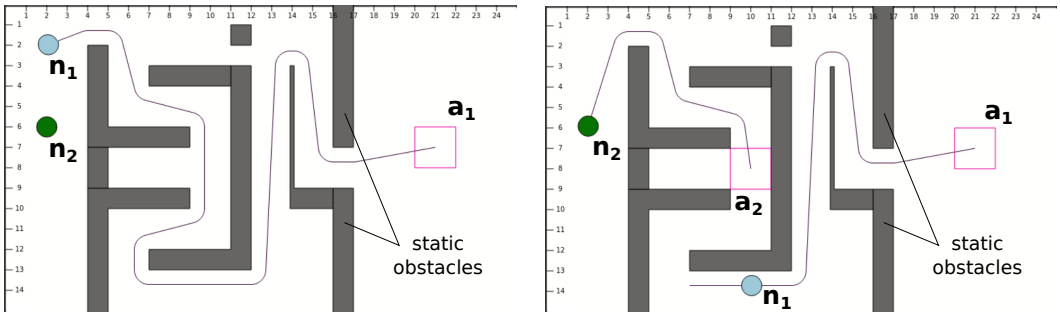
(g) Execution of as_5^l

(h) Execution of as_6^l

(i) Execution of as_7^r

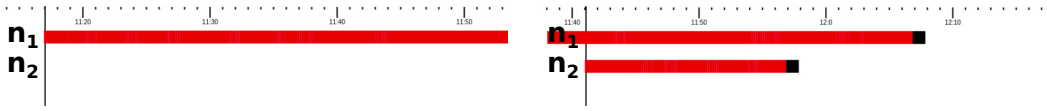
Figure 8.12: Alternating schedule of nodes while time progresses.

virtual movement. In this scenario, the journey has not been taken into account. The journey is defined as the path that is required in order to move a robot to the location of the initial action. When including the journey, the additional path length is added in equal measure to the robots trajectory as well as to the virtual movement of the



(a) Schedule of a_1 (b) Schedule of a_1, a_2

Figure 8.13: Visualization of scenario progress.



(a) Schedule of a_1 (b) Schedule of a_1, a_2

Figure 8.14: Alternating schedule of nodes while time progresses.

application. This might result in another fraction, but not in the order relation between physical and virtual movement.

8.3.3 Obstacles

In this section, static obstacles are examined. Figure 8.13 visualizes the world with static obstacles and 2 nodes. The world's size[3] is set to $x = 25, y = 15$. The 2 nodes are placed on $(2, 2)$ and $(2, 6)$. Both nodes have a velocity of $v_{max} = 1$. Static obstacles are dark shaded and form a small labyrinth. In this scenario, there are only 2 actions that are scheduled: a_1 with $g = (20, 6), (22, 8)$, $t_{min} = 2, t_{max} = 60$ and a_2 with $g = (9, 7), (11, 9)$, $t_{min} = 2, t_{max} = 60$. Both have a duration of $d = 1$. Action a_1 first comes into the system and is scheduled accordingly. Figure 8.13(a) visualizes the trajectory of the first node (n_1) that starts moving towards $\vec{x}_{a_1}$. Figure 8.14(a) shows the current schedule. The drawn trajectory depicts the shortest path between the node's position and the target given by $\vec{x}_{a_1}$. In order not to collide with the obstacles, the trajectory is calculated by moving very close to the obstacle, the nodes geometry is taken into account as described in Section 7.5 and a sufficient distance is kept to the obstacles. Due to this fact, the node is not able to move around the small squared obstacle located at position $(11, 1), (12, 2)$. The length of the trajectory is approximately 50. Since $v_{max} = 1$, the time required for moving along the trajectory is also approximately $50s$. 24 seconds after a_1 has been scheduled, a_2 is comes into the system and is scheduled (Figure 8.13(b) and 8.14(b)).

[3]The modification of the world's size has no influence on the simulation results. This setup was chosen since this size was sufficient.

The scheduler assigns a_2 to the second node (n_2). This is the only possible assignment since the temporal constraint of a_2 is given by $t_{min} = 2, t_{max} = 60$. At the point in time when a_2 arrives at the system, the remaining length of the trajectory that the node n_1 has to move along is 26. Due to v_{max}, the remaining time is $26s$. After arriving at $\vec{x}_{a_1}$, a_1 is executed for $1s$. The trajectory length beginning in $\vec{x}_{a_1}$ and ending in $\vec{x}_{a_2}$ is 38 (if n_1 would move back to $\vec{x}_{a_2}$ after executing a_1) and, hence $38s$ would be required for the movement. The total required time is $65s$ ($26s + 1s + 38s$) from the point in time when a_2 enters the system. Since t_{max} is set to $60s$, n_1 is not able to execute a_2. The only possible candidate is n_2.

8.4 Experiments on Testbed

This section presents different scenarios with and without obstacles and examines the system behavior. The experiments evaluate the system on a holistic level including the actual movement on the physical testbed. The robots are presented in Appendix A (page 165) and are located using a ceiling mounted Kinect (Appendix B, page 167). The movement is performed by the motion control component which has been introduced in Section 5.3.1 (page 63). The actual control of the two electric engines (rotation speed) is explained in Appendix C (page 171). The runtime system performed the adapted two-phase commit protocol by communicating with the involved nodes using message passing and, finally, committed or aborted the (distributed) transaction and, hence, removed one of the path alternatives. Section 8.4.1 describes experiments with four robots on the testbed. A set of 40 actions have been divided into 9 groups and scheduled accordingly.

World set-up	Description
n_1	
	shape: circle$((0,-8),(-8,0),(0,8),(8,0))$
	position: $(1700, 1400)$
	speed: $v_{max} = 100\frac{mm}{s}$
n_2	
	shape: circle$((0,-8),(-8,0),(0,8),(8,0))$
	position: $(1700, 1000)$
	speed: $v_{max} = 100\frac{mm}{s}$
n_3	
	shape: circle$((0,-8),(-8,0),(0,8),(8,0))$
	position: $(1700, 600)$
	speed: $v_{max} = 100\frac{mm}{s}$
n_4	
	shape: circle$((0,-8),(-8,0),(0,8),(8,0))$
	position: $(1700, 200)$
	speed: $v_{max} = 100\frac{mm}{s}$

Table 8.6: World set-up: 4 nodes arranged on the testbed.

The result indicates the (coordinated) robot movement on the testbed. Section 8.4.2 describes experiments with three robots on the testbed. In total 18 actions have been scheduled divided into 6 groups. Due to the spatio-temporal constraints, the robots are forced to form a triangle that moves over the testbed. This experiment has been performed in two ways: first, the triangle formation and the alignment is kept, i.e., each node maintains it particular position in the triangle. In the second experiment, the triangle rotates and, hence, the positions of the robots in the triangle change over time. Section 8.4.3 presents experiments with one robot executing a set of actions. The required movement formed a closed polygonal chain. The accuracy of movement is presented as a function of the threshold value λ. Finally, the memory allocation has been examined. Since, the experiments have been performed on the testbed, the applied spatial units are in mm.

8.4.1 Four Robot Movement

In this scenario, the entire stack ranging from the programming model over the scheduling to the real physical distributed execution is evaluated and presented as a proof-of-concept of this work.

The testbed has a size of $x = 2200$ mm and $y = 1600$ mm. The world for the scheduler is set accordingly. In the beginning there are no static obstacles. Table 8.6 states the set-up of the four nodes, their geometry, initial position and speed.

Since there are four nodes in total, there are three dynamic obstacles in the world for each node. The following scenario is cut into two parts: first, all four nodes are involved in the execution of actions. In the second part, three robots are excluded from the system and represent static obstacles. Figure 8.15 shows the first part of the scenario.

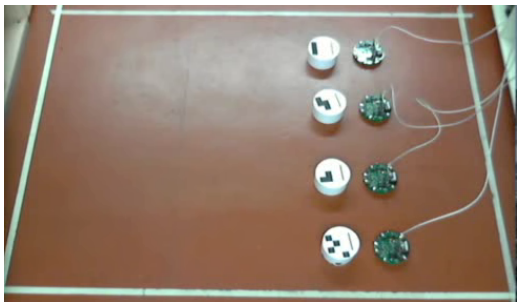
Actions are scheduled in groups in the following order (the order between groups as well as inside the group matters):

- *group1*: a_1, a_2, a_3, a_4
- *group2*: a_5, a_6, a_7, a_8
- *group3*: $a_{12}, a_{11}, a_{10}, a_9$
- *group4*: $a_{13}, a_{14}, a_{15}, a_{16}$
- *group5*: $a_{17}, a_{18}, a_{19}, a_{20}$

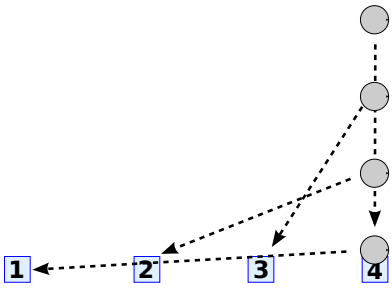
Table 8.7 states the specifications of the actions. The spatial constraint is defined as a square: the respective points (x, y) define the center of the 16×16 square that surrounds the point. Figure 8.15(a) depicts the initial line-up of the robots. The cover is used for locating the robots. The location information is forwarded to the respective system services. Figure 8.15(b) illustrates the scheduling of the first group of actions and the assignments of jobs to nodes. Figure 8.15(c) shows the robots after moving to $\vec{x}_{a_1}$ - $\vec{x}_{a_4}$ and executing the actions. In Figure 8.15(d), the groups 2, 3 and 4 are scheduled and

Group	Action	d [in s]	t_{min} [in s]	t_{max} [in s]	g [in (x mm, y mm)]
group1	a_1	1	5	19	$(150, 190)$
	a_2	1	5	19	$(600, 190)$
	a_3	1	5	19	$(1150, 190)$
	a_4	1	5	19	$(1700, 190)$
group2	a_5	1	17.5	31.5	$(150, 1400)$
	a_6	1	17.5	31.5	$(600, 1400)$
	a_7	1	17.5	31.5	$(1150, 1400)$
	a_8	1	17.5	31.5	$(1700, 1400)$
group3	a_9	1	21	35	$(400, 1400)$
	a_{10}	1	21	35	$(950, 1400)$
	a_{11}	1	21	35	$(1500, 1400)$
	a_{12}	1	21	35	$(2050, 1400)$
group4	a_{13}	1	46.5	47.5	$(400, 190)$
	a_{14}	1	46.5	47.5	$(950, 190)$
	a_{15}	1	46.5	47.5	$(1500, 190)$
	a_{16}	1	46.5	47.5	$(2050, 190)$
group5	a_{17}	1	59.5	60.5	$(450, 800)$
	a_{18}	1	59.5	60.5	$(1050, 800)$
	a_{19}	1	59.5	60.5	$(1700, 800)$
	a_{20}	1	59.5	60.5	$(1700, 1250)$
group6	a_{21}	1	–	–	$(1950, 800)$
	a_{22}	1	–	–	$(1700, 550)$
	a_{23}	1	–	–	$(1300, 800)$
	a_{24}	1	–	–	$(1050, 1050)$
	a_{25}	1	–	–	$(700, 800)$
	a_{26}	1	–	–	$(450, 550)$
	a_{27}	1	–	–	$(200, 800)$
group7	a_{28}	1	–	–	$(450, 1050)$
	a_{29}	1	–	–	$(800, 800)$
	a_{30}	1	–	–	$(1050, 550)$
	a_{31}	1	–	–	$(1450, 800)$
	a_{32}	1	–	–	$(1700, 1050)$
group8	a_{33}	1	–	–	$(1700, 400)$
	a_{34}	1	–	–	$(1500, 800)$
	a_{35}	1	–	–	$(650, 800)$
	a_{36}	1	–	–	$(450, 400)$
group9	a_{37}	1	–	–	$(450, 1200)$
	a_{38}	1	–	–	$(880, 800)$
	a_{39}	1	–	–	$(1500, 800)$
	a_{40}	1	–	–	$(1700, 1200)$

Table 8.7: Action specifications: 40 actions with spatio-temporal constraints. The spatial constraint g defines the center of a surrounding 16×16 square.

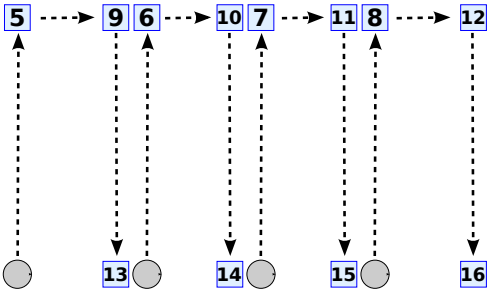


(a) Initial line-up

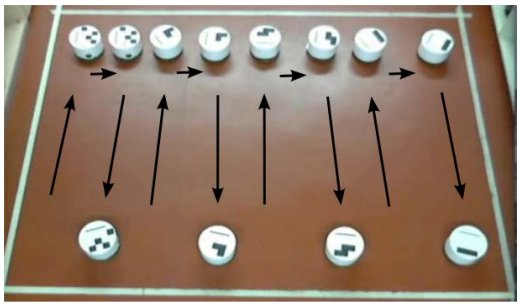


(b) Schedule *group1*

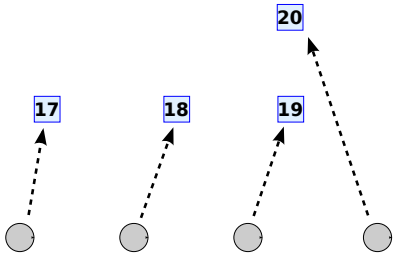
(c) Executing *group1*



(d) Schedule *group2, 3, 4*

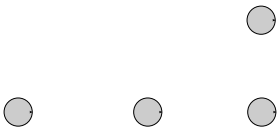


(e) Executing *group2, 3, 4*



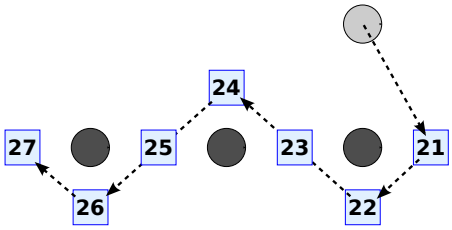
(f) Schedule *group5*

(g) Executing *group5*

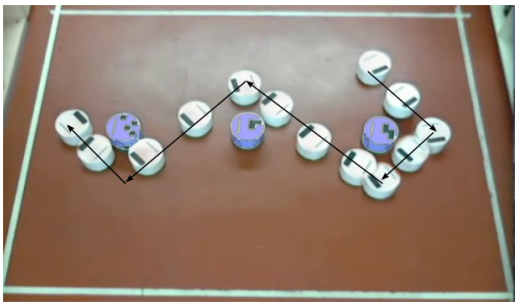


(h) Final position

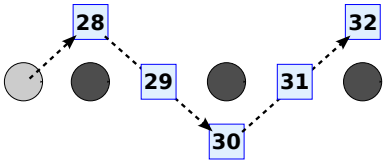
Figure 8.15: Coordinated 4 robot movement.



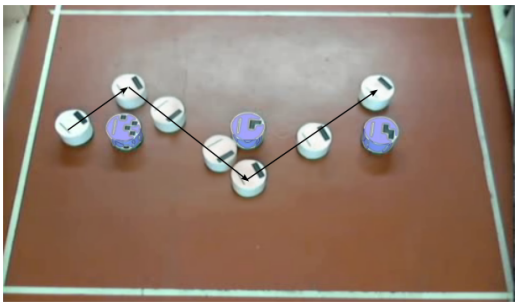
(a) Scheduling *group6*



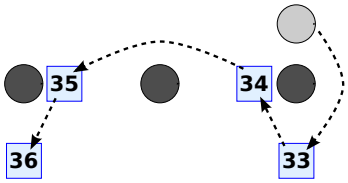
(b) Executing *group6*



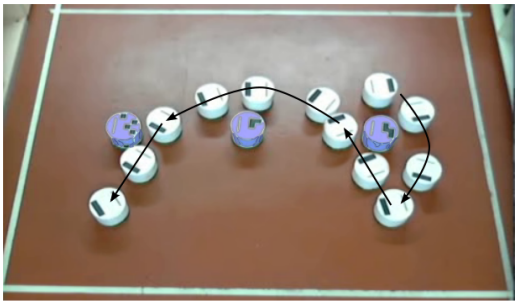
(c) Scheduling *group7*



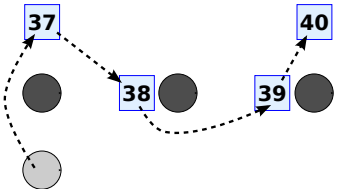
(d) Executing *group7*



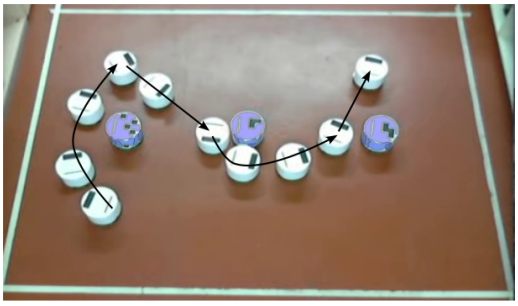
(e) Scheduling *group8*



(f) Executing *group8*



(g) Scheduling *group9*



(h) Executing *group9*

Figure 8.16: Robot sequentially executes a set of actions while moving around the obstacles.

the assignment of jobs to nodes including the trajectories are visualized. Figure 8.15(e) demonstrates the execution and movement phase of group 2, 3 and 4. Figure 8.15(f) depicts the scheduling of *group5* and Figure 8.15(g) shows its execution. Figure 8.15(h) illustrates the final positions.

In the second part, the nodes n_2, n_3 and n_4 are excluded from the system and are marked as static obstacles. Since node n_1 is the only node, it has to execute all actions and move around the obstacles. The following actions are scheduled in this order:

- *group6*: $a_{21}, a_{22}, .., a_{27}$
- *group7*: $a_{28}, a_{29}, .., a_{32}$
- *group8*: $a_{33}, a_{34}, a_{35}, a_{36}$
- *group9*: $a_{37}, a_{38}, a_{39}, a_{40}$

Figure 8.16 presents the execution of the groups 6 - 9 containing the actions a_{21} - a_{40}. Figure 8.16(a), 8.16(c), 8.16(e) and 8.16(g) once again show the scheduling of the respective groups while Figure 8.16(b), 8.16(d), 8.16(f) and 8.16(h) visualize the execution of the actions including the trace of the node movement. In Figure 8.16(a) - 8.16(d), the scheduling was straight forward since the shortest route between two adjacent actions did not lead through an obstacle. As a consequence, the robot moved in straight lines from one spot to the next one. In Figure 8.16(e) - 8.16(h) half of the adjacent actions have no line of sight since the direct path is blocked by an obstacle. Therefore, the shortest path is achieved by moving around the obstacle.

8.4.2 Triangle Formation

In this scenario, the 3-sided observation application is chosen again, but executed on the testbed. The experiments are split into two parts: in the first part, the formation is maintained while in the second part the formation rotates 90 degrees clockwise. The testbed once more has a size of $x = 2200$ mm and $y = 1600$ mm. Table 8.8 describes the line-up of the 3 nodes n_1, n_2 and n_3.

In the first part, 9 actions are scheduled that are arranged into 3 groups (*group1* - *group3*). Table 8.9 states the spatio-temporal constraints of the respective actions. Actions are scheduled in groups in the following order (the order between groups as well as inside the group matters):

- *group1*: a_1, a_2, a_3
- *group2*: a_4, a_5, a_6
- *group3*: a_7, a_8, a_9

World set-up	Description
n_1	
	shape: circle$((0,-8),(-8,0),(0,8),(8,0))$
	position: $(2050, 675)$
	speed: $v_{max} = 100\frac{mm}{s}$
n_2	
	shape: circle$((0,-8),(-8,0),(0,8),(8,0))$
	position: $(2080, 170)$
	speed: $v_{max} = 100\frac{mm}{s}$
n_3	
	shape: circle$((0,-8),(-8,0),(0,8),(8,0))$
	position: $(1780, 425)$
	speed: $v_{max} = 100\frac{mm}{s}$

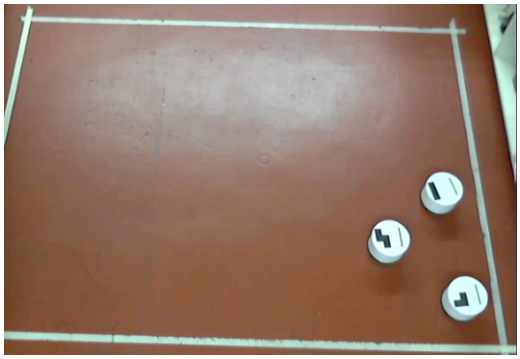
Table 8.8: World set-up: 3 nodes arranged on the testbed.

Figure 8.17(a) shows the initial line-up of the robots at the bottom right corner of the testbed. Next, *group1* is scheduled and assigned to the robots as depicted in Figure 8.17(b). As a result, the robots move to their scheduled positions ($\vec{x}_{a_1}$ - $\vec{x}_{a_3}$) (Figure 8.17(c)). The next set of actions (*group2*) is scheduled (Figure 8.17(d)) and executed as indicated in Figure 8.17(e) at location $\vec{x}_{a_4}$ - $\vec{x}_{a_6}$. Figure 8.17(f) shows the scheduling of the last group (*group3*) while Figure 8.17(g) depicts the execution. Figure 8.17(h) visualizes the final positions. During the first part the formation is maintained.

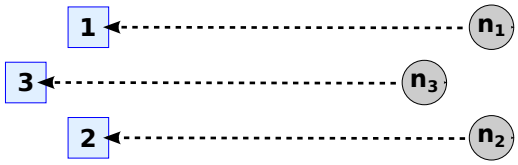
In the second part, the formation is shifted by rotating the formation clockwise by 90 degrees. The experiments have been performed by again setting up 9 actions in total $(a_{10}, a_{11}, .., a_{18})$. The actions are arranged in additional 3 groups. The actions are submitted to the schedule in the following order:

- *group4*: a_{10}, a_{11}, a_{12}
- *group5*: a_{13}, a_{14}, a_{15}
- *group6*: a_{16}, a_{17}, a_{18}

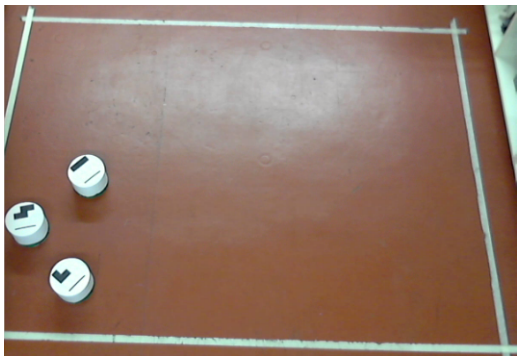
Figure 8.18(a) illustrates the initial line-up of the 3 robots. Figure 8.18(b) shows the scheduling of *group4* and the assignment of the associated actions a_{10}, a_{11}, a_{12}. The group of actions is rotated 90 degrees clockwise compared to the initial line-up. The arrows indicate the movement of the robots. Figure 8.18(c) demonstrates the execution of actions while the robots have obtained the new formation. Figure 8.18(d) visualizes the scheduling of *group5* and Figure 8.18(e) shows its execution and the formation which is rotated by 180 degrees compared to the initial line-up. Figure 8.18(f) depicts the scheduling of the last group (*group6*) and Figure 8.18(g) shows the execution. Figure 8.18(h) illustrates the final position. The first robot of the triangle (the top) is also always rotating, starting with robot n_3 over n_1 and n_2 back to n_3 (Figure 8.18).



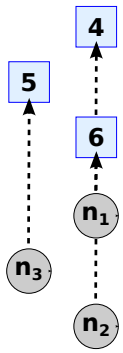
(a) Initial line-up



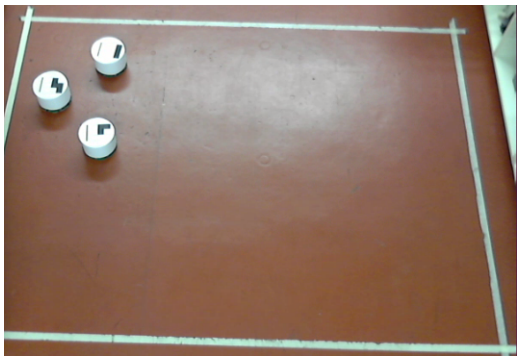
(b) Scheduling *group1*



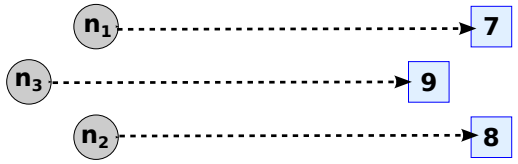
(c) Executing *group1*



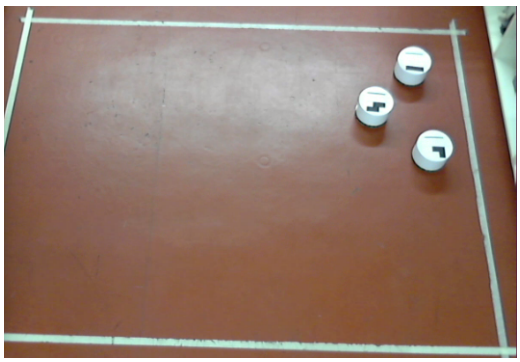
(d) Scheduling *group2*



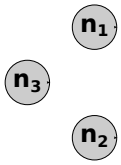
(e) Executing *group2*



(f) Scheduling *group3*

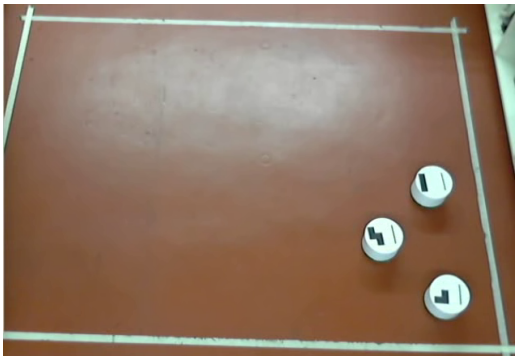


(g) Executing *group3*

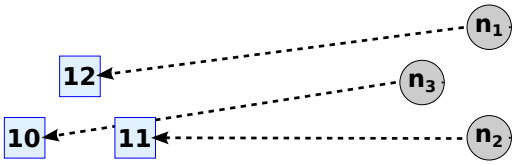


(h) Final positions

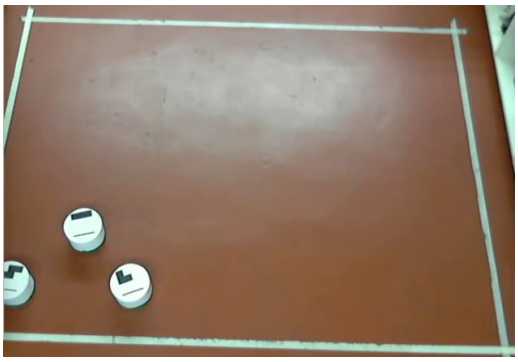
Figure 8.17: 3-sided observation application on the testbed with 3 robots (maintaining formation).



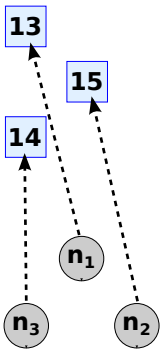
(a) Initial line-up



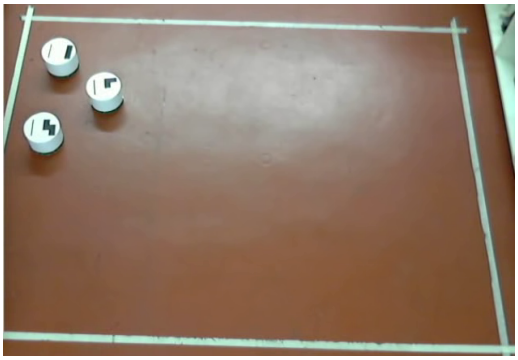
(b) Scheduling *group4*



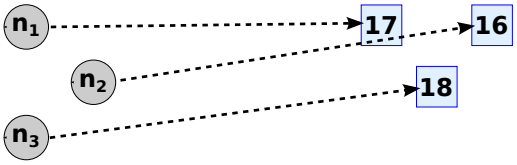
(c) Executing *group4*



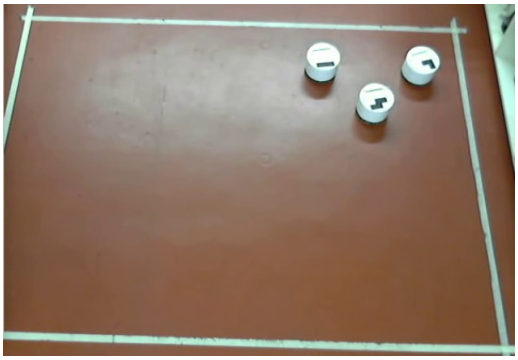
(d) Scheduling *group5*



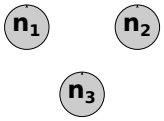
(e) Executing *group5*



(f) Scheduling *group6*



(g) Executing *group6*



(h) Final positions

Figure 8.18: 3-sided observation application on the testbed with 3 robots (changing formation).

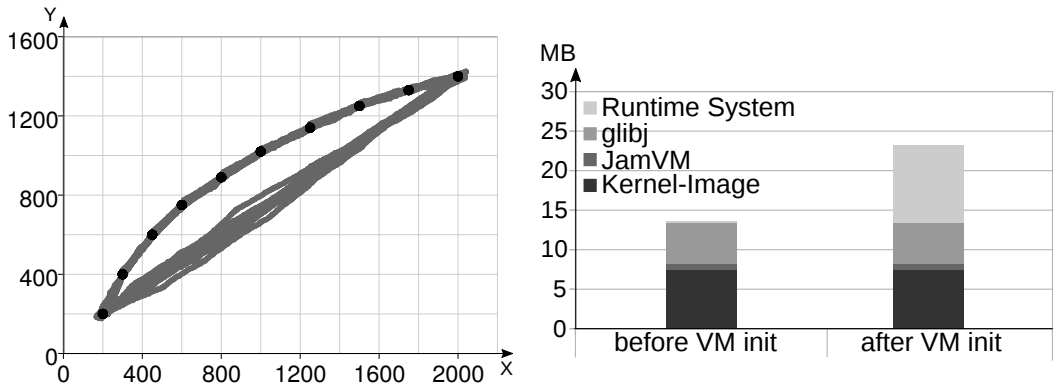
Group	Action	d [in s]	t_{min} [in s]	t_{max} [in s]	g [in (x mm, y mm)]
group1	a_1	1	18	19	$(450, 700)$
	a_2	1	18	19	$(450, 200)$
	a_3	1	18	19	$(200, 450)$
group2	a_4	1	27	28	$(450, 1400)$
	a_5	1	27	28	$(200, 1150)$
	a_6	1	27	28	$(450, 900)$
group3	a_7	1	44.5	45.5	$(2000, 1400)$
	a_8	1	44.5	45.5	$(2000, 900)$
	a_9	1	44.5	45.5	$(1750, 1150)$
group4	a_{10}	1	18	19	$(175, 200)$
	a_{11}	1	18	19	$(675, 200)$
	a_{12}	1	18	19	$(425, 450)$
group5	a_{13}	1	30	31	$(200, 1400)$
	a_{14}	1	30	31	$(200, 900)$
	a_{15}	1	30	31	$(450, 1150)$
group6	a_{16}	1	48	49	$(2000, 1400)$
	a_{17}	1	48	49	$(1500, 1400)$
	a_{18}	1	48	49	$(1750, 1150)$

Table 8.9: Action specifications: 18 actions forcing triangle formations.

As demonstrated in this scenario, the robots do not keep the formation the entire time during movement as this is not explicitly stated by the programmer. The programmer specifies spatio-temporal conditions expressed by the actions, the programmer does not specify a formation though this is possible by creating actions such that the only possible execution is to keep that formation (Section 8.3.2). However, in general, the formation itself does not matter. What matters is the application's intention and the resulting set of spatio-temporal actions which produces a certain formation.

Due to flocking or swarming, formations of swarms in nature (consisting of birds, bees, ants, etc.) are an emergent behavior caused by simple local decisions and keeping equal distances to all neighbors in the individual's proximity. This behavior is used in order to coordinate the swarm, keeping it together and constituting it as one unit. Coordination is performed on the individual's level resulting in emergent behavior: flocking.

In the approach presented in this thesis, the coordination is done by the system and, thus, the individual nodes are seen as executing components. Therefore, keeping a formation the entire time is not necessary. It is not even intended. As in bio-inspired swarms, a formation is the (emergent) result based on stigmergy in order to control the swarm, here a formation is the produced result of a set of actions that are either explicitly specified to construct that formation or are generated as a result of single incidents. In other words: the formations is a runtime product of executing a set of applications together with their spatio-temporal actions.



(a) Observation task

(b) Memory allocation

Figure 8.19: Movement accuracy and memory footprint.

8.4.3 Memory Usage and Movement Accuracy

In this section, the memory consumption of the runtime system is measured and the accuracy of the movement of the robots in the testbed is presented.

In Figure 8.19(a), the accuracy of the robot's movement is examined. Initially, one robot is placed on the location (200, 200). Then 10 actions are scheduled. The spatio-temporal constraints of the first 9 actions are arranged in increasing order, i.e., the spatial (x and y) as well as the temporal constraints are in increasing order. The respective 9 locations $\vec{x}_{a_1}, \vec{x}_{a_2}, .., \vec{x}_{a_9}$ produce an arc between the coordinates (200, 200) and (2000, 1400). The last action was constrained such that the robot has to move back to its initial location ($\vec{x}_{a_{10}} = (200\ 200)^T$). The accuracy of the movement is controlled by a threshold λ that indicates the maximum allowed deviation between the robots current orientation and the orientation to its next destination given by $\vec{x}_{a_i}$. Reaching the threshold λ, the robot has to adjust its orientation in order to face its next destination. In this experiment, the applied threshold value is iteratively increased starting from $\lambda = 1°$ to finally $\lambda = 11°$. Initially, the value is set to $1°$ and the actions $\vec{x}_{a_1}, \vec{x}_{a_2}, .., \vec{x}_{a_{10}}$ are executed. After the robot has reached its initial location, the threshold is increased to $\lambda = 2°$ and ten new actions are generated with increasing temporal constraints. This procedure is repeated until the threshold has reached $\lambda = 11°$ (last round). Figure 8.19(a) visualizes the traces of the robot while executing the actions. A detailed analysis of this experiment is given in Appendix C.

The devices do not feature a persistent storage and, therefore, all data is kept volatile in main memory. Both types of used devices as introduced in Appendix A are equipped with 64 MB SDRAM. Devices are started via net boot using TFTP. After the image which contains the Linux kernel is loaded over the network, the device initiates the boot sequence. When the system is up and running, the memory allocation is as follows: 7.5 MB is used by Linux. The experiments have been performed using the JamVM which

required 700 KB. The Java class library (*glibj*) required 5.2 MB[4]. After the runtime system has been started, which includes the initialization of the Java virtual machine and loading all necessary classes from the glibj, the memory consumption increases from 13.4 MB to 23.2 MB as depicted in Figure 8.19(b). 9.8 MB is used by the runtime system.

[4]Originally, the *glibj* has a size of 10 MB. The used version has been stripped in order to consume less memory.

Chapter 9

Conclusion and Future Work

This chapter concludes the thesis and states future work in this research field.

9.1 Conclusion

This thesis presented an approach for the programming of swarms of, especially mobile, devices on a systemic level. Error-prone aspects such as concurrency and distribution are hidden beyond the system's interface. The system's capabilities can be accessed system-wide by actions which are loosely coupled building blocks of a program. A notion of real space and time is a necessity in cyber-physical systems which when have to be addressed and handled explicitly by the programmer, further increases the complexity of application programming. Therefore, the system provides context awareness for actions by attaching spatio-temporal constraints (by the programmer). Furthermore, using the presented programming abstraction allows the programmer to specify the applications intention by giving a systemic description which states the spatio-temporal relations of resource usage: runtime parameters specify *when* and *where*—the time interval and location space in terms of physical coordinates—a certain action (*what*) has to be executed.

Using the approach presented in this thesis, application programming is strongly facilitated: programmers neither have to cope with concurrency nor distribution. Context awareness is achieved by simply assigning spatio-temporal constraints to actions. Event-based programming enables sense-and-react behavior. All spatio-temporal relations are expressible using the programming abstraction, e.g., workflows, using logical dependencies, simultaneous as well as before-after relations using temporal constraints. Using spatial constraints, every desired arrangement of locations, e.g., normal / uniform distribution, are programmable. Simple asynchronous and non-blocking system interface operations enable the allocation, reallocation or release of resources. Actions that logically belong together are encapsulated in suites in order to schedule them as a group. If confirmed by the system, contracts enable guaranteed resource allocation making a program more predictable.

All required movement of possibly multiple heterogeneous devices featuring different movement capabilities (flying, floating, grounded) that is necessary for the (distributed)

execution of the application is also completely transparent for the programmer. Before execution, the scheduler of the system has to check if all requested resources will be available according to the spatio-temporal constraints. This includes the computation of collision-free spatio-temporal trajectories. The following voting and notification phase assures that all participating nodes are informed in time.

Due to its modular architecture, the system is easily extensible. New nodes featuring new capabilities can be plugged into the system. Doing so might require implementing a device specific control algorithm for steering the device. This has to be done before the respective node is added to the system. Once developed, the new control algorithm for that device has to be registered such that the motion controller can simply invoke the new control algorithm in order to reach points of the spatio-temporal trajectory.

If the device features capabilities (sensors and actuators), it must be assured that suitable drivers are present by either using existing ones or implementing new ones. Afterwards, the node can be started and will be under the control of the swarm system.

9.2 Future Work

There are several directions in which future work can lead. Scaling up the system by increasing the number of nodes, the system performance is expected to go down since the scheduler currently is implemented as a central service. As stated in [49, 101], a "tera-swarm" is expected. Scaling up the system to such extent requires new mechanisms for scheduling. Hierarchical and decentralized scheduling will be key technologies which have to be investigated. The investigation should take the following into account:

- *Hierarchical scheduling*: the entire swarm is partitioned into cells. Each cell has its fixed amount of resources, i.e., mobile and stationary devices equipped with sensors and actuators. In a hierarchical organization, each cell has its own scheduler that manages resources in that particular cell. The hierarchy can be organized based on regions, e.g., the root node of the resulting tree represents the world while the leaves point to the respective cells. All nodes on layers in-between represent administrative regions, e.g., continents, countries, zones, etc. A request attempt to schedule an amount of actions results in traversing the tree based on the spatial constraints. Reaching the final leaf node points to the cell for which a local scheduler is responsible. If a set of actions fall into more than one cell, then the parent instance, that manages both cells, has to take care that the actions are scheduled in the respective cells. A consensus protocol has to assure that all involved scheduling instances finally adopt the same value (*commit* or *abort* the actions that they were responsible for).

- *Decentralized scheduling*: in a completely decentralized world, each node has its own scheduler. A schedule attempt is performed locally in two steps: first, the node n_1 computes a partial schedule for the respective action(s). In the second phase, the partial schedule is merged into a global schedule data structure. If the merging is collision-free, i.e., the global schedule could be updated with the partial

scheduling without violating its integrity, the schedule attempt is considered to be successful. In the other case, the merging produces a collision which might appear if another node has already merged its partial schedule and n_1 was not aware of. In this case, n_1 updates its local schedule by the portion of the global schedule in which the collision took place and then starts over with step 1 by performing the local scheduling again. Finally, in step 2 the merging will, at least with a higher probability, result in a success. In order to reduce overall data transfer, the global schedule can be partitioned along two dimensions: space and time. Only the portion that is necessary, a cuboid along the treated space and time dimension, is extracted and exchanged between the requesting node and the instance where the global schedule or a partition of the global schedule is hosted. There are different strategies that should be investigated and evaluated: the global schedule itself is partitioned and distributed among the nodes preferable with backups of the partitions. Parts of the schedule, cuboids, that span the area from the present into the near future on the time dimension are periodically exchanged among the nodes. In addition, locking strategies in contrast to lock-free approaches should be monitored.

- *Coarse-grained scheduling*: the presented scheduling algorithm has to know the spatio-temporal trajectories of all dynamic obstacles. As shown in the evaluation of the scheduler section, the cost-intensive part of the scheduling is the VPP which calculate the velocity profile along the trajectory in order to avoid collisions with dynamic obstacles. For this, forbidden regions have to be computed which is very cost-intensive. A solution for further improving the scheduler performance could be to neglect the VPP and only address the PPP. This way, the mobile nodes only have a coarse-grained path. When other dynamic obstacles are crossing their path, the robots perform a local evasive maneuver in order to avoid collisions. There are decentralized approaches in which all robots follow simple rules in order to avoid collisions when they are on a collision course. This avoids additional communication. There are also other approaches in which the entities negotiate how they proceed in order not to collide. Performing coarse-grained scheduling can be combined with the hierarchical or decentralized scheduling idea.

Besides the scheduling, there are other directions that should be addressed as well:

- *Energy consumption*: the scheduling prefers schedules which reduce overall movement. Since the necessary engine control for movement is probably the biggest impact factor on energy consumption, the scheduler somehow takes this into account by reducing the overall path length. However, energy consumption is not explicitly addressed here. This should be done in the future by incorporating energy into the scheduler model which might have an influence on the velocity and also on the assignments of jobs to nodes. Jobs that require longer paths for the execution would probably be assigned to nodes with a higher energy level. The scheduling should attempt to distribute the load according to the current energy level(s).

- *Optimistic trajectory locking*: during scheduling, certain trajectory segments are locked in order to avoid the exponential growth of path alternatives. Though the lock usually remains for only a short time, locking strategies, in general, have an impact of the system performance. Since the locking during the uncertainty period is, in general, short, the probability that the number of path alternatives explodes, is probably small. Therefore, the system performance should be compared using locking and using optimistic locking.

- *Interval ActionSuite*: as described in the programming model, an ActionSuite is a container for actions. Invoking the schedule operation requests the scheduler to schedule all contained actions. A contract is only created if the following holds for all actions a in the ActionSuite as: $\forall a \in as \mid sched(a) \wedge exec(a)$. This semantic might be too strict for some purposes. Therefore, a more weakened semantic is required that allows to specify a range $[\delta_1, \delta_2]$ with δ_1 being the minimum number of actions that have to be scheduled in order to create a contract and δ_2 being the requested number of actions that shall be scheduled. This allows the programmer to be more flexible during application development.

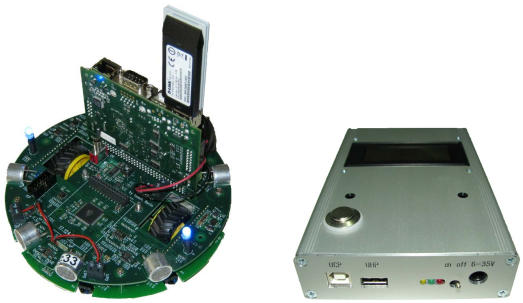
- *Cost model*: As already sketched in Section 4.6.3, a cost model is required and should be investigated. A program causes costs according to its resource usage. It might also be possible to assume that costs depend on the distance or the time a robot has to travel. This way, the unschedule operation is useful since programmers tend to free unused resources in order not to pay for them.

Appendix A

Used Hardware

For performing experiments and showing a proof-of-concept, a testbed has been set up which consists of 40 mobile robots and 24 stationary boards (Figure A.1). The specifications are shown in the following table:

	Robot	Board
CPU & Memory	400 MHz ARM9 CPU 64 MB SDRAM	180 MHz ARM9 CPU 64 MB SDRAM
Sensors	13× infrared 6× ultrasound	
Actuators	2 electric engines (no step motors) 2 RGB LEDs	1 RGB LED



(a) Robot1-0 with PortuxG20

(b) Portux920T

Figure A.1: Hardware

Appendix B

Locating System

The development of the locating system was only necessary in order to perform the evaluation on the testbed and is, therefore, not a part of the contribution of the thesis. The locating system has been developed as joint work with Christoph Brendel.

Figure B.1 shows an example of the cover for one of the robots. The cover has two objects: a two-dimensional bar code which encodes 11 bits and a rectangular bar arranged on the bottom. These two objects are examined and analyzed by the locating system in order to calculate the robot's identification number, the position and its current orientation.

The locating system is based on a hamming code: $Hamming(15, 11)$ with 11 data bits and 4 parity bits resulting in a 15 bit code. Due to the requirement of distinguishing up to 40 mobile robots, 6 data bits ($2^6 = 64$) are sufficient resulting in 64 ids necessary in order to differentiate all robots. The resulting code is an 11 bit code consisting of 7 data bits (including one spare bit) and 4 parity bits.

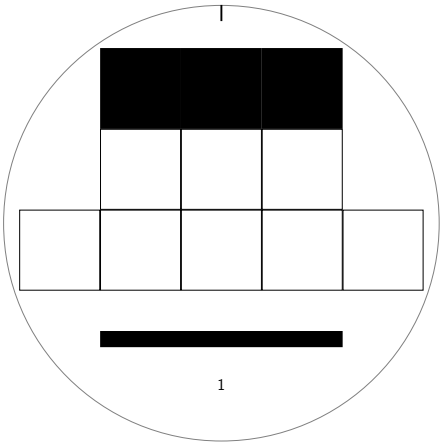


Figure B.1: Cover showing two-dimensional bar code (encoded id 1) and rectangle which is used for re-localization as well as determining orientation.

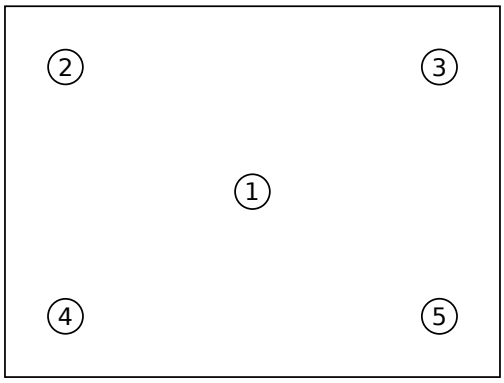


Figure B.2: Testbed with 5 designated positions.

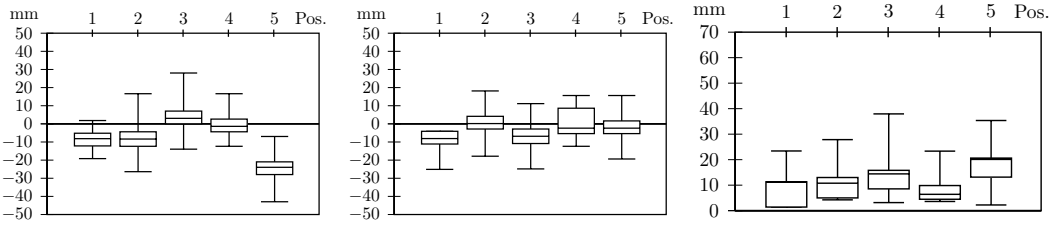
In order to evaluate the accuracy of the locating system a test robot has been placed on various positions on the testbed as shown in Figure B.2. For each of the positions the following values have been computed: The relative deviation from the current (real) position in x- and in y-direction as well as the (positive) Euclidean distance by using the depth image (Figure B.3(a)), the RGB image (Figure B.3(b)) and finally by performing a repositioning (Figure B.3(c)). All experiments have been performed 1000 times at each location which is shown in Figure B.2.

The locating system uses a Microsoft Kinect and the computation is performed in 2 stages: First, the robots are located based on the depth sensor: using this sensor a depth image is created which is transformed into a binary image. A predefined threshold interval regulates the transformation process. All areas (in the depth image) that indicate a height outside the threshold interval are marked black while all areas that indicate a height inside the threshold interval are marked white. The latter ones mark the robots. Since the robots have a circular geometry, OpenCV[1] is used in order to detect circles in the depth image which state their location. This step performs fast and efficient. The accuracy is shown in Figure B.3(a). The maximum error is ≈ 40 mm.

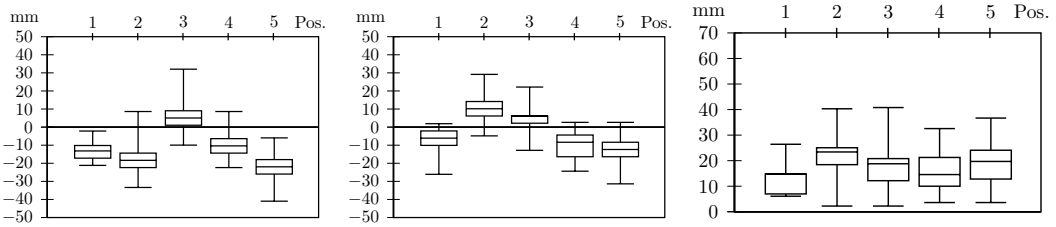
Afterwards, the RGB image is used together with the (more) coarse-grained localization information of the depth image in order to locate the robots in the RGB image; in particular, not the entire RGB image is used for the localization, but only certain sections that is where the robots are. Again, OpenCV is used in order to detect circles in the RGB image. The result of re-localization using the RGB image is shown in Figure B.3(b). Comparing the maximum error of the re-localization with the result of the depth image, there is no noticeable improvement. Hence, re-localization based on circular contour detection is not appropriate.

Nevertheless, the RGB image is required to decode the bar code which is necessary in order to obtain the robot's identification number. The decoding requires to have precise information of the position of the bar code. In order to determine that exact location of

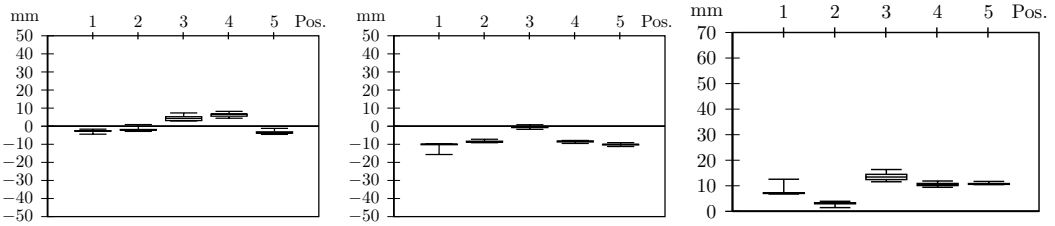
[1]OpenCV (Open Source Computer Vision) is a free library which provides algorithms for computer vision (mainly contours detection in images).



(a) Circle contour detection in the depth image (left = x, middle = y, right = Euclidean distance).



(b) Circle contour detection in the RGB image (left = x, middle = y, right = Euclidean distance).



(c) Rectangle contour detection in the RGB image (left = x, middle = y, right = Euclidean distance).

Figure B.3: Accuracy of calculating position information of the robots (x and y dimension as well as Euclidean distance is considered separately).

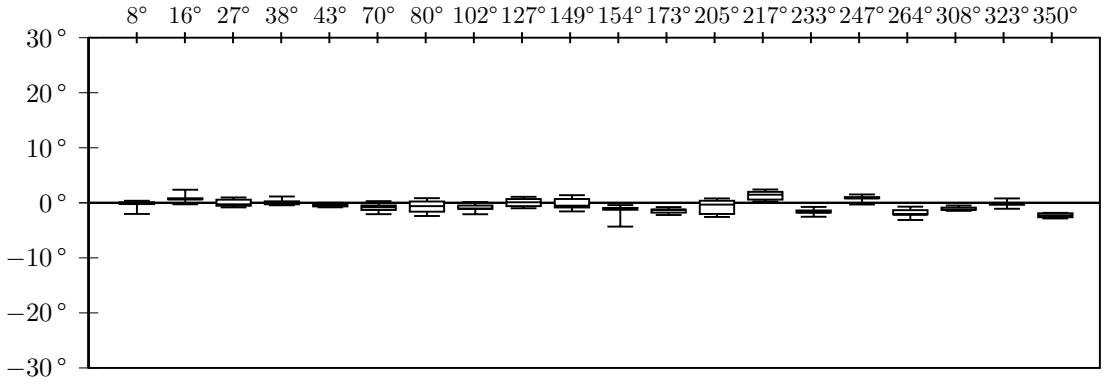


Figure B.4: Accuracy of calculating heading information of the robots based on orientation of the localized rectangle in the RGB image.

the bar code, the robot is localized in the RGB image based on the rectangle below the bar code (using the preliminary location information from the depth image). It turns out that performing a rectangular contour detection is very accurate in OpenCV. This enables to precisely decode the bar code since its position has a fixed offset from the rectangle below. As a side product, since the robot has now been located precisely, the robot's preliminary position (which has been calculated using the depth image) is updated accordingly.

As a result of the second step, the location information could be significantly improved as shown in Figure B.3(c). The maximum error is now $\approx$ 16 mm. Using the precise information of the location of the contour line, it is possible to calculate the orientation of the rectangle. This orientation is also the orientation of the robot. The accuracy is shown in Figure B.4. The maximum error of the orientation is approximately 5°.

Appendix C

Motion Control

The development of the motion control was only necessary in order to perform the evaluation on the testbed and is, therefore, not a part of the contribution of the thesis. The motion control has been published in [77].

In Section 8.4.3, a one-robot scenario has been introduced in which traces were shown that have originated from moving the robot over the testbed. Using the threshold value λ the accuracy of the movement is controllable. Accuracy is defined as the difference between the ideal trajectory (calculated by the scheduler) and the trajectory that the robot actually moves along. Figure C.1 shows the allowed orientation of the robot influenced by λ when moving to a certain destination.

Figure C.2 shows the traces of the robot influenced by λ. In total, eleven test runs have been performed by starting with $\lambda = 1°$ and iteratively increasing the value up to $\lambda = 11°$. As expected, the difference between the ideal trajectory and the actual trajectory that the robot moved along becomes smaller by lowering λ. Setting $\lambda = 1$ (Figure C.2(k)), there is no difference noticeable.

It is desirable to have low λ values, but setting λ has a large impact on other movement characteristics as shown Figure C.3. Figure C.3(a) shows the correlation between λ and the time required for moving along the trajectory. The curves show a non-linear behavior.

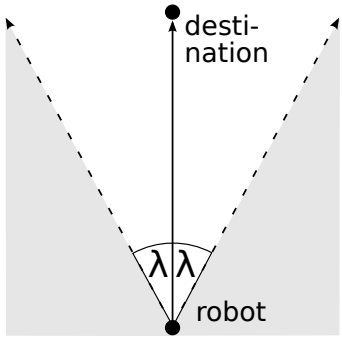
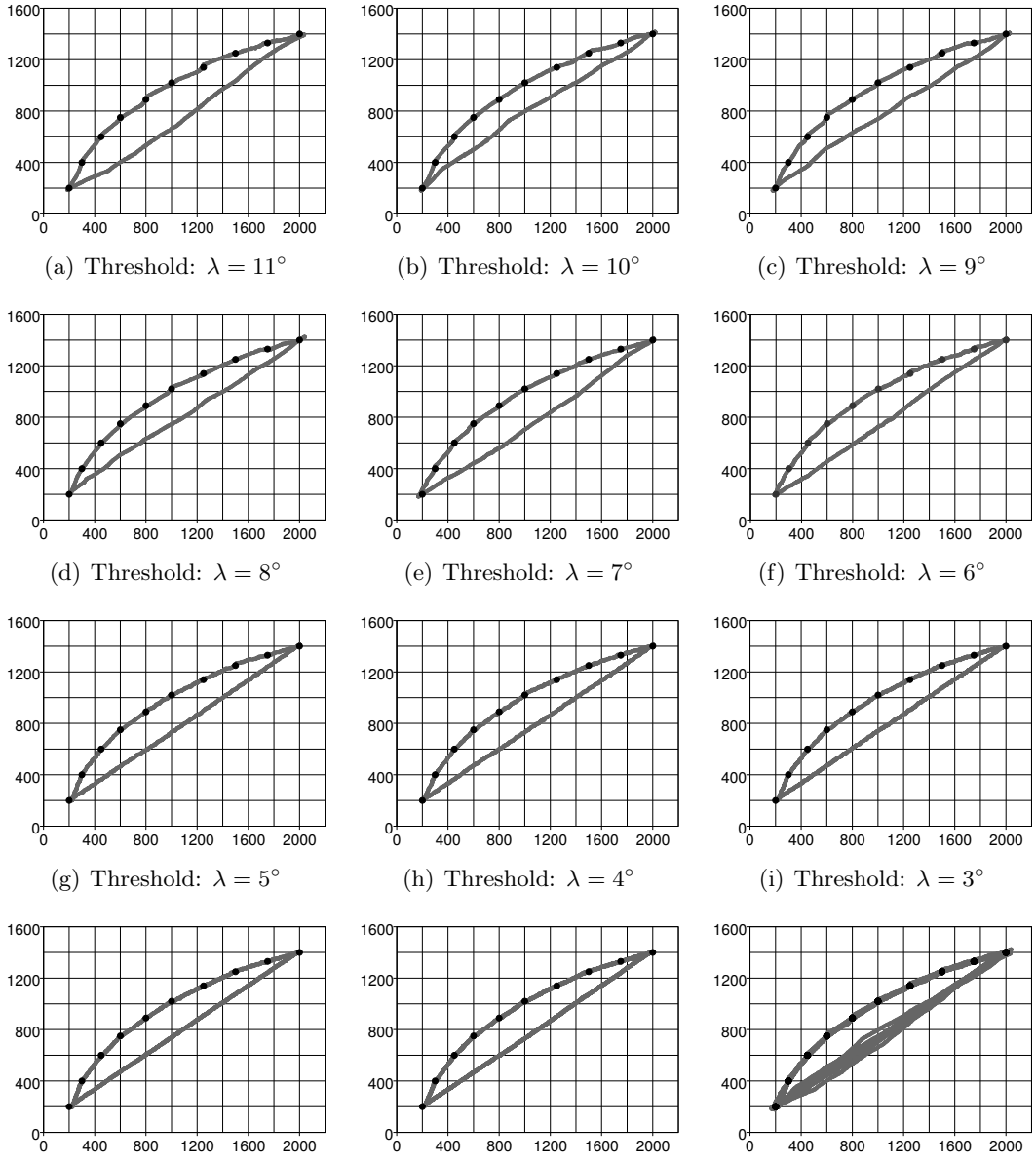


Figure C.1: Threshold value λ.



(a) Threshold: $\lambda = 11°$ (b) Threshold: $\lambda = 10°$ (c) Threshold: $\lambda = 9°$

(d) Threshold: $\lambda = 8°$ (e) Threshold: $\lambda = 7°$ (f) Threshold: $\lambda = 6°$

(g) Threshold: $\lambda = 5°$ (h) Threshold: $\lambda = 4°$ (i) Threshold: $\lambda = 3°$

(j) Threshold: $\lambda = 2°$ (k) Threshold: $\lambda = 1°$ (l) Threshold: $\lambda = 1° - 11°$

Figure C.2: Traces of the robot showing the accuracy of the movement based on different threshold values λ.

Setting $\lambda = 1$ or $\lambda = 2$ results in very long movement times. Analogously, the adopted velocity as depicted in Figure C.3(b) is very low when adjusting small values for λ. It increases by increasing λ. Increasing λ results in a higher inaccuracy of the movement as shown in Figure C.3(c) and C.3(d) in terms of average and maximum deviation from the ideal track.

In order to determine a suitable value for λ, Equations C.1 and C.2 show a trade-off between the normalized time difference and the normalized deviation difference. The result is the normalized efficiency $E(\lambda) \in [0, 1]$. The function $t(\lambda)$ returns the time that is required for moving along a path (line segment or arc) as a function of the threshold value λ. The numerator t_{max}[1] indicates the longest temporal movement and is defined as: $t_{max} := max(\forall\lambda \in \Lambda : t(\lambda))$, with $\Lambda \in \{1, 2, .., 11\}$. The fraction $\frac{t(\lambda)}{t_{max}} \in [0, 1]$ is a function of λ and indicates the normalized time. There are two E functions: E^{max} and E^{avg}.

$$E^{max}(\lambda) = 1 - \left(p_0 \frac{t(\lambda)}{t_{max}} + (1 - p_0) \frac{\delta^{max}(\lambda)}{\delta^{max}_{max}} \right) \tag{C.1}$$

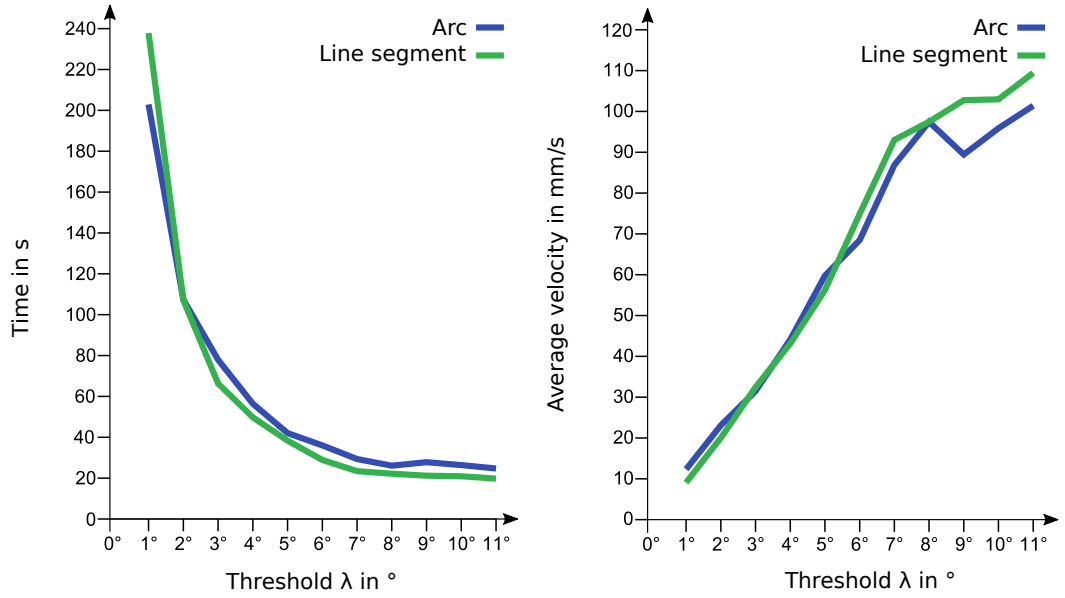
$$E^{avg}(\lambda) = 1 - \left(p_0 \frac{t(\lambda)}{t_{max}} + (1 - p_0) \frac{\delta^{avg}(\lambda)}{\delta^{avg}_{max}} \right) \tag{C.2}$$

E^{max} computes the normalized efficiency based on the maximum deviation from the track. The function $\delta^{max}(\lambda)$ returns the maximum deviation from the track as a function of λ for the line segment as well as for the arc. The numerator δ^{max}_{max}[2] indicates the maximum of all maximum deviations from all possible λ from the track and is defined as: $\delta^{max}_{max} := max(\forall\lambda \in \Lambda : \delta^{max}(\lambda))$, with $\Lambda \in \{1, 2, .., 11\}$. The fraction $\frac{\delta^{max}(\lambda)}{\delta^{max}_{max}} \in [0, 1]$ is a function of λ and indicates the normalized deviation. The parameter p_0 is a weighting factor enabling preferences (time or accuracy).

E^{avg} computes the normalized efficiency based on the average deviation from the track. The function $\delta^{avg}(\lambda)$ returns the average deviation from the track as a function of λ for the line segment as well as for the arc. The numerator δ^{avg}_{max}[3] indicates the maximum of all average deviations from all possible λ from the track and is defined as: $\delta^{avg}_{max} := max(\forall\lambda \in \Lambda : \delta^{avg}(\lambda))$, with $\Lambda \in \{1, 2, .., 11\}$. The fraction $\frac{\delta^{avg}(\lambda)}{\delta^{avg}_{max}} \in [0, 1]$ is a function of λ and indicates the normalized deviation.

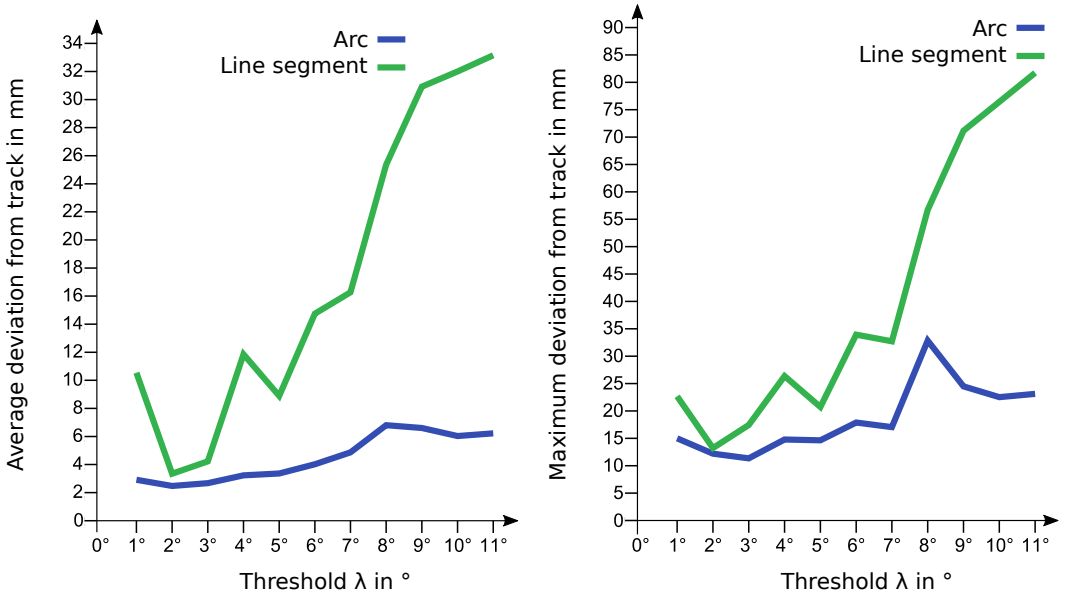
Figure C.4 shows the respective curves when setting $p_0 = 0.5$. Figure C.4(a) shows E^{max} and is based on the maximum deviation δ^{max} from the track while Figure C.4(b) plots E^{avg} and is based on the average deviation δ^{avg} from the track. Both curves have a maximum at $\lambda = 5°$ which states that independently of the applied efficiency function (E^{max} or E^{avg}), both indicate a maximal efficiency using $\lambda = 5°$ showing the best trade-off between time and accuracy.

[1] Two t_{max} values are computed: one for the line segment and one for the arc.
[2] Two δ^{max}_{max} values are computed: one for the line segment and one for the arc.
[3] Two δ^{avg}_{max} values are computed: one for the line segment and one for the arc.



(a) Decreasing time

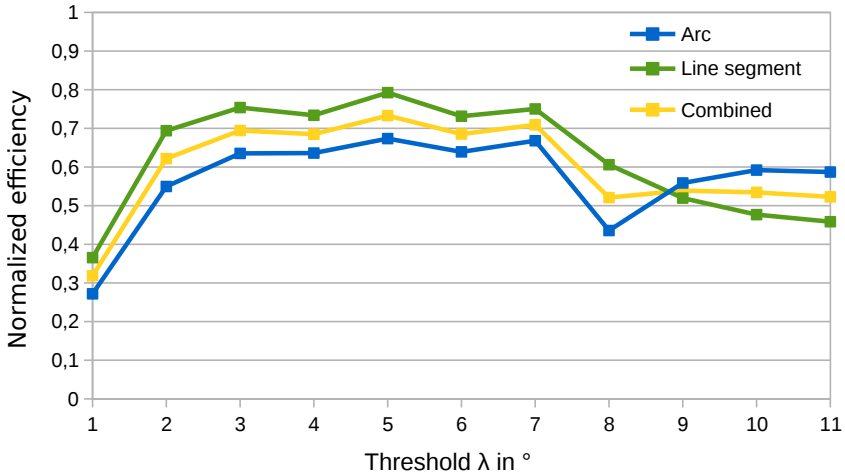
(b) Increasing velocity



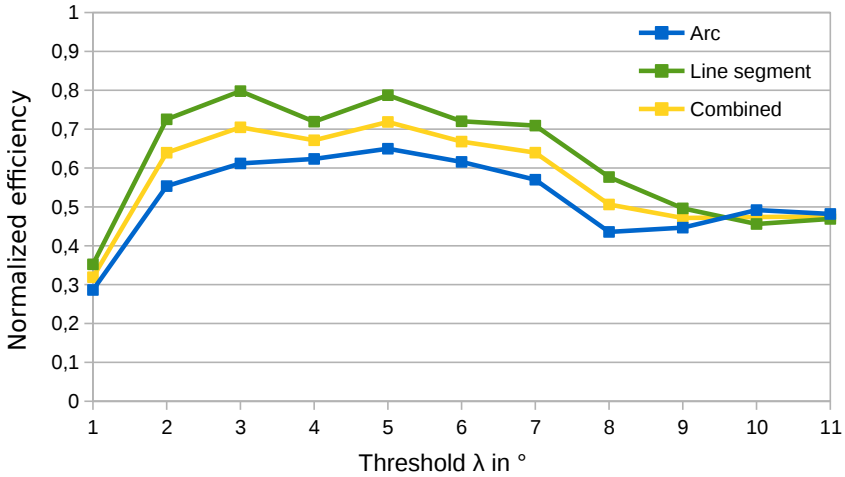
(c) Increasing average deviation

(d) Increasing maximum velocity

Figure C.3: Related movement characteristics as a function of the threshold value.



(a) E^{max}: based on maximum deviation δ^{max} (Equation C.1)



(b) E^{avg}: based on average deviation δ^{avg} (Equation C.2)

Figure C.4: Efficiency E as trade-off between maximum and average deviation from track and required time t.

Bibliography

[1] Rachid Alami, Frédéric Robert, Félix Ingrand, and Sho'ji Suzuki. Multi-Robot Cooperation through Incremental Plan-Merging. In *IEEE International Conference on Robotics and Automation*, pages 2573–2579. IEEE Computer Society, 1995.

[2] Sofia Amador, Steven Okamoto, and Roie Zivan. Dynamic Multi-agent Task Allocation with Spatial and Temporal Constraints. In *Proceedings of the 2014 International Conference on Autonomous Agents and Multi-agent Systems*, AAMAS '14, pages 1495–1496, Richland, SC, 2014. International Foundation for Autonomous Agents and Multiagent Systems.

[3] Jean Le Bail, Rene David, and Hassane Alla. Hybrid petri nets. In *European Control Conference*, pages 1472–1477, Grenoble, 1991.

[4] Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 55–59, New York, NY, USA, 1977. ACM.

[5] Masse Bloomfield. *Mankind in Transition: A View of the Distant Past, the Present, and the Far Future.* Masefield Books, 1993.

[6] Masse Bloomfield. *The Automated Society.* Masefield Books, 1995.

[7] Eric Bonabeau. Editor's Introduction: Stigmergy. *Artificial Life*, 5(2):95–96, Apr 1999.

[8] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *From Natural to Artificial Swarm Intelligence.* Oxford University Press, 1999.

[9] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems.* Oxford University Press, Inc., New York, NY, USA, 1999.

[10] Raimon Casanova, Angel Dieguez, Andreu Sanuy, Anna Arbat, Oscar Alonso, Joan Canals, Manel Puig, and Josep Samitier. Enabling swarm behavior in mm3-sized robots with specific designed integrated electronics. In *IROS*, pages 3797–3802. IEEE, 2007.

[11] CBS Interactive Inc. Amazon unveils futuristic plan: Delivery by drone, December 2013. http://www.cbsnews.com/news/amazon-unveils-futuristic-plan-delivery-by-drone/.

[12] Karthik Dantu, Bryan Kate, Jason Waterman, Peter Bailis, and Matt Welsh. Programming Micro-aerial Vehicle Swarms with Karma. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, SenSys '11, pages 121–134, New York, NY, USA, 2011. ACM.

[13] Rene David and Hassane Alla. Continuous petri nets. In *8th European Workshop on Application and Theory of Petri Nets*, Zaragoza, 1987.

[14] Rene David and Hassane Alla. On hybrid petri nets. *Discrete Event Dynamic Systems*, 11(1-2):9–40, January 2001.

[15] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[16] Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66, New York, NY, USA, 1982. Springer-Verlag.

[17] Marco Dorigo, Dario Floreano, Luca Maria Gambardella, Francesco Mondada, Stefano Nolfi, Marco Birattari, Anders Christensen, Nithin Mathews, Rehan O'Grady, and Vito Trianni. Swarmanoid: A novel concept for the study of heterogeneous robotic swarms. *IEEE Robotics Automation Magazine*, 20(4):60 – 71, December 2013.

[18] Marco Dorigo, Elio Tuci, Roderich Groß, Vito Trianni, Thomas Halva Labella, Shervin Nouyan, Christos Ampatzis, Jean-Louis Deneubourg, Gianluca Baldassarre, Stefano Nolfi, Francesco Mondada, Dario Floreano, and Luca Maria Gambardella. The SWARM-BOTS Project. In Erol Sahin and William M. Spears, editors, *Swarm Robotics*, volume 3342 of *Lecture Notes in Computer Science*, pages 31–44. Springer, 2004.

[19] Michael Erdmann and Tomas Lozano-Perez. On multiple moving objects. *Algorithmica*, 2:1419–1424, 1986.

[20] Erico Guizzo. World robot population reaches 8.6 million. IEEE Spectrum, April 2010. http://spectrum.ieee.org/automaton/robotics/industrial-robots/041410-world-robot-population.

[21] Ramon Estaña, Marc Szymanski, Lutz Winkler, and Heinz Wörn. I-Swarm. In *Jahresbericht des Instituts für Prozessrechentechnik, Automation und Robotik*, pages 16–17, 2008. http://rob.ipr.kit.edu/downloads/Forsch_JB_2008.pdf.

[22] Thiery Fraichard and Christian Laugier. Path-Velocity Decomposition Revisited and Applied to Dynamic Trajectory Planning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 40–45, Atlanta, GA (USA), May 1993.

[23] David Gay, Philip Levis, Robert von Behren, et al. The nesC language: A holistic approach to networked embedded systems. *SIGPLAN Not.*, 38(5):1–11, May 2003.

[24] Matthew Gombolay, Ronald Wilcox, and Julie Shah. Fast scheduling of multi-robot teams with temporospatial constraints. In *Proceedings of Robotics: Science and Systems*, Berlin, Germany, June 2013.

[25] Daniel Graff, Helge Parzyjegla, Jan Richling, and Matthias Werner. Verteilte aktive Objekte für verteilte mobile Systeme. In Axel Küpper and Jörg Roth, editors, *Tagungsband zum 7. GI/ITG KuVS-Fachgespräch "Ortsbezogene Anwendungen und Dienste"*, pages 55–62. Logos Verlag Berlin GmbH, September 2011.

[26] Daniel Graff, Jan Richling, Tammo M. Stupp, and Matthias Werner. Context-Aware Annotations for Distributed Mobile Applications. In Dimitrios Soudris Wolfgang Karl, editor, *ARCS'11 Workshop Proceedings: Second Workshop on Context-Systems Design, Evaluation and Optimisation (CoSDEO 2011)*, pages 357–366. VDE, February 2011.

[27] Daniel Graff, Jan Richling, Tammo M. Stupp, and Matthias Werner. Distributed Active Objects – A Systemic Approach to Distributed Mobile Applications. In Roy Sterrit, editor, *8th IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems*, pages 10–19. IEEE Computer Society, April 2011.

[28] Daniel Graff, Jan Richling, and Matthias Werner. Concepts for Swarm System Software. In *ESWeek: First International Workshop on the Swarm at the Edge of the Cloud (SEC 2013)*, September 2013.

[29] Daniel Graff, Jan Richling, and Matthias Werner. Modeling Group Scheduling Problems in Space and Time by Timed Petri Nets. *Fundamenta Informaticae*, 122(4):297–313, January 2013.

[30] Daniel Graff, Jan Richling, and Matthias Werner. Programming and Managing the Swarm – An Operating System for an Emerging System of Mobile Devices. In Kai Lin, Heng Qi, Keqiu Li, Ivan Stojmenovic, Albert Zomaya, Hongyi Wu, Song Guo, and Symeon Papavassiliou, editors, *9th IEEE International Conference on Mobile Ad-hoc and Sensor Networks (MSN 2013)*, pages 9–16. IEEE Computer Society, December 2013.

[31] Daniel Graff, Jan Richling, and Matthias Werner. jSwarm: Distributed Coordination in Robot Swarms. In *CPSWeek: Robotic Sensor Networks (RSN 2014)*, April 2014.

[32] Daniel Graff, Daniel Röhrig, Rico Jasper, Helge Parzyjegla, Gero Mühl, and Jan Rabaey. Operating System Support for Mobile Robot Swarms. In *CPSWeek: Second International Workshop on the Swarm at the Edge of the Cloud*, Seattle, Washington (USA), April 2015.

[33] Daniel Graff, Daniel Röhrig, and Reinhardt Karnapke. On the Need of Systemic Support for Spatio-Temporal Programming of Mobile Robot Swarms. In *11th IEEE International Conference on Mobile Ad-hoc and Sensor Networks (MSN 2015)*. IEEE Computer Society, December 2015.

[34] Daniel Graff, Daniel Röhrig, and Reinhardt Karnapke. Systemic Support for Transaction-Based Spatial-Temporal Programming of Mobile Robot Swarms. In *40th IEEE Conference on Local Computer Networks Workshops (LCN Workshops)*, pages 730–733, Clearwater Beach, Florida (USA), October 2015. IEEE.

[35] Daniel Graff, Tammo M. Stupp, Jan Richling, and Matthias Werner. Using Timed Petri Nets to Model Spatial-temporal Group Scheduling Problems. In Marcin Szczuka, Ludwik Czaja, Andrzej Skowron, and Magdalena Kacprzak, editors, *Concurrency, Specification & Programming (CS&P) 2011*, pages 160–168, September 2011.

[36] Daniel Graff, Matthias Werner, Helge Parzyjegla, Jan Richling, and Gero Mühl. An Object-Oriented and Context-Aware Approach for Distributed Mobile Applications. In Michael Beigl and Francisco J. Cazorla-Almeida, editors, *ARCS'10 Workshops Proceedings: First Workshop on Context-Systems Design, Evaluation and Optimisation (CoSDEO 2010)*, page 191–200. VDE, February 2010.

[37] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, UK, 1978. Springer-Verlag.

[38] Brian K. Hall and Benedikt Hallgrímsson. *Strickberger's Evolution*. Jones & Bartlett Learning, LLC, 2011.

[39] Heiko Hamann and Heinz Wörn. An Analytical and Spatial Model of Foraging in a Swarm of Robots. *LNCS 4433, Swarm Robotics - 2nd SAB 2006 International Workshop, Rome, Italy*, pages 43–55, 2007.

[40] Kamal Kant and Steven W. Zucker. Toward Efficient Trajectory Planning: The Path-Velocity Decomposition. *The International Journal of Robotics Research*, 5(3):72–89, September 1986.

[41] Idit Keidar and Danny Dolev. Increasing the resilience of distributed and replicated database systems. *Journal of Computer and System Sciences (JCSS)*, 57(3):309–224, 1998.

[42] Serge Kernbach. Swarmrobot.org - Open-hardware Microrobotic Project for Large-scale Artificial Swarms. *CoRR*, abs/1110.5762, 2011.

[43] Serge Kernbach and Olga Kernbach. Collective energy homeostasis in a large-scale microrobotic swarm. *Robotics and Autonomous Systems*, 59(12):1090–1101, 2011.

[44] Serge Kernbach, Eugen Meister, Florian Schlachter, et al. Symbiotic robot organisms: REPLICATOR and SYMBRION projects. In *Proceedings of the 8th Workshop on Performance Metrics for Intelligent Systems*, PerMIS '08, pages 62–69, New York, NY, USA, 2008. ACM.

[45] Serge Kernbach, Oliver Scholz, Kanako Harada, Sergej Popesku, Jens Liedke, Raja Humza, Wenguo Liu, Fabio Caparrelli, Jaouhar Jemai, Jiri Havlik, Eugen Meister, and Paul Levi. Multi-robot organisms: State of the art. *CoRR*, abs/1108.5543, 2011.

[46] Nupur Kothari, Ramakrishna Gummadi, Todd Millstein, et al. Reliable and efficient programming abstractions for wireless sensor networks. *SIGPLAN Not.*, 42(6):200–210, June 2007.

[47] Edward A. Lee. Cyber-physical systems – are computing foundations adequate? In *Position Paper for NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap*, October 2006.

[48] Edward A. Lee. Cyber Physical Systems: Design Challenges. Technical Report UCB/EECS-2008-8, EECS Department, University of California, Berkeley, Jan 2008.

[49] Edward A. Lee, John D. Kubiatowicz, Jan M. Rabaey, et al. The TerraSwarm Research Center (TSRC) (A White Paper). Technical Report UCB/EECS-2012-207, EECS Department, University of California, Berkeley, Nov 2012.

[50] Edward A. Lee, Jan Rabaey, David Blaauw, Kevin Fu, Carlos Guestrin, Bjorn Hartmann, Roozbeh Jafari, Doug Jones, John Kubiatowicz, Vijay Kumar, Rahul Mangharam, Richard Murray, George Pappas, Kris Pister, Anthony Rowe, Alberto Sangiovanni-Vincentelli, Sanjit A. Seshia, Tajana Simunic Rosing, Ben Taskar, John Wawrzynek, and David Wessel. The swarm at the edge of the cloud. *Design & Test, IEEE*, 31(3):1–13, June 2014.

[51] Tomás Lozano-Pérez. Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, C-32:108–120, 1983.

[52] Marc Szymanski, Lutz Winkler, Davide Laneri, Florian Schlachter, Anne C. van Rossum, Thomas Schmickl, and Ronald Thenius. SymbricatorRTOS: A Flexible and Dynamic Framework for Bio-Inspired Robot Control Systems and Evolution. In IEEE Press, editor, *IEEE Congress on Evolutionary Computation (IEEE CEC-2009), Trondheim, Norway, May 18-21*, pages 3314–3321, 2009.

[53] Leslie Marsh and Christian Onof. Stigmergic epistemology, stigmergic cognition. *Cogn. Syst. Res.*, 9(1-2):136–149, March 2008.

[54] James McLurkin. *Stupid Robot Tricks: A Behavior-Based Distributed Algorithm Library for Programming Swarms of Robots.* S.M. thesis, Massachusetts Institute of Technology, 2004.

[55] James McLurkin and Daniel Yamins. Dynamic task assignment in robot swarms. In *Robotics: Science and Systems Conference*, Cambridge, MA, USA, 2005.

[56] James Dwight Mclurkin, IV. *Analysis and Implementation of Distributed Algorithms for Multi-robot Systems.* PhD thesis, Cambridge, MA, USA, 2008. AAI0821012.

[57] Philip Merlin. *A Study of the Recoverability of Communikation Protocols.* PhD thesis, University of California, Irvine, CA, USA, 1974.

[58] Francesco Mondada, André Guignard, Michael Bonani, Daniel Bär, Michel Lauria, and Dario Floreano. Swarm-bot: from concept to implementation. In *IROS*, pages 1626–1631. IEEE, 2003.

[59] Francesco Mondada, André Guignard, Alexandre Colot, Dario Floreano, Jean-Louis Deneubourg, Luca Gambardella, Stefano Nolfi, and Marco Dorigo. Swarm-bot: A new concept of robust all-terrain mobile robotic system, 2002.

[60] Francesco Mondada, Giovanni C. Pettinaro, André Guignard, Ivo W. Kwee, Dario Floreano, Jean-Louis Deneubourg, Stefano Nolfi, Luca Maria Gambardella, and Marco Dorigo. Swarm-bot: A new distributed robotic concept. *Auton. Robots*, 17(2-3):193–221, 2004.

[61] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965.

[62] Luca Mottola and Gian Pietro Picco. Logical neighborhoods: A programming abstraction for wireless sensor networks. In Phillip B. Gibbons, Tarek Abdelzaher, James Aspnes, and Ramesh Rao, editors, *Distributed Computing in Sensor Systems*, volume 4026 of *Lecture Notes in Computer Science*, pages 150–168. Springer Berlin Heidelberg, 2006.

[63] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks with logical neighborhoods. In *Proceedings of the First International Conference on Integrated Internet Ad Hoc and Sensor Networks*, InterSense '06, New York, NY, USA, 2006. ACM.

[64] Luca Mottola and Gian Pietro Picco. Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art. *ACM Comput. Surv.*, 43(3):19:1–19:51, April 2011.

[65] Yang Ni, Ulrich Kremer, Adrian Stere, et al. Programming ad-hoc networks of mobile and resource-constrained devices. *SIGPLAN Not.*, 40(6):249–260, June 2005.

[66] Shervin Nouyan and Marco Dorigo. Chain Based Path Formation in Swarms of Robots. In Marco Dorigo, Luca Maria Gambardella, Mauro Birattari, Alcherio Martinoli, Riccardo Poli, and Thomas Stützle, editors, *ANTS Workshop*, volume 4150 of *Lecture Notes in Computer Science*, pages 120–131. Springer, 2006.

[67] Patrick A. O'Donnell and Tomás Lozano-Pérez. Deadlock-free and collision-free coordination of two robot manipulators. In *IEEE Robotics and Automation Conference*, pages 484–489, 1989.

[68] Helge Parzyjegla, Arnd Schröter, Anselm Busse, Daniel Graff, Alexej Schepeljanski, Jan Richling, Matthias Werner, and Gero Mühl. Rebeca - eine autonome Publish/-Subscribe Middleware. *Praxis der Informationsverarbeitung und Kommunikation (PIK)*, 2011.

[69] Helge Parzyjegla, Arnd Schröter, Daniel Graff, Anselm Busse, Alexej Schepeljanski, Jan Richling, Matthias Werner, and Gero Mühl. Autonomy Features and Feature Composition in REBECA. In *ICAC'11*. ACM, June 2011.

[70] Eloi Pereira, Pedro Marques, Clemens Krainer, Christoph M. Kirsch, Jose Morgado, and Raja Sengupta. A Networked Robotic System and its Use in an Oil Spill Monitoring Exercise. In *Swarm at the Edge of the Cloud Workshop (ESWeek'13)*, volume 2, pages 1–2, Montreal, QC, Canada, 2013.

[71] Eloi Pereira, Camille Potiron, Chirstoph M. Kirsch, and Raja Sengupta. Modeling and controlling the structure of heterogeneous mobile robotic systems: A bigactor approach. In *2013 IEEE International Systems Conference (SysCon)*, pages 442–447, Orlando, FL, USA, April 2013. IEEE.

[72] Peter H. Starke, Humboldt-Universität zu Berlin, Institut für Informatik, Lehrstuhl für Automaten- und Systemtheorie. Integrated Net Analyzer INA. `http://www2.informatik.hu-berlin.de/~starke/ina.html`.

[73] Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, Universität Bonn, Institut für Instrumentelle Mathematik, Bonn, 1962.

[74] Louchka Popova-Zeugmann and Matthias Werner. Extreme runtimes of schedules modelled by time petri nets. *Fundamenta Informaticae*, 67:163–174, 2005.

[75] Louchka Popova-Zeugmann, Matthias Werner, and Jan Richling. Using state equation to prove non-reachability in timed petrinets. *Fundamenta Informaticae*, 55:187–202, 2002.

[76] Jan M. Rabaey. The Human Intranet–Where Swarms and Humans Meet. *Pervasive Computing, IEEE*, 14(1):78–83, January 2015.

[77] Pavel Rabov and Daniel Graff. Ein modulares Framework für adaptive Bewegungssteuerungen für mobile Roboter zur Ausführung ortsbezogener Anwendungen. In

Jörg Roth Gerald Eichler, Volkmar Schau, editor, *11.GI/ITG KuVS-Fachgespräch. Ortsbezogene Anwendungen und Dienste*. Logos Verlag Berlin, September 2014.

[78] Marc Raibert. BigDog, the Rough-Terrain Quadruped Robot. In Myung J. Chung, editor, *Proceedings of the 17th IFAC World Congress, 2008*, volume 17.

[79] Chander Ramchandani. Analysis of asynchronous concurrent systems by Timed Petri Nets. Project MAC-TR 120, MIT, Massachusetts Institute of Technology, Cambridge, MA, USA, February 1974.

[80] Michael Rubenstein, Christian Ahler, Nick Hoff, Adrian Cabrera, and Radhika Nagpal. Kilobot: A low cost robot with scalable operations designed for collective behaviors. *Robotics and Autonomous Systems*, 62(7):966 – 975, 2014. Reconfigurable Modular Robotics.

[81] Michael Rubenstein, Christian Ahler, and Radhika Nagpal. Kilobot: A low cost scalable robot system for collective behaviors. In *ICRA*, pages 3293–3298. IEEE, 2012.

[82] Michael Rubenstein, Adrian Cabrera, Justin Werfel, Golnaz Habibi, James McLurkin, and Radhika Nagpal. Collective transport of complex objects by simple robots: Theory and experiments. In *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems*, AAMAS '13, pages 47–54, Richland, SC, 2013. International Foundation for Autonomous Agents and Multiagent Systems.

[83] Michael Rubenstein, Alejandro Cornejo, and Radhika Nagpal. Programmable self-assembly in a thousand-robot swarm. *Science*, 345:795–799, August 2014.

[84] Michael Rubenstein and Radhika Nagpal. Kilobot: A Robotic Module for Demonstrating Behaviors in a Large Scale (2^{10} Units) Collective. In Radhika Nagpal Kasper Stoy and Wei-Min Shen, editors, *IEEE 2010 International Conference on Robotics and Automation Workshop, Modular Robotics: State of the Art*, pages 47–51, Anchorage, Alaska, May 2010.

[85] Michael Rubenstein and Wei-Min Shen. Scalable self-assembly and self-repair in a collective of robots. In *IROS*, pages 1484–1489. IEEE, 2009.

[86] Michael Rubenstein and Wei-Min Shen. Automatic Scalable Size Selection for the Shape of a Distributed Robotic Collective. Taipei, Taiwan, October 2010.

[87] Erol Sahin, Thomas H. Labella, Vito Trianni, Jean louis Deneubourg, Philip Rasse, Dario Floreano, Luca Gambardella, Francesco Mondada, Stefano Nolfi, and Marco Dorigo. SWARM-BOT: Pattern Formation in a Swarm Of Self-Assembling Mobile Robots. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Hammamet*, pages 6–9. IEEE Press, 2002.

[88] Samuel Gibbs. What is boston dynamics and why does google want robots? The Guardian, December 2013. http://www.theguardian.com/technology/2013/dec/17/google-boston-dynamics-robots-atlas-bigdog-cheetah.

[89] Robert R. Schaller. Moore's law: Past, present, and future. *IEEE Spectr.*, 34(6):52–59, June 1997.

[90] Jörg Seyfried, Marc Szymanski, Natalie Bender, Ramon Estaña, Michael Thiel, and Heinz Wörn. The I-SWARM Project: Intelligent Small World Autonomous Robots for Micro-manipulation. In Erol Sahin and William M. Spears, editors, *Swarm Robotics*, volume 3342 of *Lecture Notes in Computer Science*, pages 70–83. Springer, 2004.

[91] Thierry Siméon, Stéphane Leroy, and Jean-Paul Laumond. Path coordination for multiple mobile robots: a resolution-complete algorithm. *IEEE T. Robotics and Automation*, 18(1):42–49, 2002.

[92] Dale Skeen. A Quorum-Based Commit Protocol. In *6th Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 69–80, February 1982.

[93] Tammo M. Stupp, Daniel Graff, Anselm Busse, and Jan Richling. Ein Taskmodell für Raum-Zeit-Scheduling. In *Tagungsband zum 9. GI/ITG KuVS-Fachgespräch*, September 2012.

[94] Ryo Sugihara and Rajesh K. Gupta. Programming Models for Sensor Networks: A Survey. *ACM Trans. Sen. Netw.*, 4(2):8:1–8:29, April 2008.

[95] Ying Tan, Mehmet C. Vuran, and Steve Goddard. Spatio-temporal event model for cyber-physical systems. In *Distributed Computing Systems Workshops, 2009. ICDCS Workshops '09. 29th IEEE International Conference on*, pages 44 –50, 2009.

[96] Ying Tan and Zhong yang Zheng. Research advance in swarm robotics. *Defence Technology*, 9(1):18 – 39, 2013.

[97] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[98] The Economist. The end of moore's law, April 2015. http://www.economist.com/blogs/economist-explains/2015/04/economist-explains-17.

[99] The Radicati Group, Inc. Mobile statistics report, 2014-2018, February 2014. http://www.radicati.com/wp/wp-content/uploads/2014/01/Mobile-Statistics-Report-2014-2018-Executive-Summary.pdf.

[100] University of Hamburg, Faculty of Mathematics, Informatics und Natural Sciences Department of Informatics, TGI Group. Petri Nets World. http://www.informatik.uni-hamburg.de/TGI/PetriNets.

[101] Mikko A. Uusitalo. Global Vision for the Future Wireless World from the WWRF. In *IEEE Vehicular Technology Magazine*, volume 1, pages 4–8, 2006.

[102] Charles W. Warren. Multiple Robot Path Coordination Using Artificial Potential Fields. In *IEEE International Conference on Robotics and Automation*, pages 500–505, Cincinnati, OH (USA), 1990.

[103] Mark Weiser. Human-computer interaction. chapter The Computer for the 21st Century, pages 933–940. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995.

[104] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[105] Heinz Woern, Marc Szymanski, and Joerg Seyfried. The I-SWARM project. In *The 15th IEEE International Symposium on Robot and Human Interactive Communication*, pages 492–496. IEEE, September 2006.

[106] Zhijiao Xiao and Zhong Ming. A method of workflow scheduling based on colored petri nets. *Data Knowl. Eng.*, 70:230–247, February 2011.

[107] Shang-Tae Yee and Jose A. Ventura. A dynamic programming algorithm to determine optimal assembly sequences using petri nets. *International Journal of Industrial Engineering - Theory, Applications and Practice*, 6(1):27–37, 1999.

[108] Hubert Zimmermann. Innovations in internetworking. chapter OSI Reference Model—The ISO Model of Architecture for Open Systems Interconnection, pages 2–9. Artech House, Inc., Norwood, MA, USA, 1988.