Modeling Robotic Systems with Activity Flow Graphs

Dissertation

der Mathematisch-Naturwissenschaftlichen Fakultät

der Eberhard Karls Universität Tübingen

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

(Dr. rer. nat.)

vorgelegt von

M.Sc. Sebastian Buck
aus Reutlingen

Tübingen
2017

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	09.05.2018
Dekan:	Prof. Dr. Wolfgang Rosenstiel
1. Berichterstatter:	Prof. Dr. rer. nat. Andreas Zell
2. Berichterstatter:	Prof. Dr. rer. nat. Andreas Schilling
3. Berichterstatter:	Prof. Dr. rer. nat. Karsten Berns

For my family and friends.

Abstract

Autonomous robotic systems are becoming increasingly common in our society, with research efforts towards automated goods transportation, service robots and autonomous cars. These complex systems have to solve many different problems in order to function robustly. Two especially important areas of interest are perception and high level control. Intelligent systems have to perceive their surroundings in order to facilitate autonomy. With an understanding of the environment, they then can make their own decisions based on high level control policies defined by the developers.

Robotic systems differ drastically in their sensory capabilities, their computational power, and their designated tasks. When developing algorithms, however, we need to have a common modeling framework that enables us to generalize and re-use existing solutions. A modular approach, which is coherent across different platforms, also allows faster prototyping of new systems. In this dissertation we develop a modeling framework based on data flow that achieves this goal.

We first extend the existing *Synchronous Data Flow* (SDF) model and combine it with reactive programming ideas and finite-state machines. Together, these existing frameworks enable us to model many aspects of complex robotic systems. We apply this model to a robot in a warehouse scenario to demonstrate the viability of the approach.

Using three disjoint formalisms to model a robotic system has many downsides. In a first unification step we merge SDF and reactive programming into *Hybrid Flow Graphs* (HFGs), where we explicitly model synchronous and asynchronous data flow. We then apply the HFG model to the perception system of an autonomous transportation robot.

In a last step, we eliminate the need for separate finite-state machines by introducing the concept of activity into the data flow. We therefore unify the different aspects into a single and coherent framework which we call *Activity Flow Graphs* (AFGs). The flow of activity enables us to model high level state directly in the data flow graph. The result is a single computation graph that can express both perception and high level control aspects of any robotic system. We then demonstrate this with multiple high level robotic system models.

Finally, we make use of the uniform AFG model to provide a single graphical user interface that allows a developer to rapidly prototype complete robotic systems. Since all aspects of a robot can be implemented using the same theoretical framework, there is no need to switch between different paradigms. The user interface is designed to give immediate feedback, which speeds up prototyping, testing and evaluation, as well as debugging when working with real robots.

Kurzfassung

Autonome Roboter werden zunehmend zu einem wichtigen Bestandteil unserer Gesellschaft, in Bereichen wie dem automatisierten Gütertransport, in der Servicerobotik und bei autonomen Automobilen. Diese komplexen Systeme müssen viele Problem lösen, um robust zu funktionieren. Zwei sehr wichtige Anwendungsfelder sind die Umgebungswahrnehmung und die Ablaufplanung. Intelligente Systeme müssen ihre Umgebung wahrnehmen, um autonom agieren zu können. Mit einem Verständnis der Umwelt können sie Entscheidungen treffen, welche auf abstrakten Richtlinien der Entwickler basieren.

Verschiedene Roboter weichen stark in ihren sensorischen Fähigkeiten, in ihrer Rechenleistung und in ihren zu lösenden Aufgaben voneinander ab. Bei der Entwicklung von Algorithmen wird jedoch ein einheitliches Modellierungssystem benötigt, welches die Wiederverwendung von existierenden Lösungen erlaubt. Ein modulares System, welches über mehrere Plattformen hinweg genutzt werden kann, ermöglicht eine schnellere Entwicklung von neuen Systemen. In dieser Dissertation wird ein auf Datenfluss basierendes Modell entwickelt, welches diese Anforderungen erfüllt.

Zuerst wird das existierende *Synchronous Data Flow* (SDF) Modell erweitert und mit Elementen von reaktiver Programmierung und endlichen Zustandsautomaten kombiniert. Zusammen können so viele Aspekte von Robotern modelliert werden. Das Modell wird auf einen Roboter in einem Warenhausszenario angewandt, um den Ansatz zu validieren.

Drei verschiedene Formalismen zur Modellierung von Robotern zu verwenden hat viele Nachteile. In einem ersten Vereinigungsschritt werden SDF und reaktive Programmierung zu *hybriden Flussgraphen* (HFG) kombiniert, bei denen synchroner und asynchroner Datenfluss explizit modelliert werden. Dann wird das HFG-Modell auf die Wahrnehmungsmodule eines autonomen Transportsystems angewandt.

Anschließend wird der Bedarf eines Zustandsautomaten beseitigt, indem das Konzept der Aktivität in den Datenfluss eingeführt wird. Dadurch werden alle Aspekte in einem einzigen, schlüssigen System vereinigt, welches *Aktivitätsflussgraph* (AFG) genannt wird. Der Aktivitätsfluss ermöglicht es, den höheren Systemzustand direkt im Datenflussgraphen zu modellieren. Als Ergebnis erhalten wir einen einzigen Berechnungsgraphen, der sowohl zur Beschreibung der Umgebungswahrnehmung als auch zur Kontrolle der höheren Abläufe benutzt werden kann. Dies wird anhand mehrerer Robotersysteme demonstriert.

Eine graphische Benutzerschnittstelle wird bereitgestellt, welche von dem einheitlichen Modell Gebrauch macht, um ein schnelles Prototyping von Robotern zu ermöglichen. Da alle Aspekte mit demselben System modelliert werden, muss nicht zwischen verschiedenen Paradigmen gewechselt werden. Die Nutzerschnittstelle erleichtert Entwicklung, Test und Validierung von Algorithmen sowie das Auffinden von Fehlern bei echten Robotern.

Acknowledgments

First and foremost I want to thank Prof. Andreas Zell for his supervision of this dissertation and the preceding bachelor and master theses. Thank you for all the support and for funding my research over the years. I am especially grateful for the trust I have enjoyed for the research that exceeded the PATSY project. Additionally I want to thank Prof. Andreas Schilling for agreeing to be the second assessor of this thesis, as well as Prof. Oliver Bringmann and Prof. Klaus-Jörn Lange for agreeing to be examiners.

I thank my former office-roommate Daniel Dube and project partners Karsten Bohlmann and Gerald Rauscher for their good cooperation and constant helpfulness. Many thanks to Sebastian Scherer and Julian Jordan, with whom it was always a pleasure to supervise lecture tutorials together. Thank you to Alexander Dörr for the good times at lunch and in coffee breaks, as well as for reviewing this manuscript.

A special gratitude to Vita Serbakova and Klaus Beyreuther for their constant support.

Also a special mention to the students that have worked with us at various robot competitions and have helped to test and improve the CS::APEX framework and the underlying AFG model. Especially Alina Kloss, Jan Leininger, Eugen Ruff and Felix Widmaier, who have helped the Attempto Tübingen team to win the second place at the SICK Robot Day 2014. Additional thanks to Matthias Reisenauer and the other participants of the SpaceBotCamp 2015.

I am extraordinarily grateful to my longtime office-roommate Goran Huskić, for always helping out, unfailingly spreading positivity and brightening the mood. Thank you for all the fruitful discussions and ideas for our common research projects. Likewise I owe a lot to my flatmates and close friends Richard Hanten and Adrian Zwiener, who were indispensable for the real-world application of the ideas developed in this dissertation. Thank you for all you have done, the many discussions over coffee and the creative brain-storming sessions.

I am most grateful to my parents Heiderose and Manfred, and my sister Julia, who have all provided me with moral and emotional support in my life. I am also grateful to my other family members and friends who have supported me along the way.

Thank you all for your encouragement!

Contents

Chapter 1

Introduction

1.1 Modeling Robotic Systems

Robotic systems are becoming increasingly visible in many domains of applications. There are many types of robots that feature different levels of mobility, autonomy and universality. The earliest economically successful robots were stationary manipulators in factory lines, which had only a limited perception of their environment. They are essential in our production facilities and are becoming increasingly flexible.

On the other side there are mobile robots that are not bound to a single workspace. By being able to move, they can be used in many more areas than manipulators, but this also renders their implementation more challenging. It is straightforward to provide a safe working environment for a stationary robot, since the area around it can be fully controlled. An autonomous mobile robot, however, has to perceive its environment in real-time in order to be able to react to unforeseen circumstances. Even in controlled environments, such as factories or warehouses, the robots have to perceive dangerous situations in order to be safe to work with.

A minimal level of perception can be found in primitive Automated Guided Vehicles (AGVs), that are following predefined paths in controlled environments to transport payloads with minimal human oversight. Typical AGVs detect artificial markers in the environment, in order to determine their own position and to follow their preplanned path. Collision avoidance is then performed on a hardware level by laser scanner safety zones and pressure sensitive bumpers. This low level of on-board computation made these vehicles also one of the early forms of commercially successful robotic systems.

The computational effort massively increases, when robots are employed in dynamic environments, such as hospitals and public buildings. More sophisticated perceptive capabilities are then needed to prevent the robots from harming humans, themselves and their environment. Furthermore, a robot has to possess a higher level of autonomy, so that it can make its own decisions. This means that in addition to a large amount of perceptive capabilities, a high level model of a robot's behaviors is needed to control it in such environments.

The development of perception capabilities and high level control systems is strongly task dependent. Basic software modules, such as self-localization and mapping of the environment, can be easily reused for many different types of robots. More specialized functionalities, however, often have to be newly implemented for each specific problem. Individual modules need a way to interact with each other in order to perform the required tasks together. More specifically, the information generated by perceptive modules has to be accessible to the high level control systems, which in turn have to be able to influence and control every other part of the system.

As robots become more prevalent, the responsibility to program these task specific parts may fall on non-experts. This prompts the need for new tools that simplify the implementation of complex robotic systems with manageable cost and effort. Visual programming techniques have proven promising in other engineering fields, allowing users to quickly combine existing functionality to achieve new tasks. A visual representation of complex systems also allows developers to understand an unknown system more quickly, which is especially helpful in the search for errors and defects.

1.2 Outline and Contributions

The focus of this dissertation is on modeling various aspects of robotic systems using graphical representations we call *Activity Flow Graphs* (AFGs), where both perception and high level modeling are represented in a coherent graphical framework. The framework is designed with the goal to present any robotic system to users and developers in a visual form, in order to speed up the development process, to facilitate the use of existing functionality and to encourage cooperation by increased modularity. The visual presentation through a graphical user interface allows experts and novices to prototype new algorithms more quickly, using a large collection of implemented modules.

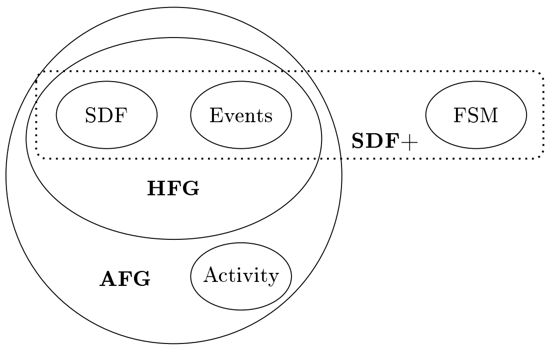


Figure 1.1: Hierarchy of the presented models.

Section 1.3 contains an overview of the different experimental platforms used within this dissertation. On these different robotic systems we consider two main applications of flow graphs: Perception and high level robot control. In the main part of this thesis, we will incrementally develop the AFG model and show illustrative examples. The incremental nature of the presented models is demonstrated in Figure 1.1, which we will use throughout this thesis to highlight the focus of the individual chapters. First, we recapitulate common approaches to model robotic applications in Chapter 2, where we summarize *Synchronous Data Flow* (SDF) and *Finite-State Machines* (FSMs).

In Chapter 3 we present a fundamental data flow model based on SDF, *Events* and FSMs, which is based on the article published in

- Buck, S., Hanten, R., Pech, C. R., and Zell, A. (2016b). Synchronous dataflow and visual programming for prototyping robotic algorithms. In *Intelligent Autonomous Systems (IAS), The 14th International Conference on*, pages 911–923, Shanghai, CN.

In the following this model is called SDF+ to emphasize the three distinct parts, as visualized in the hierarchy shown in Figure 1.1.

We describe a supplementary example application in Chapter 4 in the form of a fetch-and-delivery robotic system developed for the SICK robot day 2014. This chapter is based on the publications in

- Buck, S., Hanten, R., Huskić, G., Rauscher, G., Kloss, A., Leininger, J., Ruff, E., Widmaier, F., and Zell, A. (2015). Conclusions from an object-delivery robotic competition: Sick robot day 2014. In *Advanced Robotics (ICAR), The 17th International Conference on*, pages 137–143, Istanbul, TR,

- Huskić, G., Buck, S., and Zell, A. (2016). A simple and efficient path following algorithm for wheeled mobile robots. In *Intelligent Autonomous Systems (IAS), The 14th International Conference on*, Shanghai, CN,

Subsequently, we formulate a combined flow graph model in Chapter 5 that unifies SDF and event-based messaging into a single graphical model which we call *Hybrid Flow Graphs* (HFG).

Chapter 6 then presents the application of HFGs to the freely navigating transportation robot PATSY, where we implemented different perception algorithms, which where published in

- Buck, S., Hanten, R., Bohlmann, K., and Zell, A. (2016a). Generic 3d obstacle detection for agvs using time-of-flight cameras. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 4119 – 4124, Daejeon, Korea,

- Buck, S., Hanten, R., Bohlmann, K., and Zell, A. (2017). Multi-sensor payload detection and acquisition for truck-trailer agvs. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, Singapore,

- Hanten, R., Buck, S., Otte, S., and Zell, A. (2016). Vector-amcl: Vector based adaptive monte carlo localization for indoor maps. In *Intelligent Autonomous Systems (IAS), The 14th International Conference on*, Shanghai, CN,

or have been submitted for publication in

- Hanten, R., Kuhlmann, P., Buck, S., Otte, S., and Zell, A. (2018). Robust real-time 3d person detection for indoor and outdoor applications. (submitted).

In Chapter 7 we extend the HFG model into *Activity Flow Graphs* by introducing the concept of *Activity*, which allows the implementation of both perception and high level control algorithms in a common framework. AFGs are then applied to several example scenarios in Chapter 8, where we demonstrate how to model high level robot control and perception in a common framework. These chapters are additionally based on

- Buck, S. and Zell, A. (2018). CS::APEX: A framework for algorithm prototyping and experimentation with robotic systems. *Journal of Intelligent & Robotic Systems*,
- Huskić, G., Buck, S., and Zell, A. (2017b). Path following control of skid-steered wheeled mobile robots at higher speeds on different terrain types. In *IEEE International Conference on Robotics and Automation (ICRA)*, Singapore,
- Huskić, G., Buck, S., Ibargüen González, L. A., and Zell, A. (2017a). Outdoor person following at higher speeds using a skid-steered mobile robot. In *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*, Vancouver, Canada.

The graph-based nature of the AFG model and its constituent parts allows for a simple and intuitive graphical user interface for generation, modification and inspection. This also motivates the merging of SDF, Events and FSMs into a single framework, which can be adapted into a coherent user interface with which every aspect of the robotic system can be designed, implemented and tested.

In Chapter 9 we present our open-source implementation of a visual programming approach to the AFG model. The framework is called the *Algorithm Prototyper and Experimenter for Cognitive Systems* (CS::APEX) and is designed to allow rapid prototyping of robotic algorithms. This chapter is also based on the publications in

- Buck, S., Hanten, R., Pech, C. R., and Zell, A. (2016b). Synchronous dataflow and visual programming for prototyping robotic algorithms. In *Intelligent Autonomous Systems (IAS), The 14th International Conference on*, pages 911–923, Shanghai, CN,
- Buck, S. and Zell, A. (2018). CS::APEX: A framework for algorithm prototyping and experimentation with robotic systems. *Journal of Intelligent & Robotic Systems*.

Due to the hierarchical nature of the presented models, CS::APEX can also be used to implement SDF+ or HFG models. An additional, external implementation of finite-state machines then has to be used for high level state modeling. This has been the foundation for the experiments that have been performed in this thesis: Every graph that is shown in the following chapters has been prototyped, optimized and evaluated using the graphical user interface.

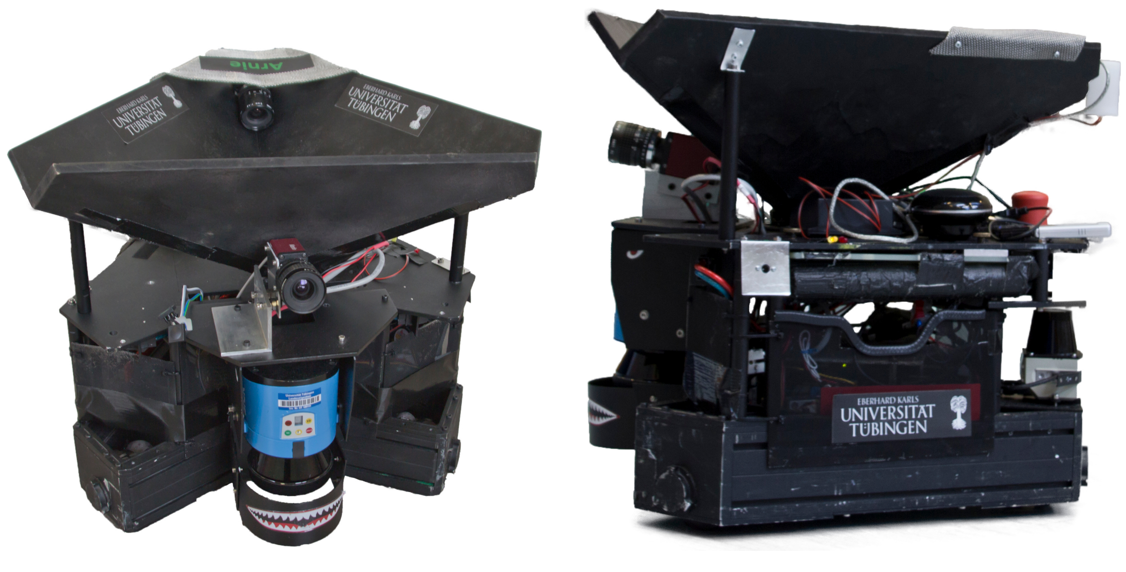
Chapter 10 concludes this thesis by summarizing and discussing the results.

1.3 Experimental Platforms and Projects

Several robotic platforms are used to experimentally demonstrate the generic applicability of our proposed methods. The platforms provide different types of kinematics and varying levels of processing power. Every robot is used in the context of a separate project, which all require different perception solutions and high level control policies. The different platforms also roughly correspond to the three hierarchical models in this thesis: The SDF+ model was mainly developed with our omnidirectional Robocup robots. The HFG model is the result of lessons learned during the development of SDF+ and was extensively used to implement many components in the PATSY project. Finally, the AFG model was developed to implement a fully autonomous Summit XL robot.

1.3.1 Omnidirectional Robocup Robots

Figure 1.2 displays one of the two identical robotic systems we have constructed for the participation in a robotics competition called SICK Robot Day 2014. The robots are based on hardware that was used at RoboCup a few years back and is described in more detail by Kanjanawanishkul and Zell (2009). These robots have an omnidirectional base with three omnidirectional swedish wheels. The original use case demanded relatively large robots to be able to interact with soccer balls.



(a) Frontal view showing the LMS 100 laser scanner and two cameras.

(b) Side view.

Figure 1.2: View of one of our two robots with a hopper on top. Visible sensors: Allied Vision Marlin camera (center), PointGrey Firefly camera (top, inside the hopper), SICK LMS100 LIDAR (bottom).

For this competition we lowered the center of gravity as much as possible to allow faster movement speeds without increased risk of toppling over in case of emergency braking. We installed an Intel Core2Duo P8700 dual-core processor with 2.53GHz clock speed and 2GB of RAM just above the motors in an open space previously unused. The platform was equipped with a basket in the shape of a pyramid with a triangular basis standing on its tip, which we used to gather wooden cubes. Inside this hopper we mounted a PointGrey Firefly camera, which was used for detecting the cubes and reading their bar codes.

Furthermore, we added the required green signaling lamp to indicate to human operators that the robot intends to receive or deliver a cube. We employed two laser scanners for obstacle avoidance and localization, a front-facing SICK LMS100 and a rear-facing SICK TiM551, which together allowed for 360° vision with only little blind spots to the sides of the robots. Additionally, we used a front facing Allied Vision Marlin camera for sign and target detection. Front and rear are just defined for discussion, since the platforms are omnidirectional and therefore do not have a constrained movement direction. We implemented multiple perception pipelines using the SDF+ model presented in Chapter 3 and modeled the high level control using a finite state machine.

1.3.2 PATSY Robot

The algorithms and frameworks presented in this work were largely developed in the context of the BMBF funded *PATSY* project, which is an abbreviation for *Person recognizing Autonomous Transport SYstem*. The project was realized in cooperation with the E&K Automation company in Reutlingen, Germany and had the goal to create an AGV that is able to navigate in dynamic environments. Specifically, the operation scenario was described as a robust transportation system that can be deployed in public areas of hospitals to transport heavy payload containers.

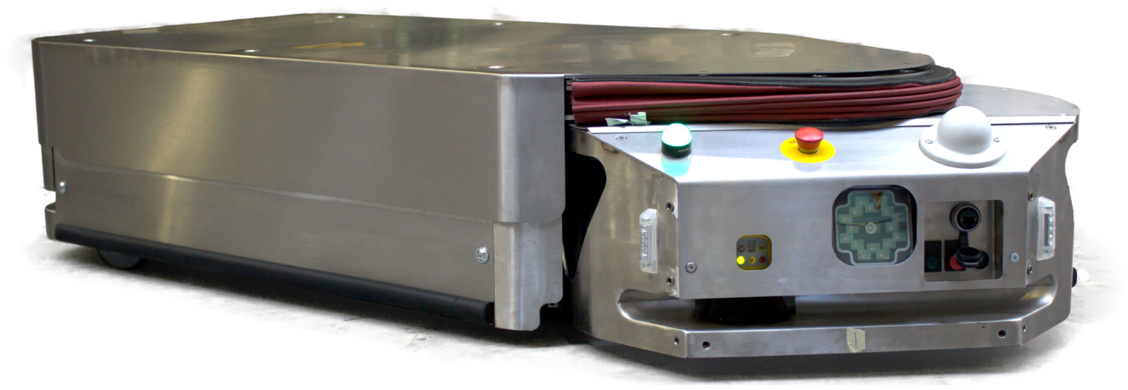


Figure 1.3: The truck-trailer AGV used in the PATSY project.

Figure 1.3 shows the AGV, which is used to transport containers by driving under them and lifting them up. The robot is equipped with a powerful Intel Core i7-3517UE CPU processor and 8 GB of RAM, which allows the concurrent application of many different perception and planning modules. A relatively large and complex system had to be developed on the software side, with different subsystems, including self-localization in dynamic environments, the detection of people and dangerous obstacles, the accurate detection and localization of large transport containers and robust navigation among dynamic objects. Many of these components where modeled and implemented using the data flow based formulations presented in Chapters 5 and 7.

The AGV has a very small ground clearance, which means that obstacles range from a few centimeters to several meters in size. The AGV is equipped with two Sick S300 safety laser range finders, which can only detect obstacles in a plane roughly 8 cm above the floor. Since the AGV is meant to operate in populated areas, objects below the laser scanner plane and overhanging obstacles have to be recognized reliably as well. Furthermore, people are hard to detect using 2D laser scanners alone, especially at larger distances.

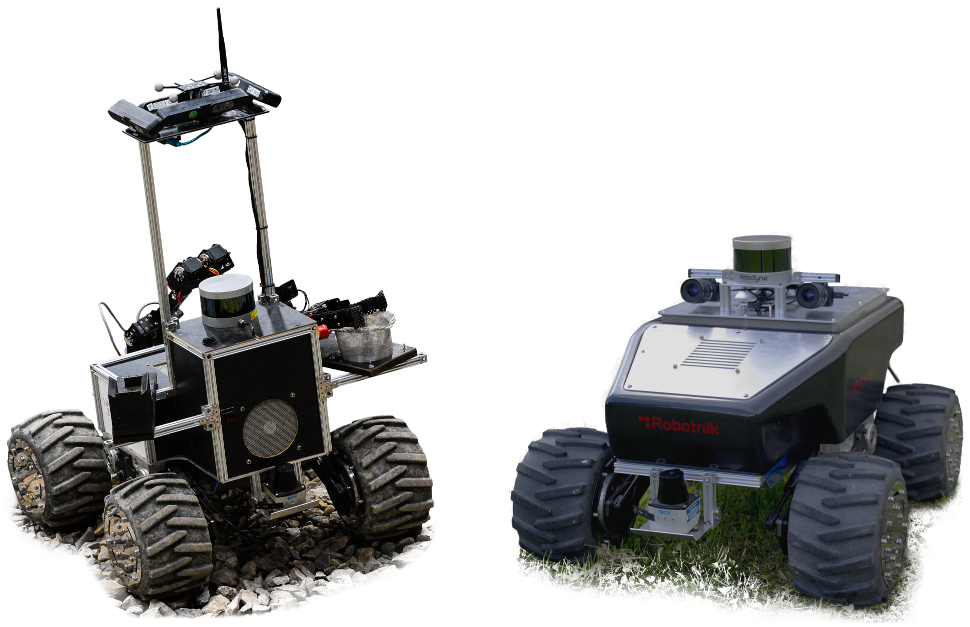
That is why three dimensional perception was the major motivation of the PATSY project. To measure 3D information, the PATSY prototype was equipped with a Fotonic E70-P *time-of-flight* (TOF) camera, after an extensive evaluation of different types of sensors by Rauscher *et al.* (2014). TOF sensors can be compared with standard cameras, yet instead of detecting color information in each pixel, these cameras directly measure the distance to the nearest object and the intensity of the reflected light in each pixel. Due to the necessarily low height of the AGV, the sensor has to operate close to the ground, which causes different types of data artifacts, which we will discuss in Chapter 6.

1.3.3 Robotnik Summit XL Robots

Many additional experiments were performed using the all-terrain robot Robotnik Summit XL, shown in Figure 1.4. Perception pipelines developed in the PATSY project, such as person detection and 3D obstacle detection, were adapted and used on the Summit XL. Whereas the PATSY robot is exclusively used indoors and requires a flat floor to operate on, the Summit XL is mainly used outdoors in rough terrain. Therefore the requirements for the applicable sensors are completely different.

The basic setup of the Summit XL consists of an Intel Core i7-4790S processor with 4 cores and 16 GB of RAM, which is more powerful than the processor of the PATSY robot. The robot is equipped with a Velodyne VLP-16 laser scanner, which allows 360° perception and mapping. A Razor 9DOF Inertial Measurement Unit, together with the on-board odometry, is used to estimate the robot's pose.

The first scenario we explored was a simulated space mission called *SpaceBot Camp 2015*, for which we used four ASUS Xtion Pro Live RGB-D cameras to achieve autonomous, high resolution mapping of a simulated planetary surface. We modeled most perception capabilities using the *Hybrid Flow Graph* model presented in Chapter 5. These include the detection of predefined objects using RGB and RGB-D cameras and the detection of obstacles in the environment for safe autonomous navigation. A separate



(a) The SpaceBot Camp configuration used four RGB-D cameras and a Velodyne VLP-16.

(b) The outdoor configuration uses a stereo camera and the VLP-16.

Figure 1.4: The all terrain robot Summit-XL by Robotnik.

finite-state machine was implemented to solve high-level planning tasks and mission control.

Some of the high level control in the SpaceBot Camp design led to the development of *Activity Flow Graphs* presented in Chapter 7. We then developed a service robot application, in which the robot was following a person with up to $2.5\,\mathrm{m\,s^{-1}}$. In this case we equipped the robot with a stereo camera to replace the RGB-D sensors, which do not work outdoors due to direct sunlight. Using the AFG model, we were able to implement the complete system, including navigational tasks, in a single framework. No additional state machine for high level control was needed.

Chapter 2

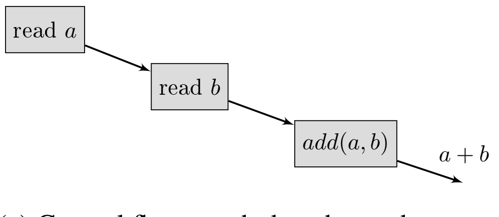
Graph-based Computation Models In Robotics

There are two ways to model complex computations in a graphical way: *data flow* and *control flow*. Both models are dual and complement each other. In data flow, the primary concern is to model the way data packages are moving through various computational steps in an algorithm. A data flow graph does not specify any ordering of different computations, rather the availability of data packets determines, which computation is performed. Control flow, on the other hand, is used to explicitly model an ordered execution, in which control is passing between components of a flow graph.

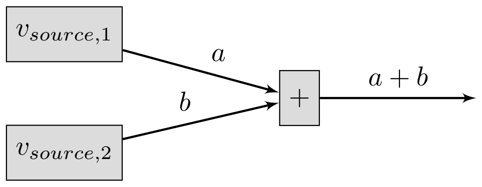
A typical computer program can be directly modeled as a control flow graph. Each instruction to the CPU can be thought of as a node in the graph. After an instruction is executed, the program counter transfers the control to the following instruction. Jump-based structures, such as *if* statements and *loops*, can be interpreted as selection nodes, where the control can flow to different child nodes, depending on currently held data.

Many sequential algorithms can also easily be expressed as a data flow graph. Figure 2.1 depicts a simple example in which the addition of two values is modeled. First it is shown as a control flow graph in Figure 2.1a and then as a data flow graph in Figure 2.1b. Instead of focusing on a sequence of instructions, the data flow model defines each instruction as an actor that consumes data and produces other data as a consequence. Functions written in any programming language can be thought of as such an actor, consuming all its parameters and producing corresponding return values. In this way, data flow models are a more functional approach, where the order of computations is determined by a scheduling algorithm, instead of the explicit ordering of instructions in control flow.

We commonly understand a system as a robot if it performs perception tasks and if it



(a) Control flow graph that shows three consecutive operations to perform addition.



(b) Data flow graph that shows two data sources and a node that performs addition.

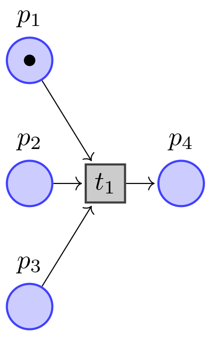
Figure 2.1: Example flow graphs to model a simple sum of two numbers.

can influence its surroundings in some form. Similarly, one of the earliest definitions of robotics goes back to Brady (1985), who wrote that "Robotics is the intelligent connection of perception to action." In this chapter we review common ways by which the two graphical models are applied in the field of robotics. In particular, we focus on data flow applied to *perception* problems and control flow in the form of *action* in *high level robot control*.

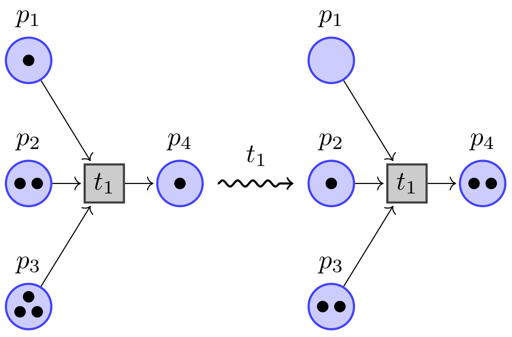
2.1 Petri Nets

Petri nets form a graph-based modeling language which is commonly used to describe distributed systems. A Petri net $N = (P, T, E)$ is a bipartite graph with vertices $P \cup T$ and edges E. The vertices are also called *places* (P) and *transitions* (T), and are two disjoint sets. Every edge $e \in E$ either connects a place to a transition, or a transition to a place.

Places are visualized by circles and represent passive structures, which can store an arbitrary amount of *tokens*, as shown in Figure 2.2a. On the other hand, transitions are active components used to represent actions or events that occur in a system. A transition, visualized as a labeled square, is *enabled* when all incoming arcs are connected to a place with at least one token. Once a transition is enabled, it can *occur* at any time, also in parallel to other transitions. We say that a transition is *fired* when it occurs, to emphasize their active role. When a transition occurs, a token from each of the incoming places is subtracted and a new token is added to the outgoing places, as visualized in Figure 2.2b.



(a) The place p_1 has a token. Transition t_1 is not enabled, because p_2 and p_3 have no tokens.



(b) Transition t_1 is enabled. After t_1 is fired, p_4 has one more token, the other places lose a token. t_1 is no longer enabled.

Figure 2.2: The transition t is *enabled*, when all incoming arcs are connected to a place with at least one token.

Petri nets are intuitive and expressive, however they are a low-level abstraction and can become very complex for larger systems. We will use Petri nets in the following chapters to model the individual components of more abstract data flow models.

2.2 Flow-based Programming for Perception

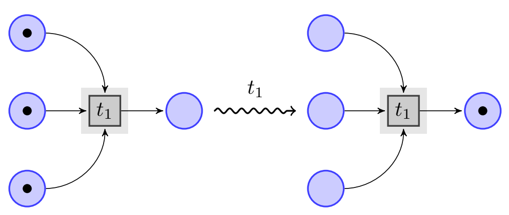
A complete robotic system has to implement different subsystems with varying requirements: A low-level controller has to be executed at frequencies over 1 kHz, whereas high-level decision making only needs to be performed very infrequently. One important subsystem between the two extremes is perception, which is preferably running at frequencies equal to the data acquisition rate, e.g. $30-60$ Hz for color cameras.

Perception is the task of interpreting and understanding the data collected by a robot's sensor. In most of such tasks, there is a clear flow of data: The robot's sensors are data sources that generate continuous streams of data. These streams are then processed, filtered and transformed by different processes, which can extract more abstract information useful at a higher level. Finally, the extracted information flow ends at sinks, which can be low-level hardware drivers that perform actions in the real world.

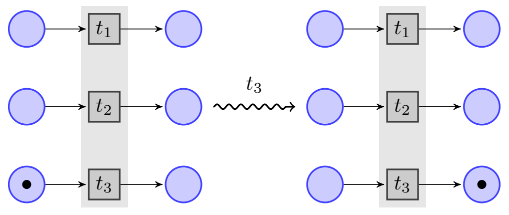
Due to the naturally emerging data flow, it is common to model perception using data flow graphs, which is also called *Dataflow Programming* (DP) or *Flow-based Programming* (FBP). These data flow graphs are also often called pipelines, because data packets are sent along a chain of processes analogous to a liquid in a pipeline. There are many different ways to implement a data flow on a robot, both within a single pipeline, and across different abstraction layers.

These approaches are equally expressive, however they focus on different aspects. One aspect is the nature of the streams or buffers between the processing nodes. Some aspects assume infinite-sized connection buffers, whereas others use bounded buffers. Another characteristic is the evaluation model: *Pull* models are a demand-driven way to control the data flow, where a node produces data only once a child node requests it. In contrast, *push* models are data-driven where nodes produce data whenever they can be executed.

Finally, a data flow model can be either *synchronous* or *asynchronous*. This distinction is also called *any*- vs. *all-activation*: In a synchronous system, a node can process incoming data only once *all* of its inputs have received messages, whereas nodes in an asynchronous system are activated once *at least one* of the inputs has a message. This way, synchronous systems can be modeled using a Petri net with one transition, where all inputs are connected places that have to hold tokens (cf. Figure 2.3a) and an asynchronous system has different transitions for each input (cf. Figure 2.3b).



(a) A Petri net model of a node exhibiting *synchronous* data flow.



(b) A Petri net model of a node exhibiting *asynchronous* data flow.

Figure 2.3: A transition in a Petri net can only occur, when all places have tokens.

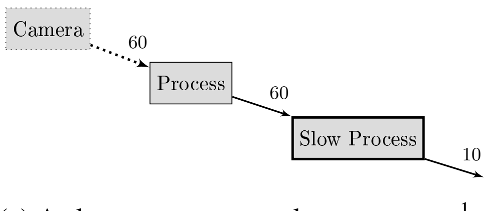
2.2.1 Publish/Subscribe

A common way to realize push-based data flow graphs is the topic-based publish-subscribe paradigm. Probably the most well-known framework using this paradigm is ROS (Quigley *et al.*, 2009). The system consists of processing nodes and individual *topics*, which are named data streams that transport data packets of a specific type. Each process declares a public interface that states the topic the node consumes data from, as well as the topics the node sends messages to.

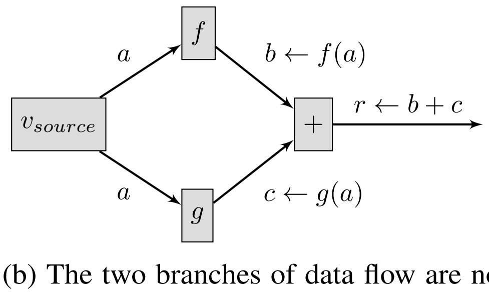
This form of message-passing has a variety of advantages: Two communicating nodes do not know of each other's existence, they are only coupled to the topic with which they are communicating. In addition, different nodes can be executed on different systems, in which case the common topics are used to transport data between machines. Lastly, the different nodes can implement subsystems at different levels of abstraction and run at different frequencies. An odometry node, for example, can produce data at a rate of 500 Hz and publish it on a topic `odom`. A higher level node, with a notably lower execution rate, can subscribe to the topic and process a batch of received messages in a single iteration.

One important downside of this approach is that the individual processing nodes do not receive feedback from their successors. A data source may run at a high frequency, yet if all consuming nodes are running at significantly lower rates, many of the produced data packets have to be dropped. Streams are generally running indefinitely, which means that processing power can be wasted, since not all information can be propagated completely. Figure 2.4a shows such an example, where a slow process cannot consume all data produced by its parent node, but rather only one sixth of incoming packets.

Furthermore, each topic is subscribed to individually, which normally means that a callback function is registered, which is called once data is available. Many processes subscribe to multiple topics, however, and managing these subscriptions has to be done for each node in the graph. For example in Figure 2.4b, the addition node is subscribed to two streams, *b* and *c*, which in turn are produced by transforming a source stream *a* with two functions *f* and *g*. The implementation of the node has to explicitly manage the



(a) A slow process can only process ca. $\frac{1}{6}$ of the incoming packets.



(b) The two branches of data flow are not guaranteed to be synchronized.

Figure 2.4: Problematic situations for data flow models.

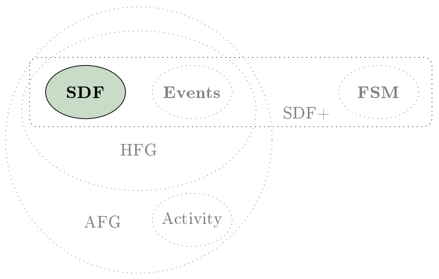
synchronization between the two topics. This process is completely asynchronous and there is no way to guarantee, that all data of the two streams is processed. Additionally, the result r can only be specified as $r = f(a_u) + g(a_v)$, where u and v are not guaranteed to be identical, which means that in general $r \neq f(a_t) + g(a_t)$ for each data packet a_t.

2.2.2 Request/Response

A request-response style approach implements pull-based data flow, similar to the publish-subscribe model. The inverted evaluation model lends itself more to a control flow approach, where consuming nodes request new data from their parent nodes once they are executed. This can be implemented using remote procedure calls. This approach is therefore rarely used in a pure data flow framework and is rather an additional tool. ROS is using both a publish-subscribe model for data flow and the request-response model for control flow.

2.2.3 Synchronous Data Flow

All flow-based programming methods can be described and designed using a data flow graph. However, the publish-subscribe and the request-response approaches only implicitly model a graph structure, whereas other methods explicitly formulate a data flow graph. One of the best developed, explicit data flow models is *Synchronous Data Flow* (SDF), which was first described by Lee *et al.* (1987). SDF is synchronous, which means that a processing node can only be executed, once all its inputs have received a message. This synchronizes parallel nodes, since the pipeline is blocked, until all nodes are finished processing.



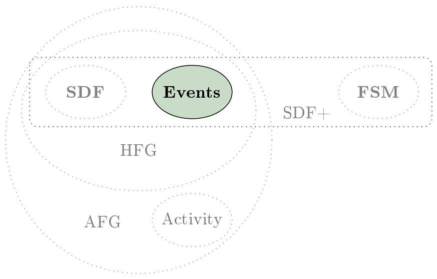
Additionally, SDF is also commonly constrained to be *homogeneous*, which means that a node always consumes and produces exactly one message on each input and output in each execution. This constraint makes every node in the graph predictable, so that a static scheduling scheme can be calculated. Non-homogeneous nodes have to be handled with care, so that no locking can occur.

A well-known example of a successful implementation of SDF is the LUSTRE language (Halbwachs *et al.*, 1991). LUSTRE is an early SDF-based programming language that has been designed to implement reactive systems. Its formalism, which is similar to temporal logic, and its synchronous nature make it well suited for time-critical programming. Furthermore, the formal approach makes it possible to implement real systems and reason about their properties within a common framework.

2.2.4 Reactive Programming

While flow-based programming is focused on the processes in a flow graph, reactive programming is more focused on the data streams themselves. Whereas we usually assume a regular flow of data in flow-based programming, reactive programming is focused on the propagation of *change*.

The change of every aspect of a system is available as an event. An event is an object in the system that can be subscribed to by registering callback functions that are executed once the event occurs. This is why reactive programming can also be called *event-based* data flow.



More specifically, reactive programming is a form of asynchronous data flow. Events are the sources of the data flow and processes are used to transform, combine or induce new data streams. The data flow has to be handled asynchronously, because it can be irregular and unpredictable. In addition, the data flow is not homogeneous, since processing nodes can modify incoming streams or even create completely new ones.

2.3 High Level Robot Control Flow

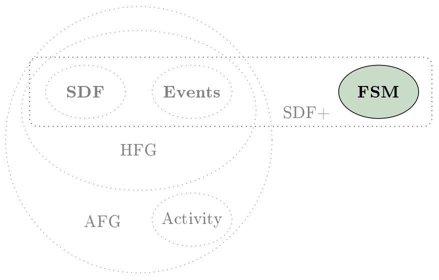
In general, data flow is driven by data alone and does not explicitly represent different *states*. This is an important advantage, because there cannot be any significant side effects in a stateless program. Additionally, without an externally visible state, there are no dependencies between different nodes in a data flow graph, which means that different nodes can be processed at the same time without race-conditions.

To solve complex problems we often need to implement a high-level control law that determines sequences of tasks the robot has to solve. This requires both a systematic way to represent the current state of the system, as well as transition rules that can change to different other states based on the robot's actions. Therefore, pure data flow models do not suffice to model large and complex robotic systems.

For simple applications, the high-level state can be represented implicitly across different processes. For more sophisticated control policies, this can lead to unmanageable complexity. Furthermore, this distributed representation impedes inspection and debugging. There are two common ways to implement complex, high-level policies: (Finite-)state-machine-based and behavior-based systems. State machines explicitly model different possible states and transition between them, where exactly one state is active at a time. Behavior-based systems, on the other hand, avoid explicit state formulation and most often implement reactive action.

2.3.1 Finite-State Machines

A finite-state machine (FSM) is commonly used to model the control flow of large systems graphically. FSMs are not Turing-complete, since the number of possible states is finite. They are, however, well suited for specifying robotic mission policies. The *Unified Modeling Language* (UML) provides different types of diagrams that are very frequently used to model various algorithms and systems. The UML defines different types of diagrams to represent control flow: Sequence diagrams, collaboration diagrams, state charts diagrams and activity diagrams. Of those, state charts and activity diagrams are especially useful for high level control flow applications.



UML state charts are a graphical way to represent finite state machines, as is demonstrated in Figure 2.5 for a fetch-delivery robot, which is described in more detail in Chapter 4. Whereas state charts only represent the state of the system, activity diagrams explicitly model the control flow. Both diagrams are therefore useful for the visualization of complex robotic mission policies. There exist multiple approaches to visually model state machines using state charts and then compile them into a FSM that can be used on a robot. This is done in many real-world applications, such as RAFCON by Brunner *et al.* (2016).

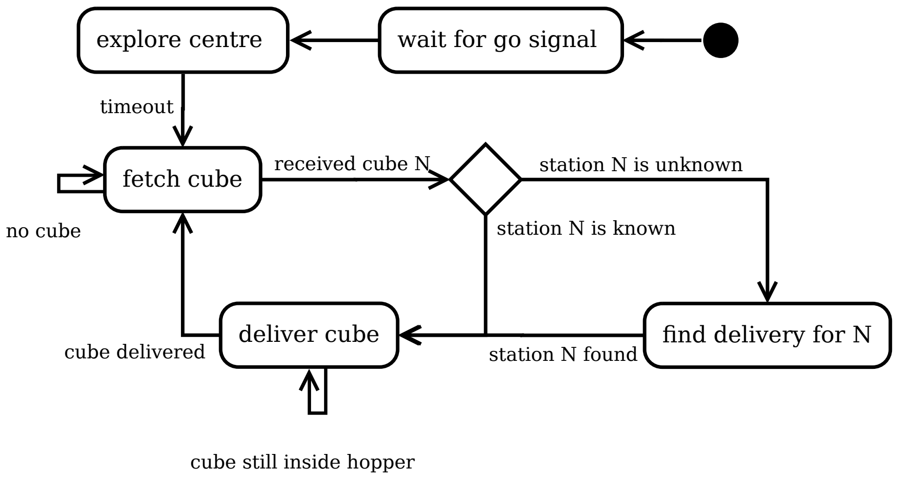


Figure 2.5: An example UML state machine chart of a fetch-delivery robot.

We demonstrate in Chapter 3 how FSMs can complement data flow to model a complete robotic system. In Chapter 7 we then apply the activity diagrams idea directly to data flow graphs. We show that modeling the flow of activity explicitly is equally useful for high level control modeling. Additionally, using an activity-based formulation, we do not have to model the state transitions and data flow graphs in separate frameworks but in a single, coherent graphical model.

2.3.2 Behavior-Based Models

Another common way to model high level robot control policies are so called behavior-based models. Here, the idea is to define different reactive behaviors, instead of modeling the different states the robot can be in. In a behavior-based system, all components are active at the same time, compared to FSMs, where exactly one state is active. To influence the high level control, currently unnecessary behaviors are inhibited, such that only the relevant behaviors influence the robot control. There exist many systems based on behaviors, such as *Integrated Behavior-Based Control* by Proetzsch *et al.* (2007).

2.4 Conclusion

We have seen three graphical components that are commonly used to model parts of robotic systems. Petri nets are useful on a mathematical level to precisely describe complex systems in a rigorous way. On the other hand, they quickly result in verbose models and are therefore rarely used directly for the final implementation of a system. The opposite is true for flow-based programming and high level modeling mechanisms, which are commonly used for abstract modeling and implementation, but are often not easy to reason with.

In Chapter 3 we use the three distinct models together to derive a the SDF+ model, which is capable of modeling a complete robotic system: We define a flow-based model based on SDF and reactive programming using the framework of Petri nets. A finite-state machine complements the data flow model and allows high level state to be represented. We demonstrate the applicability of the SDF+ model in Chapter 4, where we model all aspects of a fetch-and-delivery robot.

Using the Petri net description, the data flow and reactive programming components of the SDF+ model are then generalized in Chapter 5 into the coherent HFG model. Finally, the need for a separate finite-state machine is removed in Chapter 7, where the concept of *activity* is introduced.

Chapter 3

Synchronous Data Flow and Event-based Message Passing

In this chapter we describe a basic framework for modeling a complete robotic system, including perception, high level control and mission planning. The framework consists of graphical models for flow-based programming and a separate finite-state machine (FSM).

This layer of our approach, which is highlighted in Figure 3.1, is implemented using homogeneous synchronous data flow, which was introduced in Section 2.2.3, and event-based message passing, which allows reactive programming (Section 2.2.4). To highlight the fact that this model is an extension of synchronous data flow, we will refer to this model by the abbreviation *SDF+*.

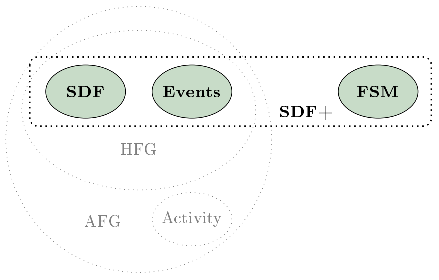


Figure 3.1: Synchronous Data Flow with Events and FSMs (SDF+)

The SDF+ model consists of two separate parts: A data flow graph combining SDF and reactive programming, supplemented by an FSM graph that is used for mission planning. Data flow graphs and FSMs models are represented by different mathematical frameworks and cannot easily be unified. Together they can represent any robotic system, as we demonstrate in Chapter 4.

This chapter is partly based on results that were first published in Buck *et al.* (2016b). The theoretical contributions have been extended using the formalism of Petri nets. Although the Petri net descriptions shown below might at first seem overly complicated, we will see their usefulness in Chapter 5, where we show how to unify the different graph-based models.

The precise nature of Petri nets serves as an exact specification for the implementation of SDF+ graphs. This is an important property of our approach, since the main motivation for developing this framework is real-world applicability. We demonstrate this in Chapter 4, where SDF+ is used both in development and in the final implementation of a fetch-and-delivery robot. The model is designed to natively support a graphical user interface for the construction and analysis of computation graphs, which is presented in Chapter 9.

3.1 Data Flow Graph

At first we will look at the data flow. We define both a formal and a graphical way to represent an SDF+ flow graph. Let $G = (V,E)$ be a directed graph of processing nodes V and edges E. Then G combines both aspects from SDF and event-based programming, where each $v_k \in V$ represents a processing node. All nodes exclusively communicate via message passing, which is realized using the connections E. This way, each node is a functional unit, solely described by its inputs and outputs.

3.1.1 Nodes

We represent each node v_k as a set of *ports* that can receive or send messages. Nodes can exclusively communicate using message passing via these ports, which defines a clear and precise interface between them. A node represents a single process in the computation graph. Each $v_k \in V$ is also assigned a *processing function* f_k that reads messages from incoming connections, then performs arbitrary computations and generates messages on the outgoing edges. In addition to the inputs, f_k can also read user-controlled parameters, which we will introduce below.

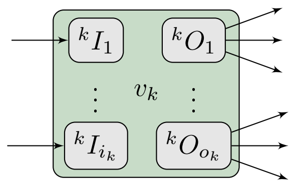
3.1.2 Data Flow Ports

For each node v_k we define a set of *inputs* $\mathcal{I}_k$ and *outputs* $\mathcal{O}_k$

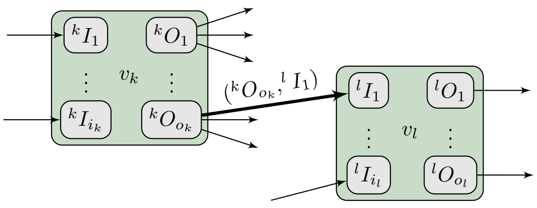
$$\mathcal{I}_k = \left\{ {}^kI_1, \ldots, {}^kI_{i_k} \right\}, \tag{3.1}$$

$$\mathcal{O}_k = \left\{ {}^kO_1, \ldots, {}^kO_{o_k} \right\}, \tag{3.2}$$

as visualized in Figure 3.2a.



(a) A process v_k reads data from inputs ${}^kI$ and writes data to outputs ${}^kO$.



(b) Outputs can send messages to many inputs. Inputs can only be connected to one output.

Figure 3.2: Synchronous data flow is described by input and output ports of process nodes.

We add an edge $({}^kO, {}^lI)$ to E if an output ${}^kO$ of node v_k is sending messages to an input ${}^lI$ of node v_l. Inputs and outputs are typed and can be connected if their types are *compatible*. An output can be connected to arbitrarily many inputs, but inputs can only be connected to one output. We allow for an input to be *optional*, which means that it is ignored, if it is not connected to an output. It is treated as a normal input, otherwise.

We call process nodes without inputs *sources* and nodes without outputs *sinks*. When the processing function f_k is executed, it will read the message from each $I \in \mathcal{I}_k$ and write a message to some of the $O \in \mathcal{O}_k$. After the execution of f_k, the messages for $O \in \mathcal{O}_k$ will be forwarded to all the connected inputs.

3.1.3 Event-based Message Passing

Pure data flow is ideal for processing indefinite streams of information. A robot's perception can be modeled using multiple subsystems that are based on data flow. There are, however, stimuli the system has to respond to, which are more irregular and often not predictable. These can be both triggered by external means or detected within the data stream. We call these stimuli *events* and introduce means to handle them in a coherent framework with the data flow itself.

To realize such asynchronous data flow communication between nodes in G, we define sets $\mathcal{S}_k$ and $\mathcal{E}_k$

$$\mathcal{S}_k = \left\{ {}^kS_1, \ldots, {}^kS_{s_k} \right\}, \tag{3.3}$$

$$\mathcal{E}_k = \left\{ {}^kE_1, \ldots, {}^kE_{e_k} \right\}, \tag{3.4}$$

representing *slots* and *events* of node v_k analogously to $\mathcal{I}_k$ and $\mathcal{O}_k$ (cf. Figure 3.3). Every node $v_k = (\mathcal{O}_k \cup \mathcal{I}_k \cup \mathcal{E}_k \cup \mathcal{S}_k)$ is therefore a composition of inputs, outputs, events and slots. In the graphical notation, we show slots on top of a node and events on the bottom.

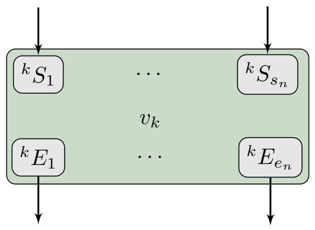


Figure 3.3: Events and Slots on a dual graph structure.

Events can be used to send *signals* to another node by connecting them to slots, contributing an edge $({}^iE, {}^jS)$ to the edges E. Using SDF+, events can only be connected to slots and outputs only to inputs, which gives as a definition for any connection e between two nodes v_i and v_j as

$$e \in \left(\mathcal{O}_i \times \mathcal{I}_j\right) \cup \left(\mathcal{E}_i \times \mathcal{S}_j\right). \tag{3.5}$$

We will lift this constraint when we consider hybrid data flow in Chapter 5.

In contrast to the synchronous data flow, events are more irregular and should be handled asynchronously once they are triggered, so that not all events have to receive a message at the same time. This means that slots can be connected to multiple events and vice versa. By not using the data flow to send events between nodes, we avoid sending special marker messages. Additionally, disjoint data flow sub-graphs can run at different frequencies but can still communicate via events.

3.2 Parameters

For each computational node $v_k \in V$ we define a set of parameters $\mathcal{P}_k$, which are treated both as inputs and as outputs of v_k by adding additional inputs $\mathcal{I}_k^P \subset \mathcal{I}_k$ and outputs $\mathcal{O}_k^P \subset \mathcal{O}_k$ (cf. Figure 3.4). Parameters are declared for each node type and control the internal processing of each instance of that node type. A node v_k can read its parameters at any time and is allowed to change them.

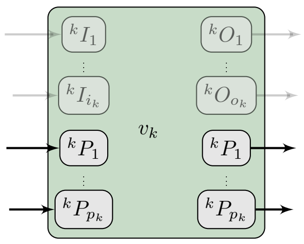


Figure 3.4: Inputs and outputs of node v_k. Parameters ${}^kP_i$ are both inputs and outputs, where messages are automatically forwarded.

A parameter's value can also be changed by other means: An incoming message at a parameter input port causes the value of that parameter to be updated. At every execution of f_k, all the parameters' values are sent as messages on the corresponding output ports. Both mechanisms together allow values of different parameters to be synchronized without their nodes knowing about each other.

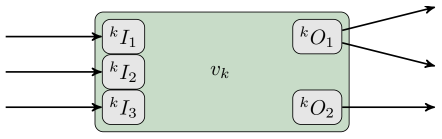
Parameters behave the same as regular input or output ports and can be connected to any other port, making the parameter accessible to the network. Values computed by a node can be manipulated using further processing nodes and then assigned to another node's parameter, for example. At the same time, this explicit modeling enables a user interface to present control panels to adjust the parameter values, giving the user a more direct control over the data flow.

3.3 Petri Net Execution Model

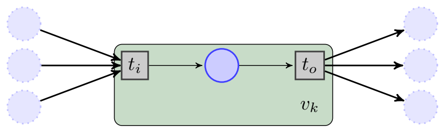
In order to precisely describe the behavior of the SDF+ model, we now develop an execution model based on Petri nets. The idea is that the resulting nets can be directly used to infer the behavior of any SDF+ graph, independently of implementation details. Additionally, we use different types of tokens: A 🖂 token represents a message that is supposed to be handled synchronously, whereas a ↯ token should be handled asynchronously. In other cases we use a generic ● token to signify that the token does not represent a message.

3.3.1 Synchronous Data Flow

We borrow some of the terms from Petri nets to describe the synchronous data flow. Consider the node v_k shown in Figure 3.5a, which has three inputs and two outputs.



(a) A node with three inputs and two outputs. In contrast to outputs, inputs need to have exactly one connection.

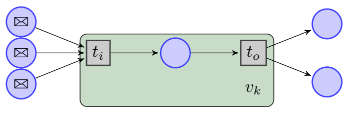


(b) The input transition t_i and the output transition t_o represent all inputs and outputs of v_k. Each connection is modeled as a place.

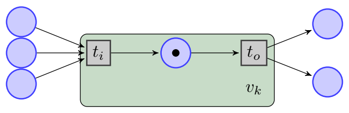
Figure 3.5: An example node and its Petri net depiction.

We say that v_k becomes *enabled* once all of its inputs have received a message and all of its outputs can send a new message. This can be (almost) directly described using an input transition t_i and an output transition t_o in the corresponding Petri net formulation shown in Figure 3.5b. Although a node can have arbitrarily many inputs and outputs, the execution model only needs these two transitions.

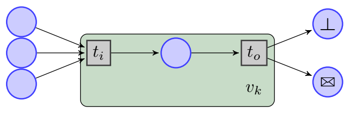
The Petri net model of v_k shows the synchronous data flow graphically: Once all predecessors of t_i hold a token, t_i can occur, which removes those tokens and creates a new one in v_k. This represents the execution of the processing function f_k, which is associated with v_k and is visualized in Figure 3.6a and Figure 3.6b.



(a) v_k is enabled.



(b) f_k is executed.



(c) Messages are sent.

Figure 3.6: A node reads all input tokens, processes them and then produces output tokens. This nicely demonstrates the flow of tokens and the synchronous data processing.

Sources are enabled whenever their outputs can send messages and sinks are enabled when all inputs have received a message. Enabled nodes can be executed whenever processing resources are available. Here we can immediately see the meaning of synchronous data flow: Tokens are synchronized on input transitions and then processed at once. After the execution of f_k, the node (possibly) generates outputs (Figure 3.6c). If there are no messages to be sent, we propagate a special $\perp$ token instead, which represents *no message* and can also be seen in Figure 3.6c. This way, every output generates exactly one message per iteration, which is also called *homogeneous* synchronous data flow.

Edges in the graph represent *connections* between nodes on which messages are sent. Each connection joins exactly one output to one input and can also be represented using two transitions, which are shown in Figure 3.7.

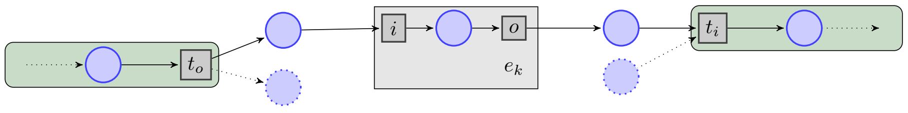
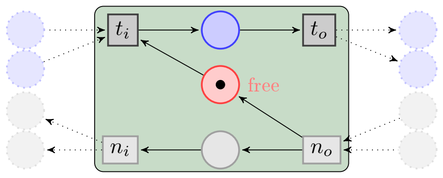


Figure 3.7: Each connection e_k transmits a token from an output t_o to an input t_i.

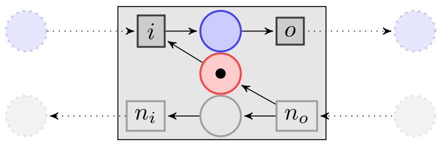
The transition i can occur once the corresponding output transition t_o has produced a token. This can be interpreted as buffering a single token in the connection e_k, which is shown as a gray box. Once e_k holds a token, the transition o can occur, relaying the message to the connected node.

3.3.2 Notifications

The simple Petri net shown in Figure 3.6 only partly represents the desired behavior, because it only ensures that all inputs have a token. The output places can accumulate arbitrarily many tokens, which we do not allow in the SDF+ model. To model the flow completely buffer-less, we introduce *notifications*.



(a) The function f_k can only be executed when the (red) enabling place holds a token.

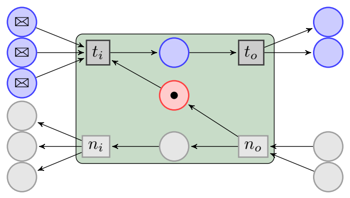


(b) A connection can also only forward a single token until it is notified.

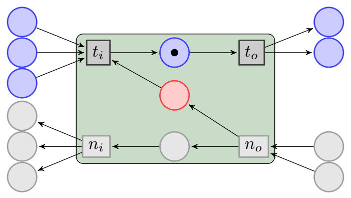
Figure 3.8: After sending tokens, nodes and connections have to be notified.

We extend the net by adding *enabling* places, which need to hold a token so that the processing function f_k can be executed. The place is shown as a red circle in Figure 3.8a. Additional notification transitions n_o and n_i are also added, which complement t_o and t_i. We denote places involved in data flow with blue circles and notification places in gray.

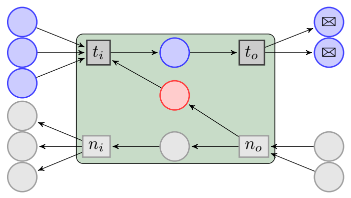
Once a node has processed incoming tokens and produced output tokens, the red place does not hold a token anymore (cf. Figure 3.9a to Figure 3.9c). This means that f_k cannot be executed again until a notification token is received at n_o. Once all places connected to the notification transition n_o have tokens, which is shown in Figure 3.9d, the node v_k can be re-enabled. This process sets a token at the enabling place and propagates the notification up-stream (cf. Figure 3.9e and Figure 3.9f).



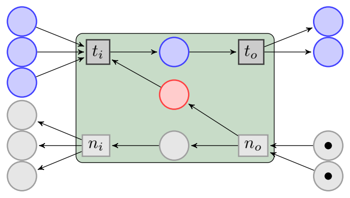
(a) The enabling place has a token, f_k can be called.



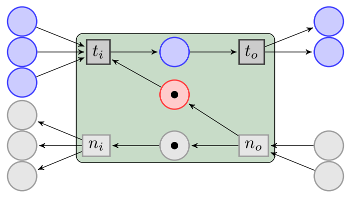
(b) The node is processing.



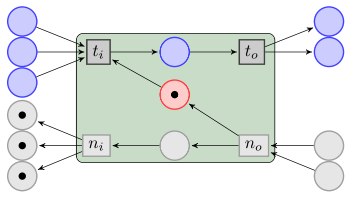
(c) Outputs are generated, v_k is disabled.



(d) The notification transition n_o is enabled.



(e) The node is re-enabled.



(f) The notification is propagated up-stream.

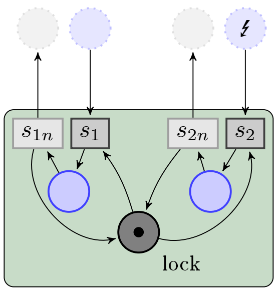
Figure 3.9: Once outputs are sent, a notification is required to allow another iteration.

In a similar fashion, we model the behavior of a connection. Figure 3.8b shows a Petri net, demonstrating the same notification behavior for a connection: The highlighted place at the center of the connection represents, whether the connection can forward a message. Once the input transition i has occurred, the incoming message is buffered inside the connection. The red place no longer has a token, so i occur again. After a connection has sent a token, it has to be re-enabled. This is again done, once it receives a notification token. Until now the main difference between the node model and the connection model is that a connection models a 1:1 relation.

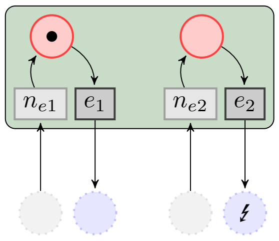
3.3.3 Events and Slots

Building on the Petri net model for synchronous data flow, we now construct Petri nets for slots and events. Slots are designed to process incoming tokens directly, in contrast to the synchronous data flow, where all incoming tokens are processed at once. Figure 3.10a shows a Petri net which implements this behavior. We have a transition s_i for each slot S_i, which means that every token arriving at a slot can directly enable the transition. In addition we have a single enabling place, called a *lock*, for all slots of the same node. This enforces that no two slot callbacks are evaluated at the same time.

In contrast, events are really independent of each other, yet they should not be triggered more than once, before the produced tokens have been processed. Otherwise we would again need to implement an unbounded buffer between events and slots. Figure 3.10b shows the Petri net for a node with two events. They both have their own enabling place, which ensures that each event is only sending new tokens when a notification has been received.



(a) Slots share a common enabling-place and cannot be handled at the same time. Here s_2 is enabled and can be triggered.



(b) Events have individual enabling-places, they can be triggered concurrently. Here event $\mathcal{E}_2$ was triggered and has produced the ↯ token.

Figure 3.10: Petri net models for slots and events.

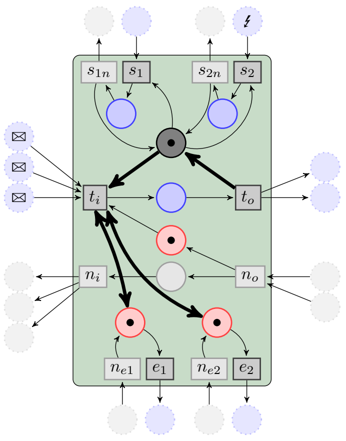
Here the difference between the SDF and events is visually apparent: SDF constructs a pipeline where a token is sent across multiple nodes, followed by a notification token that is sent in the opposite direction. Events, on the other hand, are directly handled by connected slots. The SDF+ model therefore does not require any message buffers, instead it uses the notifications to control the flow of data.

Comparing the Petri nets in Figure 3.9 and Figure 3.10 with the net for a connection in Figure 3.8b, one can see that both inputs and outputs, as well as events and slots can easily be connected using connections. This way, both SDF and event-based message passing are modeled coherently.

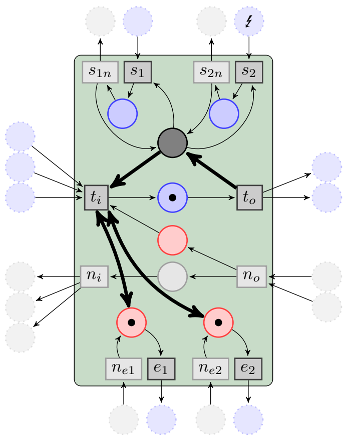
3.3.4 Synchronization

A node v_k encapsulates several possible procedures: One is f_k, i.e. processing incoming messages of the data flow ports. The other possibilities are the slots $\mathcal{S}_k$, each of which has to be processed asynchronously. To achieve correct behavior for stateful nodes, none of these calculations are allowed in parallel. The mutual exclusion scheme deployed in the SDF+ model is shown as a Petri net in Figure 3.11, which is a combination of the individual Petri nets constructed above. Additional arcs, used to connect locks and enabling places, are shown with double lines. Each v_k is assigned a *lock* that synchronizes data flow and event handling, symbolized as an additional place.

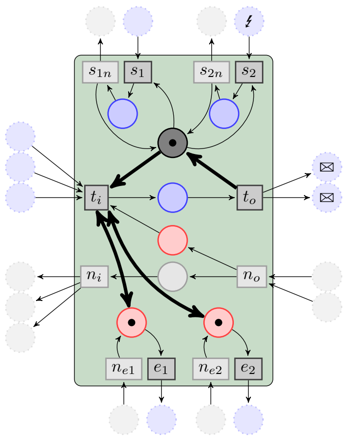
This combined Petri net allows a bi-directional flow of tokens. Places shown in blue carry message tokens: As introduced above, data flow is shown horizontally, whereas



(a) The Node is unlocked and free. It can either process data at t_i, or handle slot s_2.



(b) The node is processing the messages. The lock has been taken, so s_2 cannot occur.



(c) The node is done processing, the lock is returned. Now s_2 can occur.

Figure 3.11: While a process is executed, no slots can be evaluated and vice versa.

events are shown vertically. The net can process incoming data using the input transition t_i, and it can process incoming events using the slot functions s_i. The additional synchronization arcs, shown in bold, ensure, that no locking is necessary: While an event is not enabled, the processing function f_k cannot be executed, ensuring that f_k can trigger every event on each iteration.

Another way to view the two types of data flow is shown in Figure 3.12. Synchronous data flow is propagated to the sink nodes until the notifications are sent back (solid line). Asynchronous data flow immediately returns a notification (dashed lines).

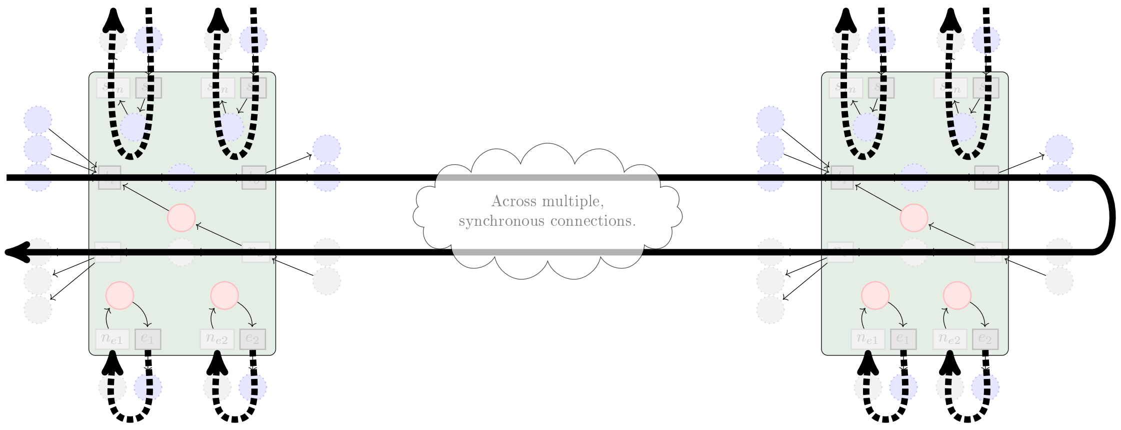
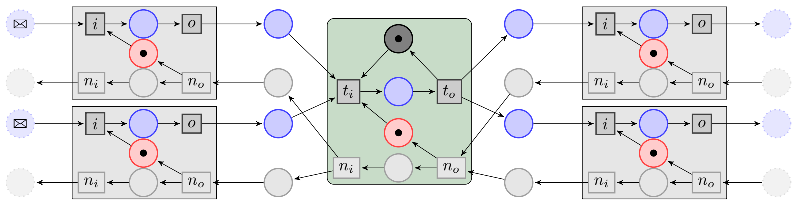


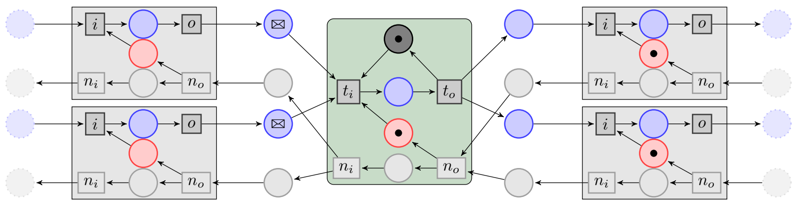
Figure 3.12: Synchronous data flow (solid line) is propagated to sink nodes before any notifications are sent. This propagation can happen across many nodes in the graph. Asynchronous data handling in slots immediately returns a notification (dashed lines.)

3.3.5 Execution Example

Using the Petri nets defined above, we now shortly look at an example that contains both a node and some connections. This demonstrates how a node with multiple inputs can be modeled. Figure 3.13 shows a Petri net for a node with two inputs and two outputs.



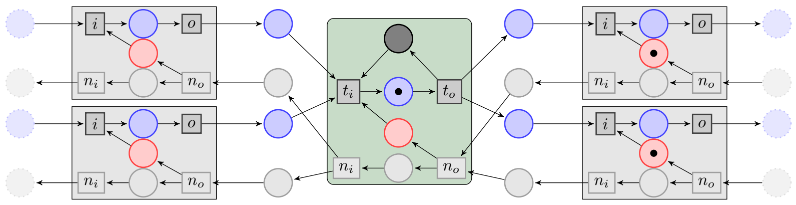
(a) Both incoming connections are enabled.



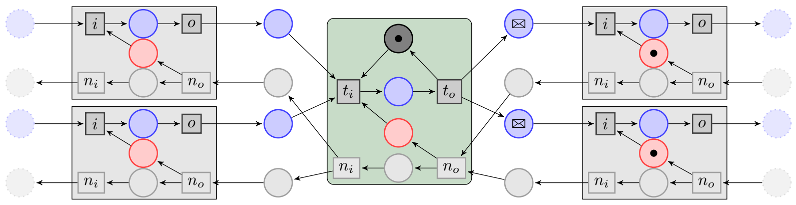
(b) The connections have forwarded the tokens and are now disabled. The node is enabled.

Figure 3.13: Message tokens are arriving at a node and thereby enable it.

After the node is enabled by receiving messages on all inputs, it can call the processing function. As shown in Figure 3.14, the node is then disabled until a notification is received.



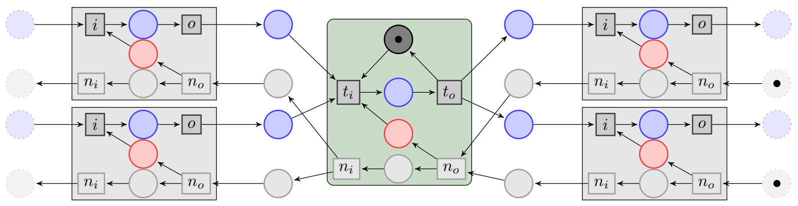
(a) The node starts processing, locking all slots.



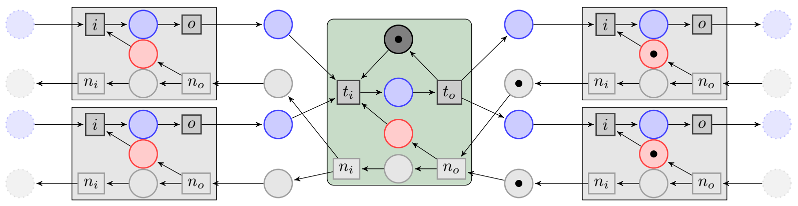
(b) The node has finished processing and has produced outputs.

Figure 3.14: Processing the inputs disables the node until a notification is received.

The graph downstream now processes the generated tokens until there are no more nodes. Sinks send back notifications, which will arrive at outgoing connections of the current node (Figure 3.15a). These then forward the notification to the node (Figure 3.15b).



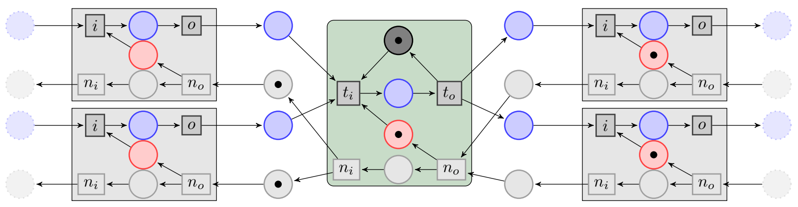
(a) Notification tokens are received at the outgoing connections.



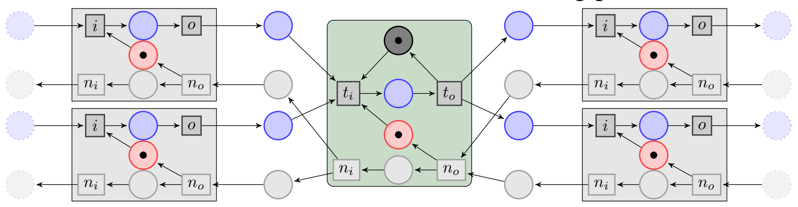
(b) The outgoing notification transition n_o of the node is enabled.

Figure 3.15: When all outgoing connections have been notified, the node itself gets notified, allowing the next iteration of processing.

Once all notification places connected to n_o have received a token, the node is notified. This corresponds to all outgoing connections being processed downstream. At this point in the example, n_o can occur (Figure 3.16a), which places a token at the enabling place of the node. Additionally, the notification is forwarded to all connections upstream, which all get re-enabled before again propagating the notification (Figure 3.16b).



(a) The notification restores a token to the enabling place of the node.



(b) The notification is fully propagated, another iteration can begin.

Figure 3.16: When the node gets notified, it can again be enabled.

This example has demonstrated the synchronous data flow of SDF+. Events are handled directly in a node and therefore do not allow this kind of process chaining. The Petri net model for a connection, however, is also valid for events.

3.3.6 Scheduling

Scheduling the execution of G requires a policy to decide when to execute the different nodes. There can be multiple enabled nodes at any time, which allows concurrent execution. The order in which these nodes are executed does not matter, they can run in a sequence or fully in parallel. To allow dynamic modification of the data flow graph at run-time, a static scheduling scheme, as originally developed for SDF graphs by Lee *et al.* (1987), cannot be employed.

3.3.7 Execution Pruning

Since we propose a push-based model, the active drivers of a flow graph are the source nodes, which are pushing messages through the graph. In addition, the SDF approach is homogeneous, meaning that for each invocation of a node, every output has to produce exactly one token. That is why we publish a $\perp$ token for all outputs that would otherwise produce no token in an iteration.

A homogeneous flow is required, otherwise there could arise a situation as the one depicted in Figure 3.17. Here t_i is still waiting for a token from the upper connection, which will never be received. Using a $\perp$ token solves this problem. This *marker message* has to be forwarded to every node in the connected component down-stream.

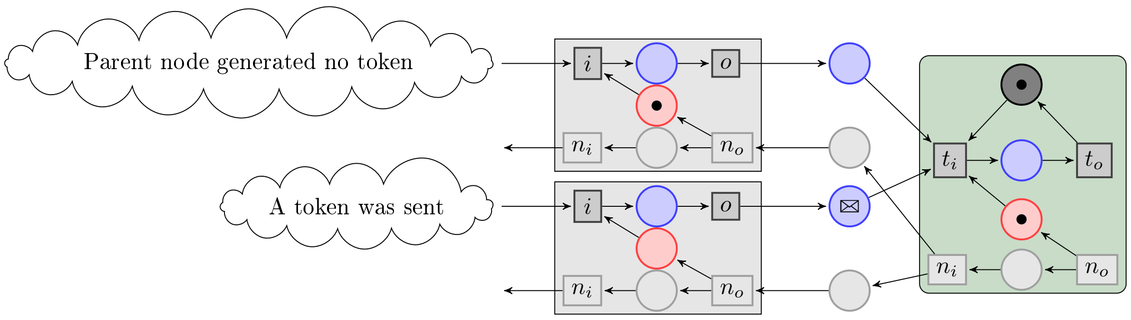


Figure 3.17: Without the use of $\perp$ tokens, nodes can cause a dead-lock at t_i.

When a node v_k receives a $\perp$ token on a non-optional port, the function f_k will not be executed. Instead all incoming tokens are read and $\perp$ tokens are published on all the outputs. The flow is then synchronized, however there are many marker messages transmitted. In larger graphs, this can cause performance problems: Every marker has to be propagated, even though the nodes down-stream cannot be executed.

To avoid this penalty, we make full use of the graph-based nature of SDF+. If a node v_k cannot be executed due to a $\perp$ token, we analyze the connected component of v_k to determine if the $\perp$ token has to be propagated to the children of v_k. This analysis can be precomputed and only has to be updated, once the graph is changed.

There are special cases of nodes that need to receive marker messages, which we call *essential* nodes. The following three characteristics are computed for each node:

$$v_k \text{ is essential} \Leftrightarrow v_k \text{ needs to process marker messages.} \tag{3.6}$$

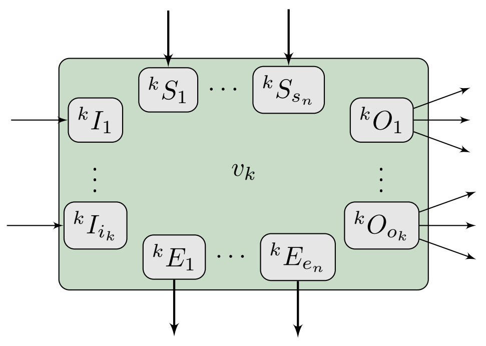
$$v_k \text{ is required} \Leftrightarrow \text{There exists a path from } v_k \text{ to an essential node.} \tag{3.7}$$

$$v_k \text{ has siblings} \Leftrightarrow \text{There exists a node down-stream of } v_k \text{ joining streams.} \tag{3.8}$$

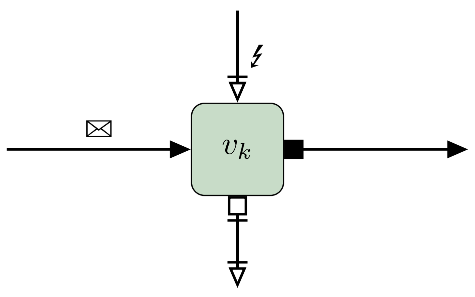
If v_k is required, i.e. it leads to an essential node, we need to forward the $\perp$ token, otherwise the behavior of the system would change. If v_k has siblings, we also need the token, otherwise the scenario shown in Figure 3.17 can occur. In any other case, that is if v_k is not required and if it does not have siblings, we can drop a $\perp$ token completely.

3.4 Simplified Graphical Representation

In most cases, nodes have multiple different message and event ports. This can quickly complicate the graphical notation, especially in large graphs. We therefore simplify the notation and do not show the individual ports where possible, as is demonstrated in Figure 3.18. We omit multiple arcs between nodes, if they are not essential in the current situation. We show synchronous data flow with dark arcs and asynchronous flow with light arcs.



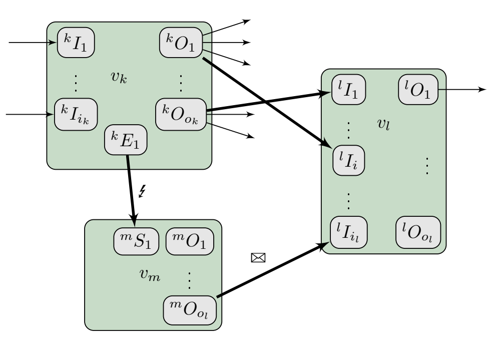
(a) Nodes with many different ports can become unclear when fully visualized.



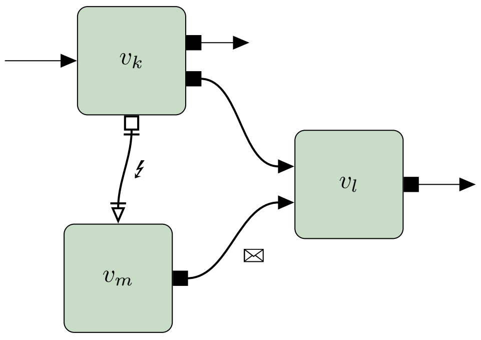
(b) Dark edges carry data flow, light edges carry events. Available messages are marked with ✉ for data flow and ↯ for events.

Figure 3.18: We omit representing ports and multiple arcs in contexts where they are not necessary. The side on which arcs connect to nodes also determines their function.

Using this simplified notation reduces the complexity of larger graphs. The example graph shown in Figure 3.19a can be reduced to the one shown in Figure 3.19.



(a) An example graph in the detailed depiction.



(b) The example from Figure 3.19a in the simplified variant.

Figure 3.19: We omit representing ports and multiple arcs in contexts where they are not necessary. The side on which arcs connect to nodes also determines their function. The connection labeled with ↯ is asynchronous, the one labeled ✉ is synchronous.

3.5 Finite State Machine

Simple mission control could be modeled using data flow, where each data flow node represents a single task and the flow of data determines which tasks are executed. This approach breaks down, when the missions become too complex, since our variant of SDF cannot represent recursive connections, which means that at some point the flow reaches a sink and the program is finished.

Reactive programming using events could more readily be used to model a robot's mission, if the usage of events were to be constrained. However, each event in a graph can be fired at any moment, resulting in concurrent events flowing through the event graph. This makes it difficult to represent a single sequence of actions that is guaranteed to be performed without any other parts of the graph interfering.

That is why both data flow and event-based reactive programming, as described so far, are defined to be basically state-less. It is also the reason for the third component in the SDF+ model: A separate finite-state machine. The FSM is used to represent any form of high level state that is necessary to model mission control for a robot, i.e. performing sequences of tasks at a high level of abstraction. The state machine can generate events that can be reacted to by the flow graph, e.g. by disabling currently unnecessary subgraphs. In addition, the output of any node can be forwarded to the active state in the FSM, which can be used to trigger a state change. More examples of how the FSM and the flow graph can interact will follow in Chapter 4.

3.6 Exemplary Usage

3.6.1 Conditional Branching

One of the most basic control flow building blocks of a program are conditional statements. We can model an `if`-statement using SDF alone, as we demonstrate in Figure 3.20.

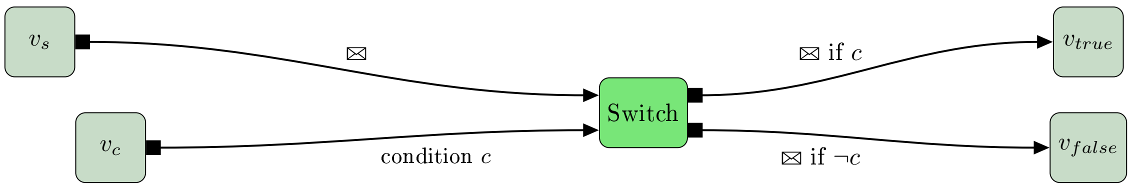


Figure 3.20: The *switch* node takes a boolean and forwards an incoming token based on the boolean's value on one of two possible outputs.

A source node v_s generates a message, which is sent to the *switch* node, which also gets a boolean message c from some other node v_c. Based on the value of c, the switch node either sends the message to v_{true}, if c is `true`, or to v_{false} if c is `false`. The other node receives a $\perp$ token.

3.6.2 Filter

Similarly to the conditional branching example, it is straight-forward to implement a filtering node using the $\perp$ token in combination with execution pruning. Figure 3.21 shows the interface of a generic *filter* node.

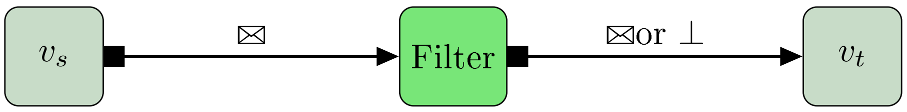
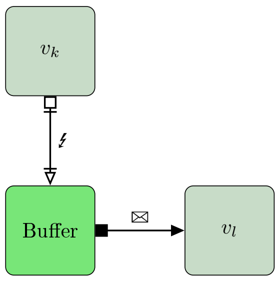


Figure 3.21: The ⌧ token can be dropped by the filter node, resulting in a $\perp$ token.

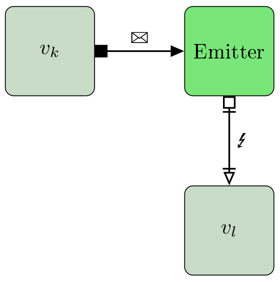
Let us, for example, consider a *throttle* node that receives a video stream and forwards only every n-th image. This can easily be achieved by an internal state variable in the implementation of the *filter* node, where we ensure to send exactly $n-1$ tokens of type $\perp$ messages before relaying another ⌧. In graphs where the $\perp$ token can safely be pruned, this results in a negligible overhead.

3.6.3 Translation between Asynchronous and Synchronous Flow

So far we have viewed SDF and asynchronous events separately, since connections between events and inputs, or outputs and slots are not allowed in the SDF+ model. It is,



(a) The *buffer* family of nodes converts signals to synchronous data flow.



(b) The *emitter* nodes convert data flow to events.

Figure 3.22: Special nodes can translate between SDF and asynchronous events.

however, possible to translate between the two using additional nodes. For this we define two families of nodes: *Buffers* and *emitters* (cf. Figure 3.22a).

Buffer nodes are nodes that are *sources* in the synchronous data flow and receive the data they produce via an asynchronous slot. Buffers can vary in their behavior, e.g. they can repeatedly publish the latest token they have received (aka. *latching*). They can also just publish the latest node once. Both these behaviors effectively act as a *decoupling* device, since a high frequency signal stream can be translated in a low frequency SDF or vice versa. In contrast, emitter nodes (Figure 3.22b) are used to translate SDF into events, e.g. when a token fulfills some specific criteria. This translation between the two types of data flow has to be done explicitly, there is no direct way. We address this issue in Chapter 5, where the two data flow paradigms are merged.

3.6.4 Parameter Optimization

In the SDF+ model, parameters are first class objects - they fully participate in the data flow. This enables many meta applications, one of which is parameter optimization. Here the key idea is that we can generate a set of parameters $\mathcal{P}$ and evaluate the performance of a SDF+ graph G' by calculating a *fitness* $E(\mathcal{P})$ value for the parameters. By minimizing E, we can optimize the behavior of G'.

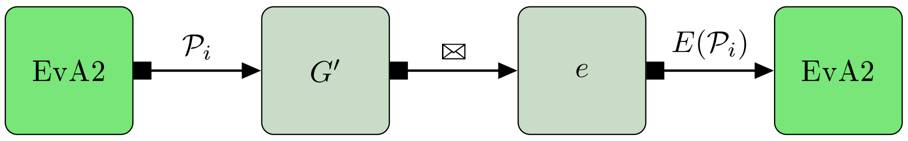


Figure 3.23: Evolutionary optimization of a graph G' using EvA2.

One way, visualized in Figure 3.23, is to use *Evolutionary Algorithms* (EA), e.g. from EvA2 by Kronfeld *et al.* (2010), which randomly generates populations of parameters and mutates them according to selectable mutation laws. In each iteration, the EA generates a parameter set $\mathcal{P}_i$ which is transferred to the graph G', which we want to be optimized.

The graph generates messages which flow to an evaluation node e, which has to be implemented for each specific optimization. Once the fitness $E(\mathcal{P}_i)$ is calculated, it is then returned to EvA2, which uses it for the EA. This process can then be repeated indefinitely, until an optimal parameter set $\mathcal{P}^*$ is determined.

As can be seen from Figure 3.23, only a single new node has to be implemented to optimize a new problem: The fitness calculation node e. It is therefore easy to optimize the free parameters of any graph constructed using SDF+, given that such a node e can be implemented.

3.7 Conclusion

The SDF+ model uses nodes to merge both synchronous data and event messages in a single process. Each node processes all its inputs at the same time, which means that there is no way to achieve asynchronous processing of individual inputs. This way of processing is only possible using event-based message passing. A single node can receive an arbitrary amount of events or trigger several events itself. There is, however, no way to handle an event in a synchronous sub-graph, which means that the sender of an event will immediately continue operating, once a triggered event has been posted to all its children.

Another limitation of using SDF and events separately is that every node in each connected SDF component of the graph has to operate at the same frequency. Although it is possible to translate messages into events, there is no way to achieve guaranteed synchronous data processing with varying frequencies.

Many problems can be solved by SDF+, i.e. using synchronous data flow and event-based message passing separately. Translating between the two ways of passing messages, however, requires additional nodes in the graph. Additionally, there are some problems that cannot easily be solved: Homogeneous SDF is incapable of modeling loops. This also means that it is not possible to model iteration of a set of messages. Instead, such looping and iteration has to be done inside a single node.

We address these limitations in Chapter 5, where we formulate a unified data flow model. In addition, we propose in Chapter 7 a way to also eliminate of the use of a separate FSM model by introducing additional properties. In the next chapter, we show an example application of the SDF+ model to model a fetch-and-delivery robot.

Even though the SDF+ model has many limitations, it can be applied to many robotic systems. Figure 3.24 shows a real-world SDF+ graph that was developed for the participation in the SICK robot day 2014 competition. The graph is shown as a screen shot of the CS::APEX graphical user interface, which allows the construction, evaluation and application of SDF+ models. Problem specific nodes in the graph are shown in blue, the other nodes are completely generic and show the re-usability aspects of the graph based system. The individual components shown are described in detail in the following chapter. The architecture of the CS::APEX framework, which is presented in Chapter 9, also allows the construction of HFG and AFG graphs, as derived in Chapter 5 and Chapter 7.

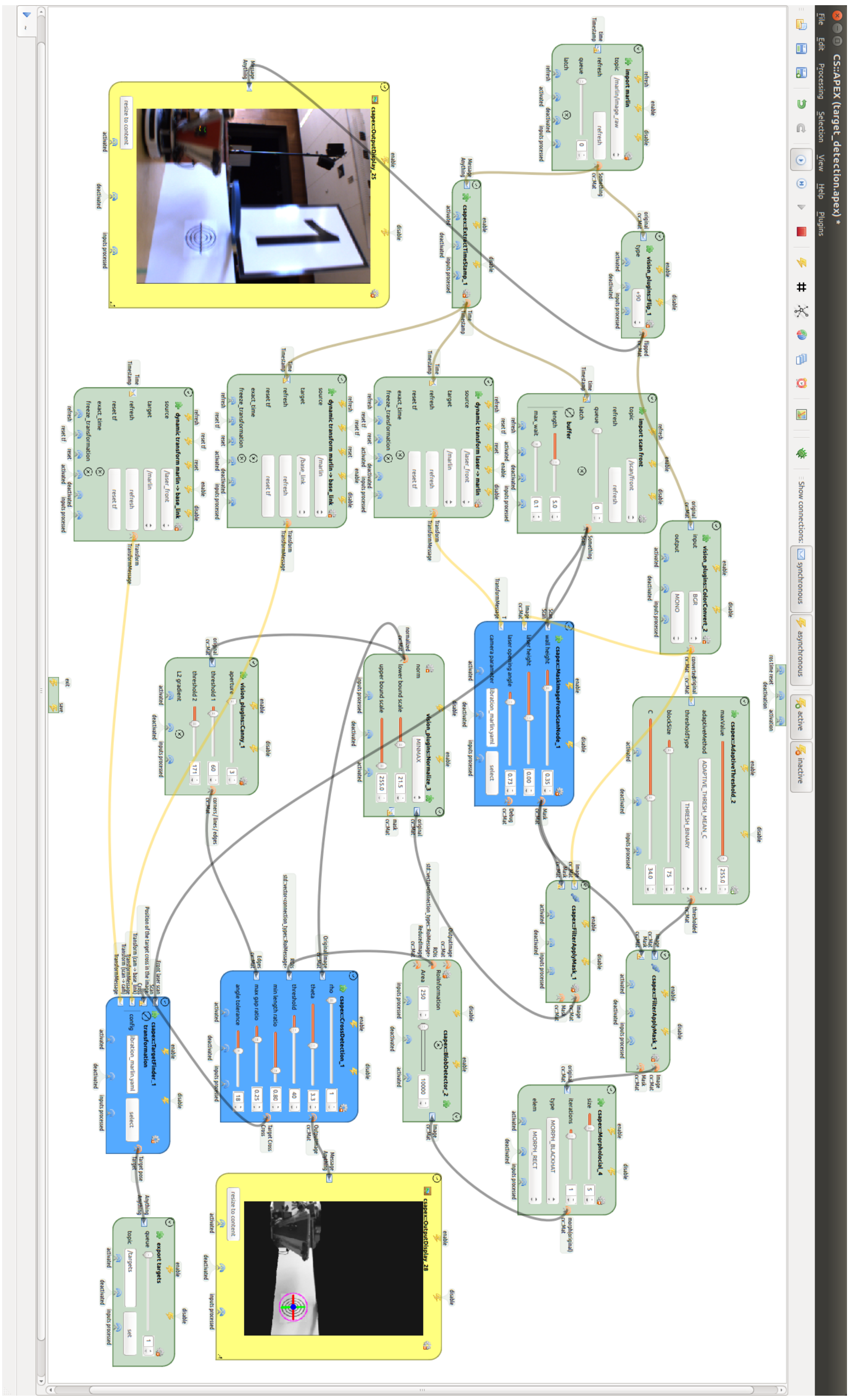

CS::APEX (target_detection.apex) *
File Edit Processing Selection View Help Plugins
import marlin
vision_plugins::Flip_1
csapex::ExtractTimeStamp_1
csapex::OutputDisplay_25
import scan front
buffer
dynamic transform laser -> marlin
dynamic transform marlin -> base_link
vision_plugins::ColorConvert_2
csapex::AdaptiveThreshold_2
csapex::MaskImageFromScanNode_1
vision_plugins::Normalize_3
vision_plugins::Canny_1
csapex::FilterApplyMask_1
csapex::BlobDetector_2
csapex::CrossDetection_1
csapex::TargetFinder::transformation
csapex::Morphological_4
csapex::OutputDisplay_28
export targets
Show connections: synchronous asynchronous active inactive

Figure 3.24: Screen shot of a fully functional SDF+ graph in CS::APEX. The graph performs several computer vision calculations to detect bull's eye signs for the SICK robot day 2014 competition, which is described in detail in Chapter 4. The graph was designed and evaluated using the GUI, which was turned off during the competition to reduce computational overhead.

Chapter 4

Modeling a Fetch-and-Delivery Robot

4.1 SICK Robot Day 2014

Participation in a competition poses multiple constraints on the design and implementation of a robotic system. Contrary to many other situations, deadlines cannot be extended and the system has to perform autonomously on the first try. Every part of the system has to perform at the same time and every part has to collaborate with every other part reliably. Competitions are therefore a challenge that motivate a design that is simple and efficient, yet robust to influences that cannot be controlled or anticipated. Smaller robotics competitions are an exercise in team work and act as benchmarks, just as well as larger contests like RoboCup, the DARPA challenge and others (Behnke, 2006).

The SICK Robot Day is a bi-annually hosted competition by the SICK AG, Waldkirch, Germany, a well known producer of sensor systems. In the past the objectives have varied notably, from perception and interaction, as described by Scherer *et al.* (2011), Masselli *et al.* (2013), Cigolini *et al.* (2013) and Fejfar and Obdržálek (2014), to navigational tasks as depicted by Fredriksson *et al.* (2007). Additional difficulty has always been caused by multiple competing robots performing simultaneously in an arena that is small enough to provoke robots encountering each other.

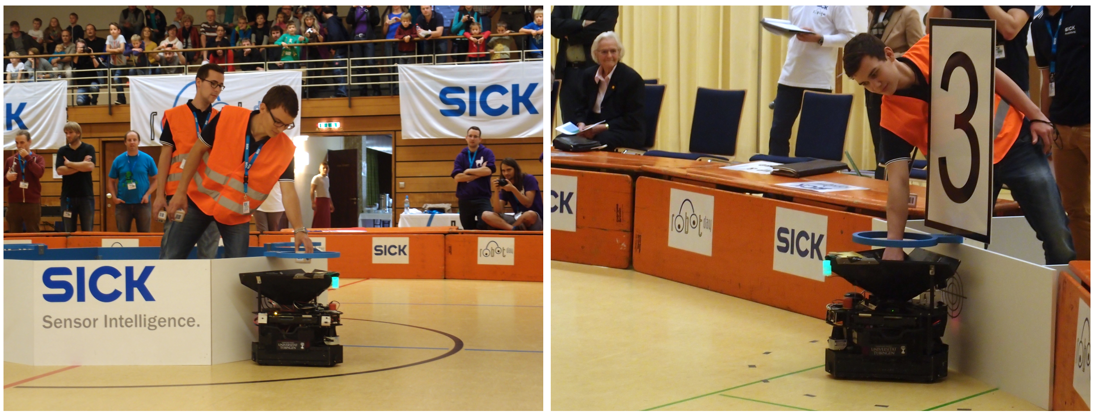
In this chapter we describe our design for the participation in the SICK Robot Day 2014, which is based on the SDF+ model. This time, the challenge was to fetch and deliver objects in a scenario resembling the tasks in an automated warehouse, where four robots were competing in a relatively small area at the same time. Besides a mechanism for collecting and delivering the objects, participating robots had to be able to detect filling- and delivery stations, navigate autonomously, and at the same time avoid collisions with other robots. The goal was to deliver as many objects as possible, where each correctly delivered object was awarded one point and each erroneous delivery one penalty point.

The rest of this chapter is structured as follows: Section 4.2 states the requirements posed by the rules of the competition. Section 4.3 illustrates the design of the resulting system and presents the different components of the SDF+ model. Section 4.4 presents experimental results. Section 4.5 summarizes the conclusions. This chapter is based on the articles published in (Buck *et al.*, 2015) and (Huskić, Buck, and Zell, 2016).

4.2 Requirements

The competition was held in an approximately circular arena with about 12 m diameter in which four robots competed at the same time. With a time limit of 10 minutes, each robot had to alternately collect labeled objects at *filling stations* and transport them to *delivery stations* based on the object label (cf. Figure 4.1). The octagonal collection area, where the robots had to approach one of four filling stations, was located in the central part of the arena. Once a robot had reached the designated filling spot, it had to signal the human operator to provide an object to be delivered. These objects were known to be wooden cubes of side length 6 cm, which were labeled on each side with the designated delivery station using a bar code imprint. On the outer boundary there were four delivery stations, one for each possible object label. Whereas the filling stations could be selected freely, every delivery station could only be used to hand off one specific object type. Therefore robot-robot interactions were inevitable and had to be accounted for.

Both filling and delivery stations comprised of a ring of diameter 30 cm at a height of around 50 cm and a bull's eye target sign for precise positioning (cf. Figure 4.1b). Delivery stations were additionally marked with a large sign displaying one of the digits printed onto the wooden cubes. The robots therefore had to be able to detect both the bull's eye and the number signs. Once a robot had reached a station, it had to activate a green signal lamp to indicate its intention to collect or deliver an object. A human operator would then insert or remove a cube within at most 10 seconds.



(a) The robot receives an item. (b) The item is removed at a delivery.

Figure 4.1: The robot signals the operators to add or remove an item using a green light.

To successfully participate, a robot therefore had to fulfill the following requirements:

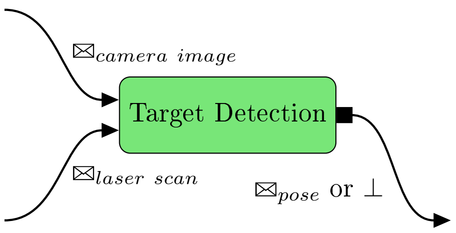
- Detection of signs displaying the digits 1 to 4, as well as bull's eye target signs
- Receiving a cube and reading the imprinted bar codes
- Localization and navigation in a shared space with three other robots

4.3 Design and Implementation

Here we present the final system design and implementation we used for participating in the SICK Robot Day 2014. All perception tasks are implemented using the SDF+ model. For each task we first present an interface describing the inputs and outputs of the graph, as well as the major components of the graphs.

4.3.1 Target Detection

The primary part of localizing filling and delivery stations is the detection of bull's eye patterns as seen in Figure 4.3a. These patterns are used to accurately position the robot below wooden rings through which the objects are thrown by the human operators.



$\boxtimes_{camera\ image}$
Target Detection
$\boxtimes_{laser\ scan}$
$\boxtimes_{pose}$ or $\perp$

Figure 4.2 shows the final SDF+ implementation, which corresponds to the abstract interface. Since the height of the surrounding border of the arena is known to be about 50 cm, we first use a mask to quickly dismiss large parts of the camera image by extracting a band of 35 cm around the expected height of target signs. This is achieved by projecting each point p_i of the front laser scanner onto the camera image using the camera calibration matrix A and the static transform between camera and laser scanner frames ${}^{C}T_{L}$ to calculate the image coordinates $(u,v)_i$.

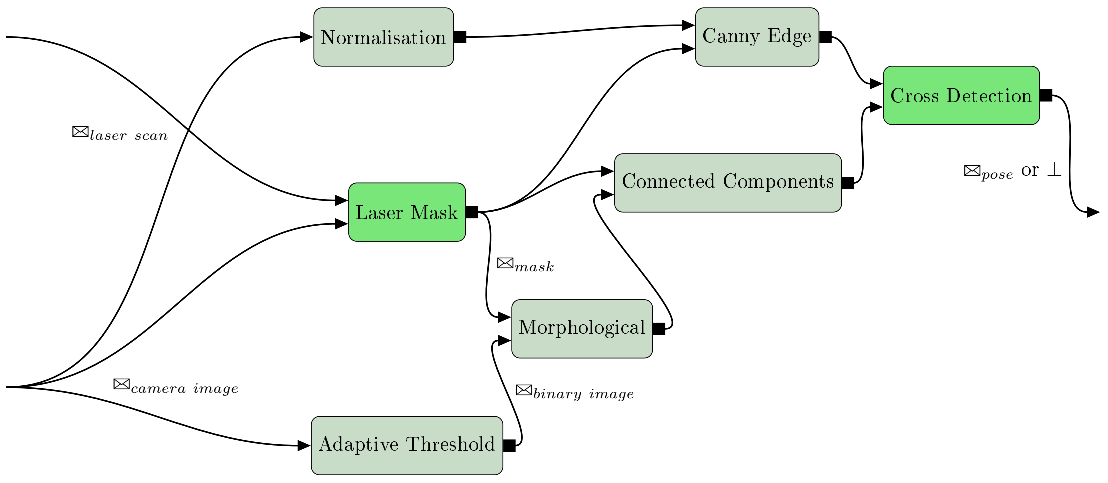
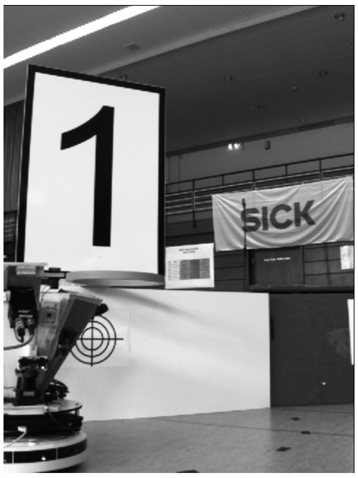
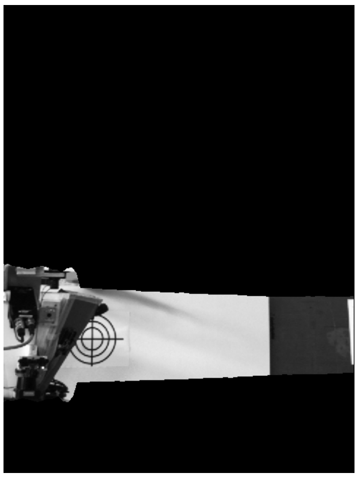


Figure 4.2: The SDF+ graph that implements the target sign detection has the camera image and the current laser scan as inputs and returns the pose of the target, or $\perp$ if the target is not found. Highlighted nodes are problem specific.

The points $(u,v)_i$ are calculated twice, first for a height of $z=0$ and then for $z=35$ cm, resulting in two sets L_{low} and L_{high} where L_{low} corresponds to the projection of the laser scan and L_{high} is shifted upwards. The polygon between L_{low} and L_{high} is then filled to generate a matte as shown in Figure 4.3b. The masked image is then examined for regions of interest by first applying a blackhat morphological operator and then performing a connected component analysis using cvBlob (Linán, 2014). Large components are normalized to improve robustness against lighting changes and are then further processed.



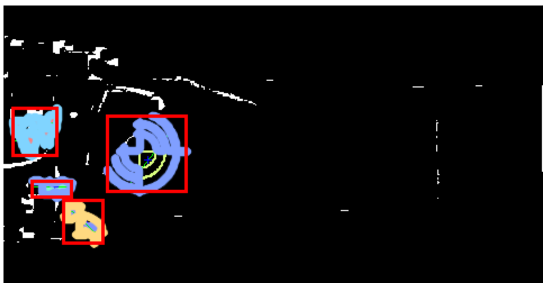
(a) Number sign and partly occluded bull's eye.



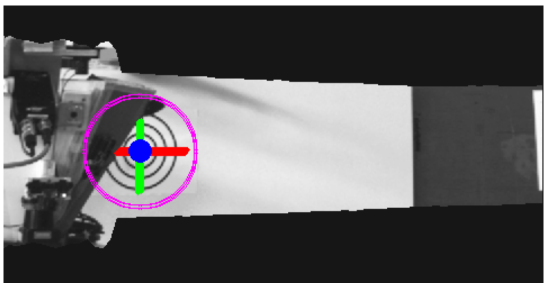
(b) Image masked based on laser projection.

Figure 4.3: Bull's eye detection is performed on regions of interest.

We calculate Canny edge features (Canny, 1986) and apply the probabilistic Hough transform (Matas *et al.*, 2000) in order to extract the horizontal and vertical line in the pattern. Detected lines are processed using various heuristics to reject short or skewed lines. Remaining intersecting lines with an angle of intersection near 90° are classified as targets, resulting in a situation as shown in Figure 4.4b. In the end, only two problem specific nodes have to be used.



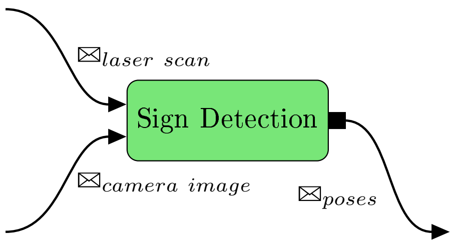
(a) Regions of interest.



(b) Resulting detection of a target.

Figure 4.4: Dark patches with light backgrounds patches are then extracted and analyzed.

4.3.2 Sign Detection



Delivery stations are marked with signs displaying the numbers 1 to 4, whereby the number represents the type of cube that can be delivered. Figure 4.5 shows the graph to detect these signs and read the depicted number. To solve this problem we compared two *multilayer perceptron* (MLP) neural networks, the one we used at SICK robot day 2010 based on the Stuttgart Neural Net Simulator (SNNS) which was described in detail by Scherer *et al.* (2011) and a later approach using JANNLab by Otte *et al.* (2013). The need for an alternative classifier stemmed from the observation that some numbers could not be detected as well as others by the existing network.

To optimize the runtime cost, we also use the laser projection mask as described in Section 4.3.1. Here we extract a band above the wall in which the signs are expected, which further reduces costs compared to the previous approach. Detected signs are then mapped for future reference, such that the robot does not have to search for the appropriate sign for every delivery run. We use a Gaussian mixture model to be able to model hypotheses in a way robust to false detections. For the model of a sign we use a two dimensional Gaussian distribution for the x-y position and a von Mises distribution for the orientation. As shown by Bishop *et al.* (2006), the maximum-likelihood solution $\bar{\theta}$ for the orientation of a hypothesis is then given by

$$\bar{\theta} = \operatorname{atan2}\left(\frac{1}{N}\sum_{i=1}^{N}\sin(\theta_i), \frac{1}{N}\sum_{i=1}^{N}\cos(\theta_i)\right), \tag{4.1}$$

where θ_i are the individual measurements of the orientation and N is the number of measurements. For path finding we refer to this sign map and choose the target pose based on the best hypothesis.

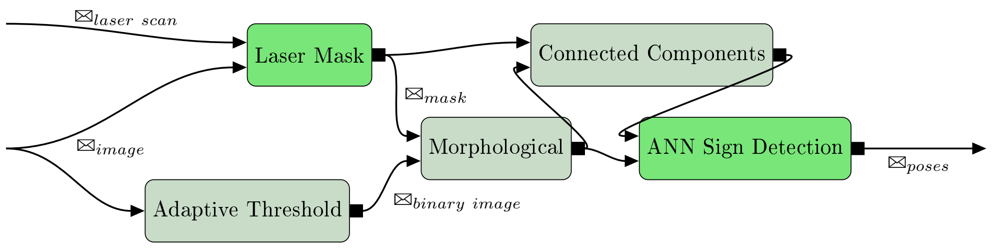
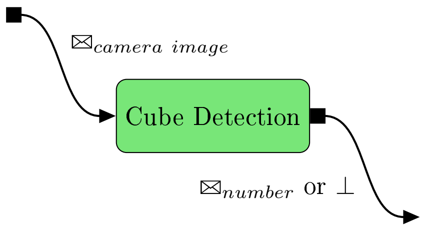
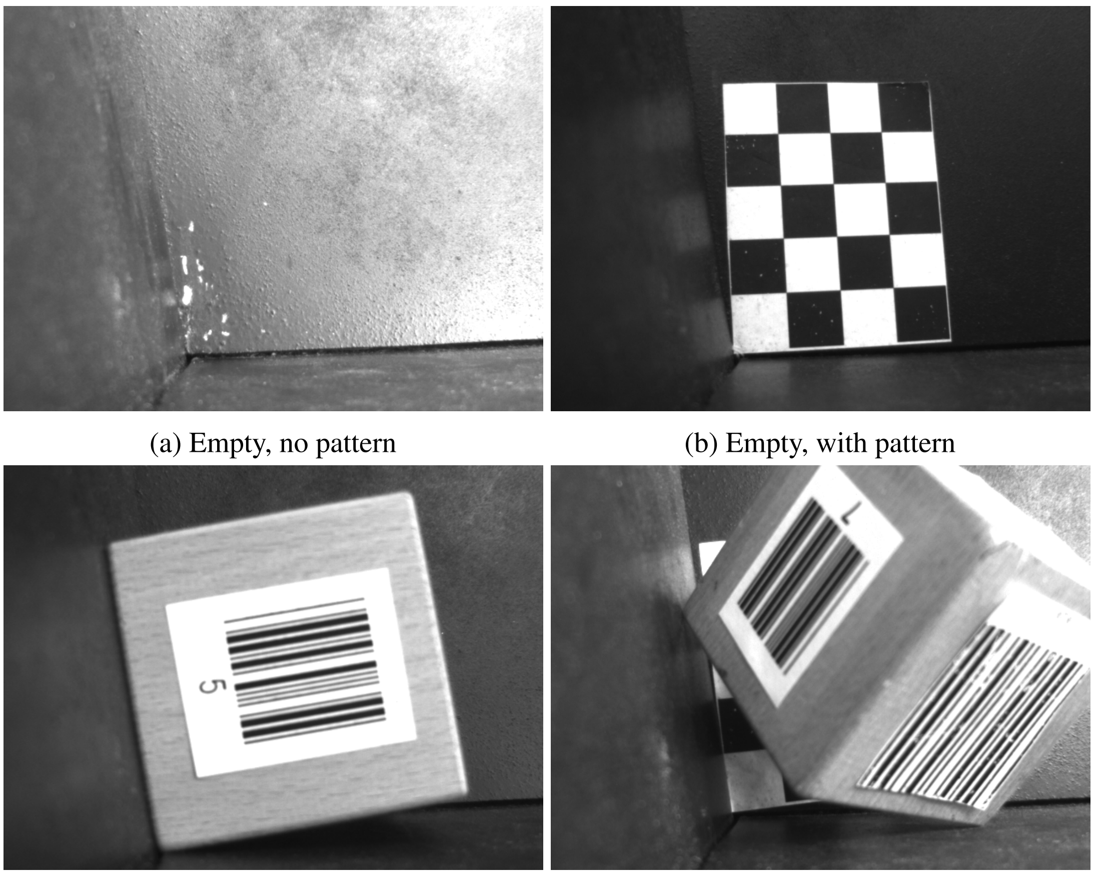


Figure 4.5: The SDF+ graph that implements the number sign detection. Highlighted nodes are problem specific.

4.3.3 Cube and Bar Code Detection



After collecting a new cube, the target delivery station has to be determined. To get the numerical value of the cube's bar code we use the ZBar library (Brown, 2013). Bad lighting conditions or a wedged cube can cause a failure of the bar code analysis. Therefore we use a robust secondary system to frequently check whether an object is in the hopper or not. Figure 4.6d displays such a case in which no bar code can be detected even though a cube is in the basket.


(a) Empty, no pattern (b) Empty, with pattern

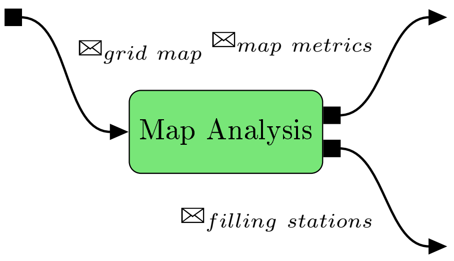
(c) Cube in optimal position (d) Cube wedged

Figure 4.6: The cube hopper camera scene.

Our first approach to realize a cube detection was to calculate the standard deviation of the intensity values. The robot's hopper is painted black, so the standard deviation of the empty camera scene is minimal, as can be seen in Figure 4.6a. The moment a cube enters the scene, the standard deviation rises considerably. A problem with this approach is, however, that it heavily relies on a well configured camera which adapts to the lighting conditions.

As JANNLab is already a part of the software infrastructure, we therefore implement a more robust solution based on neural networks. We use a multilayer perceptron neural network (MLP) to detect the characteristics of the pattern seen in Figure 4.6b independently of the lighting conditions and resulting image brightness. The network is trained using data generated under several different lighting conditions. We also take conditions into account which are more extreme than those expected at the competition. As input for the MLP, we use a normalized vector of 100 intensity values, extracted from the pattern within the scene.

4.3.4 Map Analysis



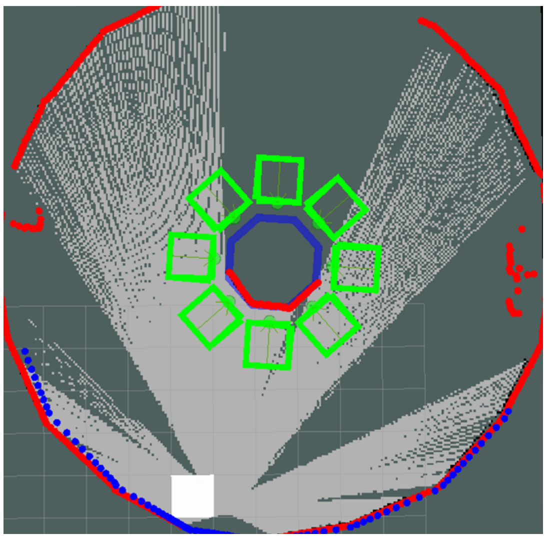
One important reason for performing SLAM instead of driving reactively is to be able to use a map of the environment for determining the positions of filling and delivery stations. Delivery stations are assumed to be on the arena border. In case a station is unknown, the robot has to scan the border systematically by driving counter-clockwise around in the arena while looking in the tangential direction. The filling stations are placed on every second side of an octagonal island at the center, whereby the side facing the starting position of the robot is guaranteed to be a station.

To determine the exact positions of the four filling stations, we first classify every occupied cell in the grid map into *center* and *border*. Hereby we assume that the segmentation filter described in Section 4.3.5 is able to completely remove every robot from the laser scans such that the map only contains occupied cells for walls. On all cells labeled as *center* we perform RANSAC with an octagon model: We sample two points and assume that they are the endpoints of one side s_1 of the octagon. We then extend s_1 in the direction of the centroid of the map to a full octagon by adding s_2 to s_8 and checked the hypothesis.

Using the best hypothesis model, we select side s_i as a filling station side, for all i which minimize the angle between side s_i and the starting position. We then add the appropriate three remaining sides and check a rectangular box in front of the filling station sides for laser scan hits in order to decide which filling station is free and which is in use. As shown in Figure 4.7a, we also perform this procedure for the remaining sides, because we had assumed a square center and only had limited time to change the algorithm.

4.3.5 Localization and Mapping

To localize the robot in the arena, a graph based simultaneous localization and mapping (SLAM) algorithm is used. This module is not implemented using the SDF+ model. We decided against a pure localization algorithm, because the exact map of the arena was


(a) Analysed map showing the detected octagon and 8 free sides

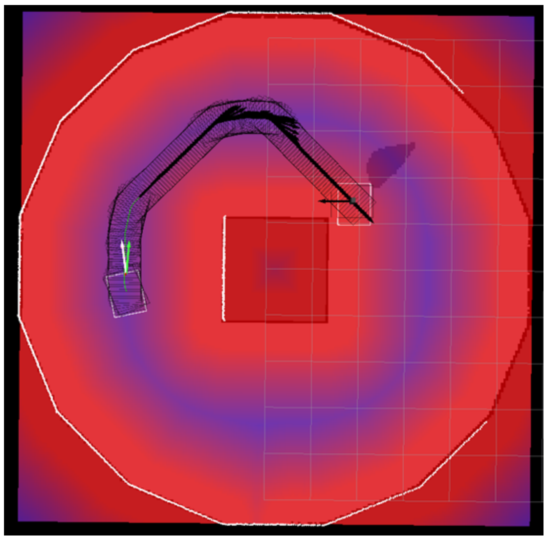

(b) Cost map (simulation) used for path planning. High costs are red.

Figure 4.7: RANSAC analysis of the grid map given by SLAM and generation of a cost map using the distance transform.

unknown before the competition and we did not want to depend on prior information. A SLAM algorithm using only scan matching without odometry information, like the open-source module of the Hector framework by Kohlbrecher *et al.* (2014), did not work, because the arena was too symmetric to correctly detect movements from the scans. Out of the evaluated open-source implementations of graph-based SLAM, Karto (Vincent *et al.*, 2010) was found to be the most accurate for our case. However, this algorithm assumes a static environment and is usually used for exploration and not for long term use, which required us to implement extensions.

The requirement of a static environment is not given, since there are other robots in the arena that move and sometimes stand still. If another robot is not moving for a certain time, it will be included as occupied space in the map, which would then be remembered by the algorithm for several iterations, even after the robot moves away. This would complicate path planning by adding unnecessary costs to the cost map. To solve this problem we pre-process the laser scan with a jump distance segmentation, which filters a scan using the difference $\Delta r = |r_k - r_{k+1}|$ of two neighboring rays r_k and r_{k+1}. If Δr exceeds a threshold T a new segment is started. A segment is removed if the euclidean distance of its start and endpoint is below a minimum value S_{min}.

The second problem is that Karto stores all matched scans. Every time a map is requested from the path planner, all n stored scans have to be integrated into a grid map. Therefore, the computational complexity and the time to provide a map increases linearly with the number of stored scans. Since long waiting times would cause time-outs in the path planner, we limit the map creation time without decreasing the localization accuracy. This is done by limiting the number of scans being used for a map update. To also include some older scans in the map, scan selection for n scans is divided in to three intervals: A dense interval comprises the latest scans s_k with $k \in [n-100, n)$. Additionally, a semi dense interval with $k \in (n-200, n-100)$ and $k = 0 \pmod{10}$ is considered. Finally, we consider a sparse section with $k \in (n-1000, n-200)$ and $k = 0 \pmod{100}$. In the background all scans are kept in the graph to achieve high localization accuracy from loop closures and to avoid drifting of the map.

4.3.6 Navigation

We use a full navigation stack including path planning and following, that has been developed to work with all of our robot platforms. For this implementation, the navigation stack is not using the SDF+ model. The map generated by the SLAM subsystem is preprocessed and then used for path finding. The resulting path is post-processed and then used by a path following controller.

Map Preprocessing

Before every request $r = (s, g)$ for a path from starting pose s to goal pose g, our navigation stack requests a grid map representation m of the current pose graph. The map m is treated as an image and submitted to a morphological convolution to grow obstacles and close small unknown regions, whereas circular regions around s and g are kept unchanged to allow planning as closely to obstacles as possible. Afterward, a combined map m' is computed by integrating the current readings from both laser scanners into the map m. Using m' we compute a cost map m_c, by calculating the distance transform and then scaling the values such that costs are maximal at obstacles and go to 0 for distances larger than 2.5 m, which can be seen in Figure 4.7b.

Path Planning

Given the map m and a cost map m_c, we use a variant of A^* using the simple L_2 heuristic given by the norm $||\cdot||_2$. Since our robots are omnidirectional, we also implement a simple omnidirectional motion model using the natural 8-neighborhood of a grid map. For the past-cost we use $g'(q) = g(q) + m_c(q)$, where $g(q)$ denotes the known distance from s to the current configuration q and $m_c(q)$ the cost of configuration q. We manually tune the scale of m_c such that resulting paths stay as far away from obstacles as possible while not completely ignoring configurations q with $m_c(q) > 0$.

In a post-processing step we use a heuristic to simplify path segments by recursively removing nodes in regions with no costs associated and checking if the resulting path is still valid. This is motivated by the fact that slanted paths in an 8-neighborhood are often longer and more ragged than necessary due to only being able to represent multiples of 45° when expanding a node. After simplification we perform path smoothing and interpolation to allow for a better path following.

Path Following

The path following algorithm is based on the ideas described by Mojaev and Zell (2004), which was further extended in Li (2009). We use a simple and efficient control law based on the orthogonal projection of the vehicle to the planned path, which is shown in Figure 4.8.

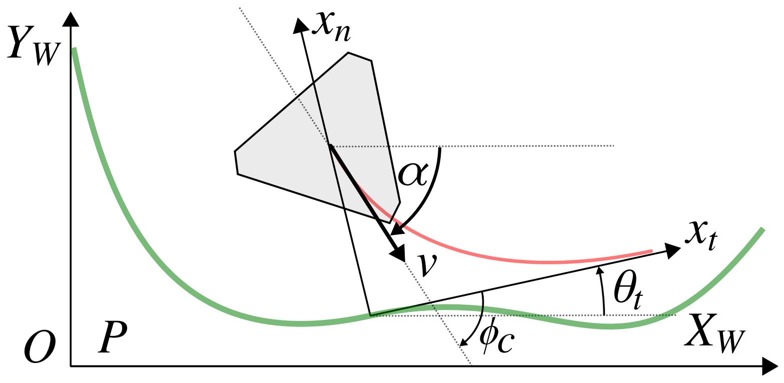


Figure 4.8: The principle of the orthogonal projection path following algorithm.

The orthogonal projection x_n is defined as an exponential function of the tangential component x_t in the projected point on the path. Let x_{n_0} be the initial distance from the path, then we get

$$x_n = x_{n_0} \exp(-kx_t), \tag{4.2}$$

where k is a positive constant which regulates the convergence speed of the orthogonal projection. The tangential angle of the exponential function is given by

$$\phi_c = \tan^{-1}(-kx_t). \tag{4.3}$$

If we consider that the robot is omnidirectional, its linear velocity in the world frame can be described as

$$\begin{aligned} v_x &= v\cos(\alpha), \\ v_y &= v\sin(\alpha), \end{aligned} \tag{4.4}$$

with $\alpha = \phi_c + \theta_t$. Here, v is the linear velocity, θ_t the tangential angle of the desired path and α is the driving direction angle (cf. Figure 4.8).

For the given scenario, we extend the algorithm by adding velocity control (Huskić, Buck, and Zell, 2016). The angular velocity is controlled with a PID controller, independently of the linear velocity. In contrast to (Mojaev and Zell, 2004) and (Li, 2009), the linear velocity is dependent on the curvature, the distance to the obstacles, the distance to the goal and the angular velocity. The basic idea from Maček *et al.* (2005) is slightly changed, so that the velocity changes exponentially. The combined velocity is then given by

$$v = v_n exp\left(-\left(K_\kappa \sum_{i=i_P}^{l} ||\kappa_i|| + K_\omega|\omega| + \frac{K_o}{d_o} + \frac{K_g}{d_g}\right)\right) \tag{4.5}$$

Here, v_n is the nominal velocity, the constants $K_\kappa, K_\omega, K_o, K_g$ are used to adjust convergence, i_P is the index of the currently projected point on the path, and l is the index of the look-ahead distance. The curvature in a certain point is κ_i, ω is the angular velocity, d_o the distance to the nearest obstacle, and d_g the distance to the goal.

Obstacle Avoidance

The path planner is fast enough to allow a simple obstacle avoidance scheme: Once we detect an obstacle that stays on the current path for too long, the robot stops and issues the system to generate a new path. We have developed several layers of safety: The path planner prefers paths that stay away from obstacles because of the cost map. The control in Equation (4.5) decreases the velocity of the robot in the vicinity of obstacles. We additionally employ a safety field in the driving direction, such that if an obstacle is detected in this zone, the robot is stopped completely.

4.3.7 Finite-State Machine

For high level robot control we again use the SDF+ model, i.e. we implement a finite-state machine, which is depicted in Figure 4.9. After waiting for a user to give the "go" signal by pressing a sequence of buttons on the remote control, the robot is completely autonomous.

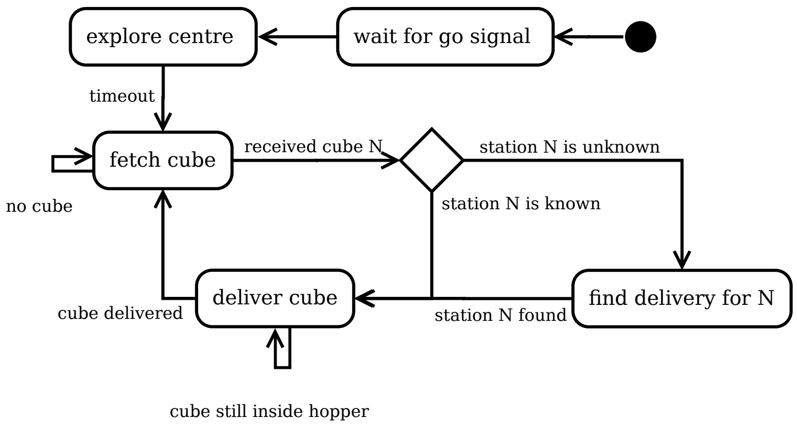


Figure 4.9: The finite state machine we implemented. Error states are omitted.

A first state is introduced to explore the central area, such that the map analysis component is able to determine the position of the filling stations. This initial exploration consists of driving a predetermined distance diagonally in order to allow the SLAM system to build up a map. Afterwards we iterate fetching and delivering cubes, where both states consist of several sub-states.

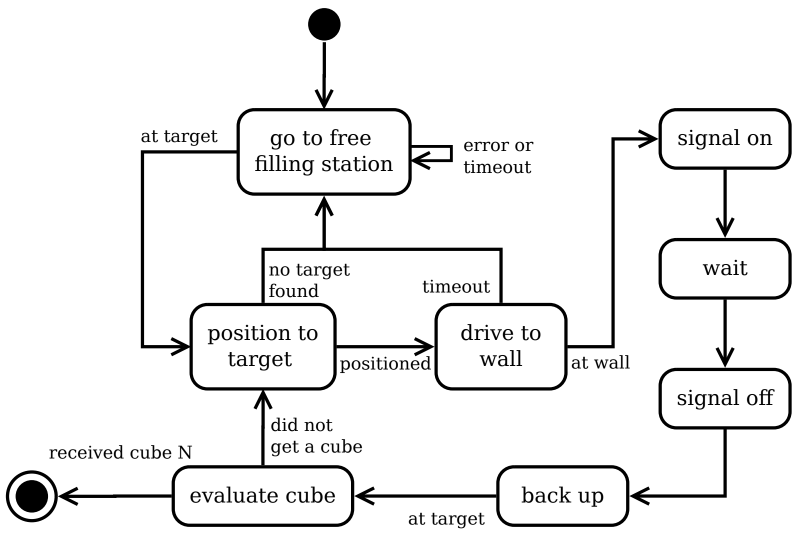


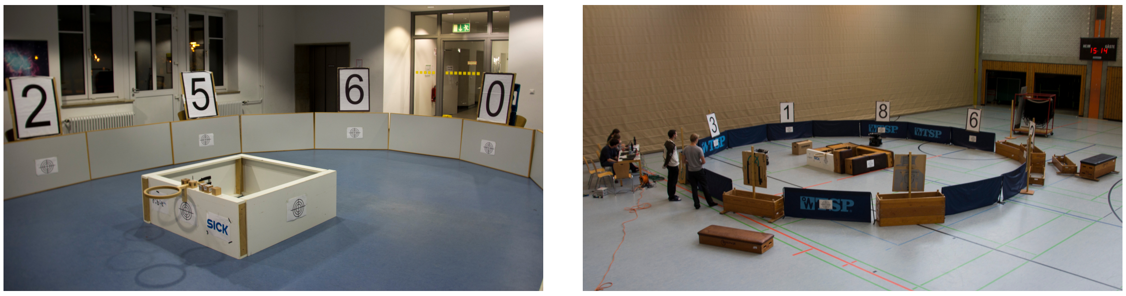
Figure 4.10: The meta state *fetch cube* in Figure 4.9.

Fetching a cube, visualized in Figure 4.10, includes selecting a free filling station, as well as planning and following a path there. Once the calculated target pose is reached, the robot uses the bull's eye target detection to orientate itself towards the station. It then drives forward until the distance to the wall is below 5 cm, signals the human operators and then backs up again. If at this point no cube had been received, e.g. if the target position is not accurately reached, the robot issues a retry. Otherwise the cube is analyzed and the sub-state is left.

Delivering a cube is essentially the same as fetching a cube, only instead of going to a free filling station, the path finding module is requested to plan a path to the appropriate target station if it is known. Otherwise we switch to finding this unknown station by driving around the arena in the counter-clockwise direction.

4.4 Experimental Results in the Lab

Experiments have been performed in our laboratory (cf. Figure 4.11a) and in a gym of the University of Tübingen (cf. Figure 4.11b). Some experiments have been repeated after the competition for more precise analysis.



(a) The experimental setup in the first floor lobby of our building Sand 1.

(b) The setup in the gym of the university.

Figure 4.11: The setup for experiments closely matches the competition arena.

4.4.1 Target Detection

The performance of the bull's eye target sign detection was acceptable for the competition. There is an inherent downside to our line extraction approach due to motion blur. Even at relatively low rotational speeds, motion blur was too much of an influence for target signs to be detected at larger distances. However, we only needed to detect these signs in the final step of approaching a station. The map analysis was sufficient to allow planning a path that ended directly in front of a filling station, where the signs were clearly visible. In the case of delivery stations we relied on detecting number signs at larger distances and then used path finding to position the robot in front of these number signs, where again the target signs were easily visible. That is why the targets only had to be robustly seen at distances of 2 m or less, in which case motion blur did not affect the detection rates.

4.4.2 Sign Detection

For the evaluation of the sign detection we refer to Scherer *et al.* (2011). The approach has not been modified, except for masking out irrelevant parts of the camera image, which had an impact on the runtime but not on the classification performance. The new neural net based on JANNLab was trained to achieve higher detection rates and worked fine in our test environments, resulting in more true positives and fewer false positives. On site at the competition, however, there were a few reproducible false positive detections caused by the environment. This is why we decided to use the old neural net, for the benefit of fewer false positives.

4.4.3 Cube and Bar Code Detection

The cube detection was evaluated on datasets containing images taken under several different poses and differing lighting conditions. In Table 4.1 the results of the evaluation can be observed. The standard deviation based approach was tested on 1306 frames without the background pattern used by the second approach. The variance threshold was set to 50 to detect an object within the camera scene. The MLP based approach was tested on 1325 different frames. The results show that the MLP based approach is more robust.

Std Dev. based		Prediction	
		empty	*full*
Actual	*empty*	563	150
	full	158	435

MLP		Prediction	
		empty	*full*
Actual	*empty*	662	14
	full	1	648

Table 4.1
CONFUSION MATRICES FOR CUBE DETECTION APPROACHES.

4.4.4 Localization

The segmentation filter produced good results because the maximum robot size was limited to 0.6 m by the rules. We set the segment size minimum value to $S_{min} = 1.2\,\text{m}$ to be sure that a segment is removed even if two robots are standing close to each other. The threshold for the minimum distance of rays was set to $T = 0.1\,\text{m}$ so that small robots can still be separated from a wall, and the octagon is still seen as one segment. Our method of limiting the map creation time also showed good results as we could limit it to 62 ms, while the time increases with 0.5 ms per scan using the standard algorithm.

4.5 Results

The competition took place on October 11, 2014 in Waldkirch, Germany. Every attending team had a few hours before the competition to test their systems on site. The arena, which nobody had seen before, was mostly as expected. There were only four delivery stations as opposed to the expected worst case of ten stations. The size of the arena matched our expectation, yet the central area for filling stations was not as we had anticipated: The rules had specified that the filling stations would be located at the 4 sides of a square island in the center of the arena, which ended up being an octagon with 4 unused sides as described in Section 4.3.

Every robot had two runs of 10 minutes each, whereby the better run was to be used for ranking. Out of the 14 competing teams, five managed to score one or more points. Our robot managed to correctly deliver six cubes in ten minutes, resulting in a second place with as many points as the winner RIMLab with their robot PARMA[1]. The tie was broken by consulting the other run, in which our robot could only deliver three cubes, because of a malfunction which resulted in our robot standing still for five minutes after it had already moved for about a minute.

The delay was most likely caused by a driver problem we had infrequently observed a few times earlier, in which the driver for our main laser scanner would freeze up and not publish any new data. This is in accord with the log files, which show that the laser scanner reported obstacles directly in front of the robot when clearly there were none. We had implemented a watch dog process to restart everything related to the laser scanner subsystem in cases when no data was transmitted for a while, but it seems that multiple restarts were required.

There are a few optimization possibilities which would have allowed us to decrease the time per object in order to deliver at least one more object:

- Our robot waited for the full 10 seconds on both filling and deliver stations for safety reasons. For six delivered objects, this alone resulted in 2 minutes of waiting time, which mostly could have been avoided.
- To avoid CPU overload, which would have been detrimental to our SLAM module, we decided to using events to disable parts of the system when they were not needed. This effectively reduced the frame rates of our cameras and resulted in the need to wait in front of expected bull's eye signs to increase the confidence in the observation, which cost another unnecessary 5 seconds per station and about one minute overall.
- We used a maximum velocity of $1\,\mathrm{m\,s^{-1}}$ in the first run, which was already relatively fast compared to other teams but was only a safety precaution to absolutely avoid any dangerous situation.

For the second run we were able to quickly change our state machine and decrease waiting times to the absolute minimum. We also further increased the driving speed to $1.5\,\mathrm{m\,s^{-1}}$, however the effort was wasted because of the above mentioned inactivity problem.

In Figure 3.24 we have already seen a screen shot of the actual graph used for target sign detection. Figure 4.12 shows the second graph that was used to detect the number signs using a neural network.

[1]http://rimlab.ce.unipr.it/SickRobotDay2014.html

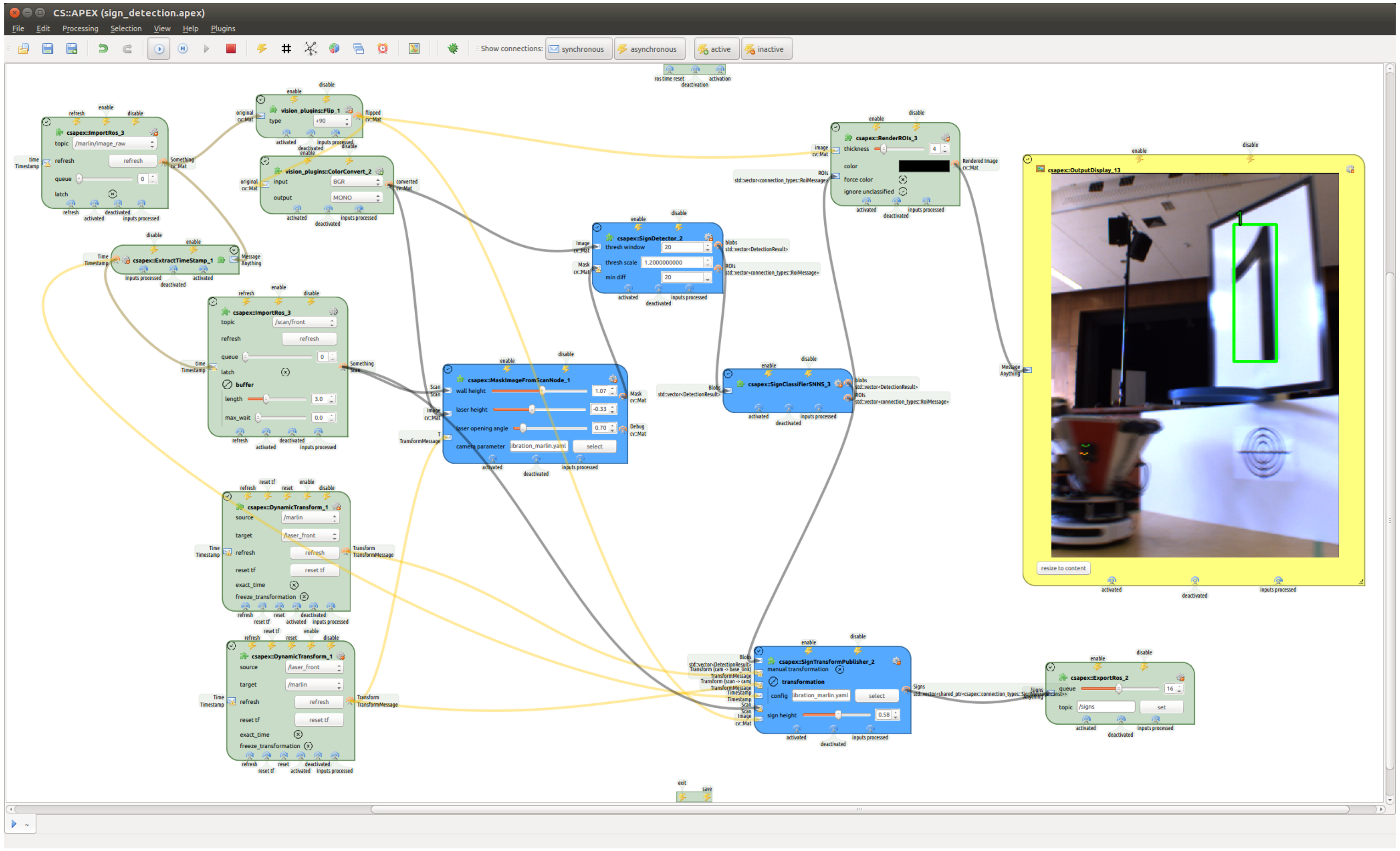

CS::APEX (sign_detection.apex)
File Edit Processing Selection View Help Plugins
Show connections: synchronous asynchronous active inactive
csapex::ImportRos_3
vision_plugins::Flip_1
vision_plugins::ColorConvert_2
csapex::ExtractTimeStamp_1
csapex::ImportRos_3
csapex::SignDetector_2
csapex::MaskImageFromScanNode_1
csapex::SignClassifierSNNS_3
csapex::RenderROIs_3
csapex::OutputDisplay_13
csapex::DynamicTransform_1
csapex::DynamicTransform_1
csapex::SignTransformPublisher_2
csapex::ExportRos_2

Figure 4.12: Screen shot of a fully functional SDF+ graph in CS::APEX. The graph performs several computer vision calculations to number signs for the SICK robot day 2014 competition using a neural network classifier. The graph was designed and evaluated using the GUI, which was turned off during the competition to reduce computational overhead.

Chapter 5

Hybrid Flow Graphs

The combination of synchronous data flow (SDF) and event-based message passing has been introduced as the SDF+ model in Chapter 3. Most limitations of the SDF+ model are linked to the distinction of the two flow types, which requires nodes in the computation graph to translate between synchronous data flow and asynchronous event flow. In this chapter, we introduce a unified graph model, which is a hybrid of SDF and Events, as visualized by the second layer in the flow hierarchy shown in Figure 5.1. The *Hybrid Flow Graph* (HFG) model is an extension of SDF and Events in a single and coherent model. HFG still distinguishes synchronous from asynchronous data flow, however it allows for direct connections between the two, without explicit translation nodes. This enables new use cases and overcomes many of the limitations of SDF+. Using hybrid connections allows the graph to be cyclic, which is not possible in the SDF+ model.

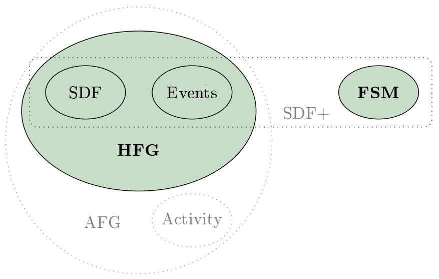


Figure 5.1: Hybrid Flow Graphs (HFG)

The HFG model is defined hierarchically. This means that a complete graph can be seen as a single node, or *sub-graph*, in a more abstract data flow graph. In contrast to SDF+, the HFG model is therefore able to represent iterative calculations by executing a sub-graph for each value in a list. This support for list processing enables more generic programming, since each sub-graph used for list iteration only needs to process an individual message and is therefore equivalent to a common node. There is no need for specialized list processing functionality, although such nodes can also be implemented easily.

While iteration can be used to model for-loops in an algorithm, indefinite while-loops are not easily representable in HFG. This means that complex high level state and control flow cannot be modeled in the graph. The HFG model therefore still requires an FSM for high level state and mission planning. The need for an FSM will be eliminated in Chapter 7, where we introduce the concept of activity. The model is also designed to natively support a graphical user interface for the construction and analysis of computation graphs, which is presented in Chapter 9.

5.1 Unified Data Flow Graph

The HFG flow graph model is derived directly from the the SDF+ model. Let G be an HFG graph, then each node $v_k \in V$ again consists of four different types of ports,

$$\begin{aligned} \mathcal{I}_k &= \left\{ {}^k I_1, \dots, {}^k I_{i_k} \right\}, && (5.1)\\ \mathcal{O}_k &= \left\{ {}^k O_1, \dots, {}^k O_{o_k} \right\}, && (5.2)\\ \mathcal{S}_k &= \left\{ {}^k S_1, \dots, {}^k S_{s_k} \right\}, && (5.3)\\ \mathcal{E}_k &= \left\{ {}^k E_1, \dots, {}^k E_{e_k} \right\}, && (5.4) \end{aligned}$$

which are equal to the SDF+ model. As with SDF+, inputs $\mathcal{I}_k$ and outputs $\mathcal{O}_k$ are expected to be handled *synchronously*, whereas slots $\mathcal{S}_k$ and events $\mathcal{E}_k$ are *asynchronous*. We now generalize the arcs E to allow hybrid connections, which means that a valid edge e between two nodes v_k and v_l is given by

$$e \in (\mathcal{O}_k \cup \mathcal{E}_k) \times (\mathcal{I}_l \cup \mathcal{S}_l) . \tag{5.5}$$

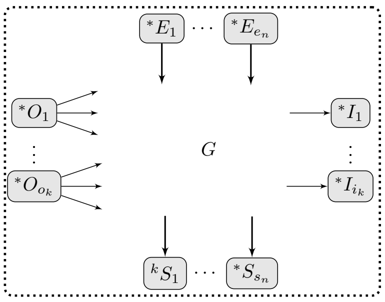
This definitions allows the creation of hybrid connections, connecting outputs to slots and events to inputs. As we will see below, this allows more flexible graph constructions, such as synchronously reacting to an event. Furthermore, no more translation nodes are necessary to transition between asynchronous and synchronous data flow.

We add additional ports for the graph: $\mathcal{I}_*, \mathcal{O}_*, \mathcal{S}_*, \mathcal{E}_*$, which are not associated with any node, but with the graph itself. The graph is then specified by

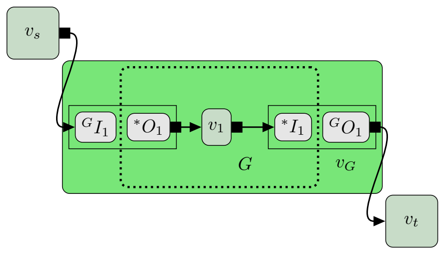
$$G = (V,\ E,\ (\mathcal{I}_* \cup \mathcal{O}_* \cup \mathcal{S}_* \cup \mathcal{E}_*)) = (V, E, P_*), \tag{5.6}$$

where we denote the set of all graph ports with P_* and where E now also contains connections between node ports and graph ports. Figure 5.2a demonstrates these additional ports for a general graph G. These ports can be considered to be external data sources and sinks, which behave exactly the same as the node counterparts. To name a few examples: $\mathcal{O}_*$ can contain outputs that publish externally produced data, $\mathcal{I}_*$ can provide inputs that forward data to the system, $\mathcal{S}_*$ can contain slots to perform actions like terminating the execution of the graph and $\mathcal{E}_*$ can have events triggered by the operating system.

Using these graph ports, G itself can be made to implement the interface of a node v_G: For each graph port we add the complementary port to v_G and relay messages between them. For example, an input ${}^G I_1$ is created for v_G in Figure 5.2b to represent the graph output ${}^* O_1$. Tokens received by ${}^G I_1$ are then forwarded to ${}^* O_1$ and notifications are relayed in reverse. Relay outputs, events and slots are similarly created for the other graph ports. This allows the creation of composite structures, with graphs containing sub-graphs as nodes, as demonstrated in Figure 5.2b.



(a) Each HFG graph G has additional, external ports which can be connected to its nodes.



(b) The graph G itself implements the interface of a node v_G. Here v_G is transparently used in a graph with v_s and v_t.

Figure 5.2: Additional ports allow a sub-graph to act as a node in a higher level graph.

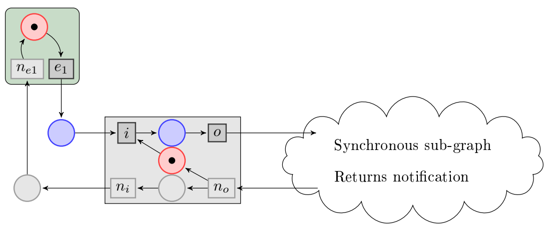
This type of nested graphs could also be achieved using the SDF+ model. The added complexity, however, reduces the similarity to the original definition of SDF and would therefore make the SDF+ model less comparable. Furthermore, the largest benefit of composite graphs can be achieved by using hybrid connections, as we will show in Section 5.4.

5.2 Hybrid Execution Model

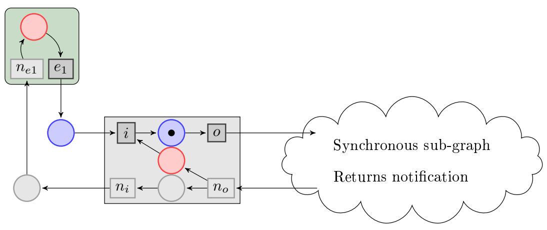
The execution model mostly corresponds to the SDF+ model, using the same Petri net components. We will demonstrate the newly achievable structures using hybrid connections. In addition, we give a model for relaying messages in a sub-graph.

5.2.1 Event to Data Flow

Using HFG it is possible to synchronously process a token generated in an event. This is demonstrated in Figure 5.3a and Figure 5.3b, where an event is connected to a connection which leads to a synchronous sub-graph.



(a) When e_1 is triggered, a token is sent to the connection.



(b) e_1 is disabled, until the token is processed. e_1's node is disabled as well.

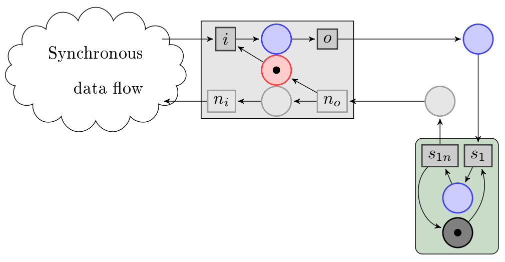
Figure 5.3: A hybrid connection between an event and a synchronous sub-graph.

Once the event is triggered it becomes disabled until a notification returns. If the connection would lead to a slot, this would result in the same behavior as seen in the SDF+ case: A notification would be sent immediately after the token was processed at the slot. With a hybrid connection to a synchronous graph, however, the node sending the event token is disabled until the token is processed. This is because a notification in a synchronous graph is only sent once all sinks have received the token.

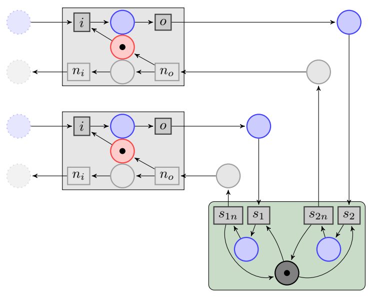
Using this construct, we can now synchronously handle asynchronous events in a sub-graph. With SDF+ this would only be possible, if the sub-graph consisted of only one node, because the pure asynchronous connections immediately return notifications. This allows the creation of pipelines to handle events.

5.2.2 Data Flow to Event

Hybrid connections also allow to connect a synchronous data stream to a slot, as shown in Figure 5.4a. This means that a connection between an output and a slot is used to translate the synchronous data flow to an asynchronous event in each iteration. As Figure 5.4b demonstrates, this can be used to asynchronously handle different, synchronous streams in a single node. A hybrid connection between an output and a slot behaves in the same way as a synchronous connection from an output to a sink node. Each data token is received and a notification token is returned immediately.



(a) A hybrid connection from data flow to a slot is equivalent to a sink in regular synchronous data flow.



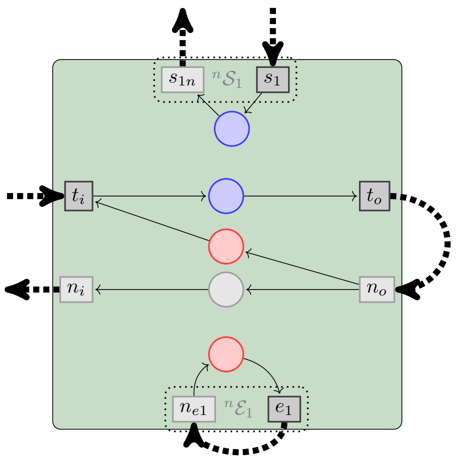
(b) Multiple synchronous data tokens can be handled asynchronously.

Figure 5.4: Hybrid connections allow a node to asynchronously handle incoming tokens from synchronous sources without a need for buffers.

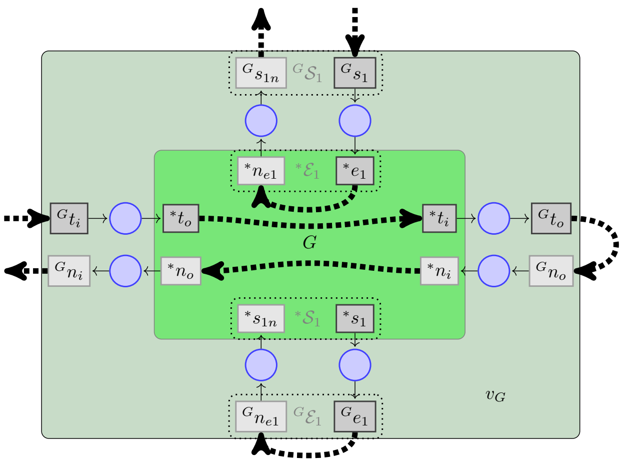
There is therefore no difference in using an asynchronous slot to replace a single input for sink nodes. For general nodes, with one input and one or more outputs, there is a semantic difference: Even though there is only one token received, the slot will immediately return a notification, while an input will only send a notification up-stream, once the node itself is notified. A hybrid connection to a slot can therefore be used to break a synchronous stream, e.g. to allow sub-graphs to run at different frequencies, as shown in Section 5.4.

5.2.3 Sub-Graph Relaying

A sub-graph G can be used as a black-box node v_G, i.e. it behaves exactly like any other node to the parent graph. This puts some constraints on the implementation shown in Figure 5.2: Inputs of v_G still have to be handled synchronously, whereas slots have to be handled asynchronously. We can use the same model for inputs as before, where each input ${}^G\mathcal{I}_k$ of v_G *relays* incoming tokens to the graph-external outputs ${}^*\mathcal{O}_k$ of G. This is demonstrated in Figure 5.5b.



(a) Review of the essential interface transitions of a node. Locking places are not shown.



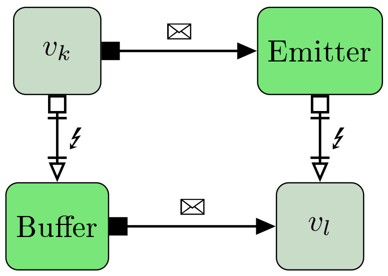
(b) A Petri net showing token relaying to a sub-graph. Locking and enabling places are not shown. The graph G corresponds to Figure 5.2a.

Figure 5.5: Comparison of a node and a nested graph. The only difference is that the slots do not immediately return notifications but relay tokens to G. The dashed lines show the external control flow, when valid connections are constructed.

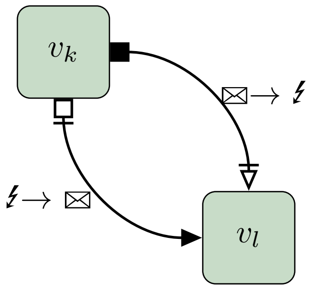
For the external events ${}^*\mathcal{E}_k$ of G we need a slightly different model to correctly implement relaying messages from a slot ${}^G\mathcal{S}_k$ to the graph event ${}^*\mathcal{E}_k$. The difference can be seen by comparing the event in Figure 5.5a with the internal event ${}^*\mathcal{E}_1$ in Figure 5.5b: Just relaying all tokens could lead to a build-up of queued messages, if the source is running at a higher frequency. For this reason, notifications for each slot ${}^G\mathcal{S}_k$ can only be sent, once the relaying event ${}^*\mathcal{E}_k$ is notified, which is achieved with Figure 5.5b. The dashed lines show the flow of individual tokens and the corresponding notifications. The sub-graph has to be constructed such that incoming, synchronous data reaches the output transition.

5.3 Updated Graphical Representation

Since SDF+ graphs cannot contain connections between synchronous and asynchronous ports, the visual notation there only allows horizontal or vertical connections between nodes. In contrast, the hybrid connections of HFG graphs can be directly visualized using arcs that change direction from vertical to horizontal or vice versa. This circumstance is visualized in Figure 5.6, where we compare the two ways to translate between asynchronous and synchronous data streams: In Figure 5.6a with SDF+ and in Figure 5.6b with HFG. Since HFG is a superset of SDF+, however, both are valid HFG graphs.



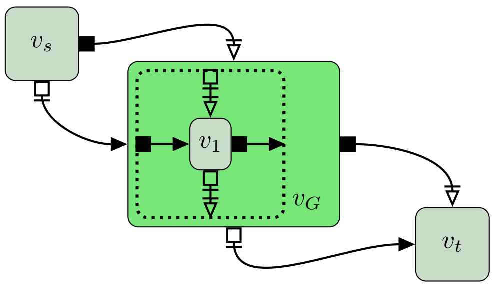
(a) In the SDF+ model, translation requires additional nodes and connections.



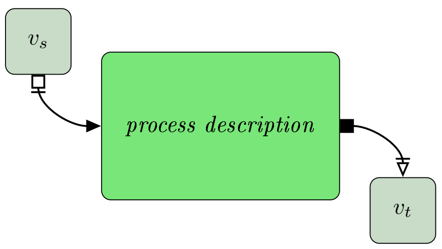
(b) With HFG the translation is directly possible and is therefore less verbose.

Figure 5.6: Translation between asynchronous and synchronous data streams is natively supported by HFG graphs.

In addition to hybrid connections, HFG also introduces nested graphs. Every sub-graph consists of many global and relaying ports, as shown in Figure 5.5, which quickly leads to cluttered visualizations. We therefore also omit these types of ports, when they are not explicitly needed to convey any meaning, resulting in graphs such as shown in Figure 5.7.



(a) To show a nested graph, we also omit graph-global ports, when possible.



(b) A sub-graph is equivalent to a node. The node can be used as a black box by giving a description.

Figure 5.7: In some contexts, nested sub-graphs can be used as a black box.

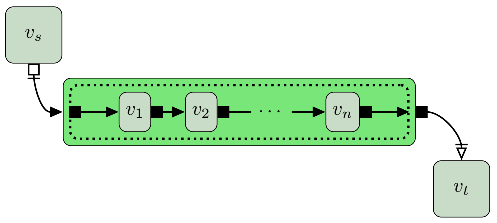
5.4 Exemplary Usage

5.4.1 Synchronous Reaction to Asynchronous Event

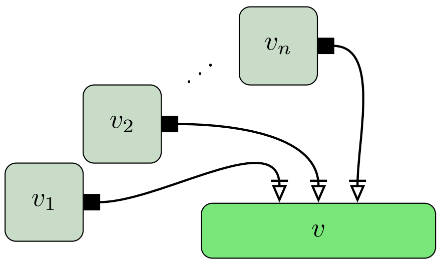
In SDF+ we introduced buffers to translate events to SDF, however avoiding the use of an unbounded buffer is preferable in long lasting data flow graphs. On the other hand, bounded buffers require tokens to be dropped, once the bound is reached. In contrast, the HFG model allows complete interaction between synchronous and asynchronous data streams. This makes it possible to implement a synchronous reaction to an event, guaranteeing that every event is handled. Figure 5.8a shows an example graph, where an event is synchronously handled and then converted back into an event.

5.4.2 Converging Data Steams

Besides a synchronous reaction to an event, where we use hybrid connections from events to inputs, we can also make use of hybrid connections from outputs to slots. This is demonstrated in Figure 5.8b, where a node v has several slots that each receive tokens from possibly different, synchronous data streams. All of these streams can operate at different frequencies, they can even be irregular. The node v handles all received tokens separately, in contrast to the synchronous data flow case, where all inputs have to hold a token for the node to become enabled.



(a) A synchronous sub-graph is used to handle the event generated by v_s.

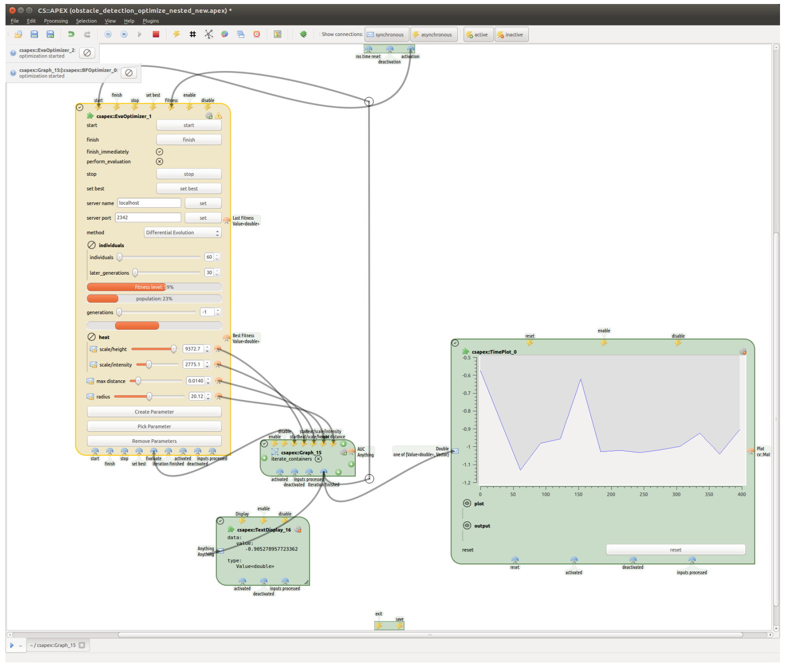


(b) Node v handles different synchronous data streams asynchronously.

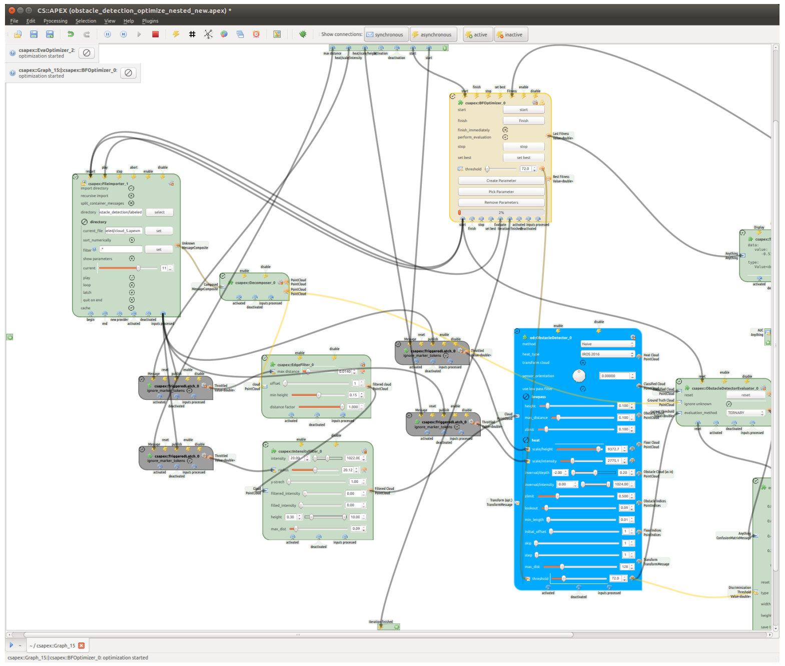
Figure 5.8: HFG allows nodes to directly translate synchronous to asynchronous streams.

5.5 Conclusions

HFG resolves most of the limitations of SDF+, however we still need a separate FSM to model high level state. In Chapter 7 we show how to eliminate of the use of a separate FSM model by introducing activity flow. Figure 5.9 shows a hierarchical HFG that was used in the PATSY project to find optimal parameters for an obstacle detection algorithm (Chapter 6). The figure shows two screen shots from CS::APEX, one of the top-level graph and a second of the sub-graph that is optimized.



(a) A generic optimization node (yellow) generates parameter values using Differential Evolution (DE). These are forwarded into the central sub-graph (Figure 5.9b), which evaluates them and generates a fitness value. The fitness is plotted over time.



(b) The sub-graph used in Figure 5.9a. The parameters received on the graph slots on top are buffered for further use (dark gray). An obstacle detection algorithm (blue) is evaluated on multiple test images. Multiple thresholds are tested in sequence by a nested grid-search optimizer (yellow). The fitness of the parameter set is returned after the internal optimization (graph event on the bottom).

Figure 5.9: A hierarchical HFG that uses two levels of optimization to find a good parameter set for an obstacle detection algorithm. Exact details of this algorithm are developed and evaluated in Chapter 6

.

Chapter 6

Modeling a Person-detecting Autonomous Guided Vehicle

6.1 Container Transport in Dynamic Environments

Automated guided vehicles (AGVs) are mainly used for the automated transport of heavy payloads in large-scale environments, such as hospitals (Fung *et al.*, 2003), (Ozkil *et al.*, 2009), (Evans *et al.*, 1989), warehouses (Ronzoni *et al.*, 2011), (Martinez-Barbera and Herrero-Perez, 2010) or factories (Cardarelli *et al.*, 2014). A typical mission of a single AGV may consist of navigating to a central station, picking up a payload package and then delivering it to a target station. Dedicated sending stations are used to prepare containers for transport, where the pose of the containers is known. This allows AGVs to pick up containers without explicitly detecting their pose.

An AGV is traditionally guided by markings on the factory floor or precisely mapped reflective markers throughout the workspace. It can follow any predetermined path via a network of routes, which have to be kept devoid of obstacles. Externally guided vehicles can be localized very precisely, however they can only be employed in controlled environments. The landmarks used for localization have to be clearly visible, which may not be the case in more dynamic environments like hospitals. To implement an AGV system in such environments, it is beneficial to use a probabilistic localization approach, such as Monte Carlo Localization. These systems can easily cope with dynamic environments by allowing the localization certainty to decline in some situations. The downside of a less certain pose estimate is that precise, absolute positioning might not always be possible. This does not pose a problem for navigation, however the acquisition of payload containers can then become infeasible.

To fulfil industrial safety standards, AGVs have to be equipped with certified sensors that can immediately halt the vehicle in case of a safety hazard. These sensors are usually 2D laser scanners that monitor a virtual safety field around the vehicle. Better safety can be achieved using infrastructure sensors (Sabattini *et al.*, 2015). The vehicles cannot, however, be autonomously used in areas inexperienced people have access to, without considering further safety measures. The aim of our research is to expand the domain of applicability of AGVs to these areas.

Contrary to controlled conditions in closed-off areas, AGVs have to be able to detect small obstacles below the plane of measurement of the laser scanners with on-board sensors. Obstacles above that plane and people have to be detected and treated properly as well. Cameras can be used to detect such obstacles in 3D (Wang *et al.*, 2001), but they depend on proper lighting conditions. Additional information can be gained by using active sensors such as actuated 2D LIDAR systems and 3D laser scanners. There exist laser scanners that can survey not only a 2D plane, but a 3D volume. These scanners can be used to detect obstacles that are beneath or above the plane of the safety fields. They function similarly as 2D scanners by having a moving mirror that sweeps a laser beam across the environment. These moving parts are the biggest downside of 3D laser scanners, because they result in large and heavy sensors.

3D time-of-flight (TOF) cameras are a more recent active sensing technology: Just as 3D laser scanners, they probe a volume in front of the sensors and can therefore be used for detecting obstacles which two dimensional sensors cannot see (Alenyà *et al.*, 2014). Contrary to 3D laser scanners, however, they do not utilize moving parts and can therefore be built smaller and more robustly. They send out modulated light and capture the reflection of this light from the environment. This allows TOF cameras to have a comparatively high resolution and information density, which makes them very interesting for 3D scene understanding. However, TOF cameras suffer from various types of measurement errors that cause data artifacts (Frank *et al.*, 2009) which have to be handled.

In this chapter we present a complete model of an AGV, largely based on *Hybrid Flow Graphs*, that addresses the following main contributions:

1. We detect arbitrary obstacles using TOF cameras that are operated close to the ground on an AGV with limited resources, using a simple and efficient classification process. We handle different types of data errors caused by the TOF measurement principle. Our approach does not assume that the floor is planar and can detect obstacles that are smaller than the average sensor noise by using depth and intensity information. The algorithm is applicable to any TOF camera system and has also been used with other depth sensors, e.g. RGB-D and stereo cameras.

2. The 3D information extracted by the obstacle detection is used to detect the presence of people in front of the AGV.

3. The pose of the payload is detected in 2D laser range scans based on an exhaustive search. The 3D pose is then recovered and the existence of the payload is verified using the detected 3D obstacles as well.

4. A reactive controller is used to steer the AGV beneath the payload containers.

5. We use a variant of Adaptive Monte Carlo Localization (AMCL) that is based on architectural maps in vector form.

This chapter is based on previously published articles (Buck *et al.*, 2016a, 2017), and (Hanten, Buck, Otte, and Zell, 2016), as well as a currently submitted article (Hanten, Kuhlmann, Buck, Otte, and Zell, 2018).

6.2 Related Work

6.2.1 Obstacle Detection for AGVs

Bostelman *et al.* (2006) present an approach in which they detect obstacles using both intensity and depth information from a TOF camera. They treat all points with high intensity values as obstacles, if they are high enough above the floor. Compared to our approach, they assume a perfectly calibrated camera and a flat floor surface, to which they need to fit a plane model. They verify their approach using a set of vertical obstacles that vary in width but have a minimum height large enough for a planar laser to detect them. Experiments were performed 0.8 m above the floor, where the effects we discuss in Section 6.4.2 are negligible. Another TOF-based approach by Wang *et al.* (2014) is based on clustering. Their approach also explicitly handles wrapping artifacts and uses intensity values to estimate the noise. We cannot compare the algorithms easily since the authors only provide qualitative detection results.

There are many similar approaches for obstacle detection, most of which are using actuated LIDAR sensors to generate point clouds. All of them have in common that they do not consider measurement artifacts, since LIDAR sensors are not as prone to measurement errors as TOF cameras. Furthermore, all these algorithms are either meant for use with robots that have a high ground clearance or the authors do not mention how the algorithms perform for small objects. Schafer *et al.* (2008) detect obstacles based on LIDAR measurements. They use a vertically spinning laser scanner, and iterate columns of data to calculate the slope of the environment, ignoring intensity information. However, their approach requires the data to be sorted by distance before the iteration and cannot cope with the presented TOF artifacts. Morton and Olson (2011) present a classifier that can detect both positive and negative obstacles using 3D LIDAR. In their approach they discretize the point cloud into a grid and detect obstacles based on the comparison of cells with a size of 15 cm. This approach works very well for actuated laser scanners, however the runtime is 200 ms, which means it cannot be used to interpret all the data of a TOF camera.

Nüchter *et al.* (2006) detect drivable surfaces in outdoor scenarios using high resolution point clouds recorded by a tilting laser scanner. They calculate the slope at every point in the cloud using a fixed index offset to determine the preceding and succeeding points. Similarly, the approach by Zhu *et al.* (2010) also uses a fixed step size to calculate features for each point in the point cloud that are then classified using a support vector machine. Compared to our local search, these fixed step approaches cannot handle large depth discontinuities very well and we found that a fixed offset causes many erroneous classifications when used indoors. They do not use the intensity of the measurements and they do not handle measurement artifacts.

A well-known approach to detect obstacles in autonomous cars is the stixel-world approach by Badino *et al.* (2009). Stixels represent vertical slices in a stereo depth image. Compared to our approach, these vertical slices are more coarse and optimized to detect larger obstacles. This algorithm is tailored to be used outdoors in cars and is therefore not suitable for detecting small obstacles indoors. Broggi *et al.* (2013) present another stereo camera based approach for outdoor use, which is real-time capable with a frequency of ca. 10 Hz. The approach is not able to detect obstacles smaller than 25 cm due to the use of a voxel grid approximation.

RGB-D and fusion based approaches are also common. Stimming *et al.* (2015) detect obstacles based on the fusion of stereo vision and LIDAR scanners. In their scenario, they can position the 3D sensor high up, which causes fewer artifacts and they rely on their 2D scanners for detecting small objects. Lee *et al.* (2012) use RGB and depth information, however, they do not aim to detect very small objects.

6.2.2 Payload Detection for AGVs

AGVs are commonly used to automate the transport of goods in large environments. Typical scenarios, as surveyed in (Niechwiadowicz and Khan, 2008), (Karabegović *et al.*, 2015), consider a manually designed system in which human designers are laying out all the paths that the vehicles can travel on. This approach requires a very precise positioning system or external markings that allow the vehicles to accurately follow their course. Payload containers are usually not explicitly detected, but have to be positioned into sending and receiving stations by human operators. These stations are built in a way that the wheels of the containers are automatically positioned with their wheels in parallel, such that the robots can easily drive beneath them.

Object detection from 3D data is well studied in literature. Liebelt and Schmid (2010) employ a detection pipeline that detects parts of an object and then fits a 3D model. Other approaches, such as (Rusu *et al.*, 2008) and (Lai *et al.*, 2012), require many resources and do not achieve real-time performance.

Varga and Nedevschi (2014) describe an architecture using stereo vision to detect payloads in the form of euro pallets. The approach employs a scrolling window search on a restricted sub-image and requires several hundred milliseconds of run-time. The authors therefore propose to position the AGVs at a distance of approximately 2.5 m in front of the payload to calculate a trajectory. In contrast, our approach detects container carts and relies on direct distance measurements from 2D and 3D sensors and can run at the speed of data acquisition, which allows us to detect the pose of the payload continuously.

Weichert *et al.* (2013) propose a system that can detect payloads on euro pallets using 3D TOF cameras. Their system is capable of detecting individual packages on a pallet using a multiple stage model fitting algorithm that aligns simple box models with the observed data. In contrast, our approach uses a similar box model to detect the pose of the payload container itself. Thamer *et al.* (2013) similarly detect the components of the payload using 3D information by employing a segmentation and surface fitting pipeline. The detection of freely positioned container carts is so far not discussed in the literature.

6.2.3 Person Detection

Person or pedestrian detection is extensively studied in literature. Many common approaches are based on image processing and combine feature extraction with machine learning techniques. The arguably most popular feature, described by Dalal and Triggs (2005), is the *Histogram of Oriented Gradients* (HOG) descriptor. Other approaches, such as Haar-like features (Viola and Jones, 2001), ACF (Dollár *et al.*, 2014) and FPDW (Piotr Dollár *et al.*, 2010) are proposed as faster alternatives.

Another common class of algorithms is based on fusing visual with metric sensor information, such as approaches to detect people in RGB-D data (Salas and Tomasi, 2011). Some of these approaches extend HOG to RGB-D data (Luber *et al.*, 2011; Spinello and Arras, 2011; Wu *et al.*, 2011), others use stereo cameras (Enzweiler *et al.*, 2010). Many approaches require the removal of the ground plane, which is not feasible in our case. Other approaches are very computationally expensive and require a GPU (Jafari *et al.*, 2014), which also includes approaches based on deep neural networks, such as the algorithms proposed by Sermanet *et al.* (2013) and Toshev and Szegedy (2014).

For our scenario, we require an approach with reduced computational complexity that nonetheless achieves sufficiently good results. We use a feature-based approach, using the generic obstacle detection as a proposal generator.

6.2.4 Vector-based Adaptive Monte Carlo Localization

Navigation is arguably the most important task of an AGV. To enable successful navigation, a robust localization method has to be available. This is commonly achieved using external landmarks that are introduced into the workspace. In our case, however, we aim to develop a freely navigating AGV that can also operate in areas that are populated by people. For this reason we use Monte Carlo Localization based on prior knowledge, in our case using architectural maps of the building.

Using vector-based maps of the environment is a natural approach to localization in known environments. We make use of pure localization algorithms, although there are SLAM approaches using vector representations (Garulli *et al.*, 2005; Sohn and Kim, 2009; Elseberg *et al.*, 2010). Existing localization algorithms usually make use of either manually created maps or manually translated architectural plans (Cox, 1991; Sohn and Kim, 2008; Luo *et al.*, 2008).

Many existing approaches are not directly applicable, because the do not model uncertainty, i.e. they do not implement Bayesian filters (Thrun *et al.*, 2005). Instead we extend an existing MCL implementation based on Fox *et al.* (1999) to directly work with unaltered architectural plans of the environment.

6.3 Overview

6.3.1 Requirements

Figure 6.1 again shows the AGV developed for the PATSY project, which is designed to transport containers by driving under them and lifting them up.

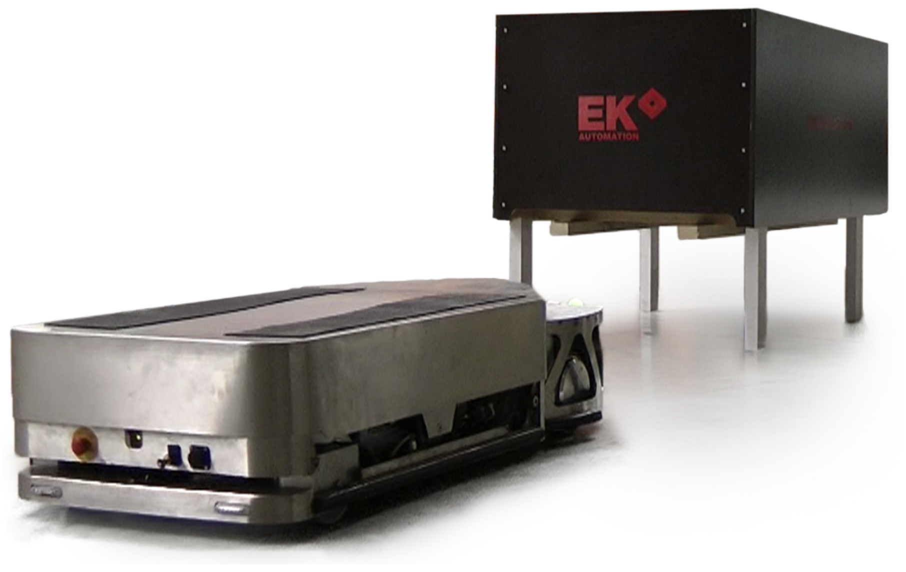


Figure 6.1: The AGV has to be low enough to be able to drive under containers to lift them up. A TOF camera for 3D obstacle detection in the front of the AGV is mounted close to the floor, at a height of 10 cm and an inclination of 15°.

We can state the following requirements:

- Since the AGV has a small ground clearance, obstacles range from a few centimeters to several meters in size.
- People have to be recognized for additional safety.
- Payload containers have to be reliably collected, without the use of sending or receiving stations.
- The AGV has to localize itself in a known environment.
- Due to the low height of the AGV, all sensors have to operate close to the ground.
- All algorithms have to run in real-time on the on-board computer.

In the following we present the final system architecture that implements these requirements. The HFG model is used to implement every perception component.

6.3.2 System Architecture

To fulfil these requirements, we use a collection of different modules that together form a complete robotic architecture, which is visualized in Figure 6.2. Most of the modules are implemented using the HFG model, except for low-level control modules that are communicating with the hardware. The navigation and localization functionality are also not implemented using data flow models, because they are based on pre-existing functionality.

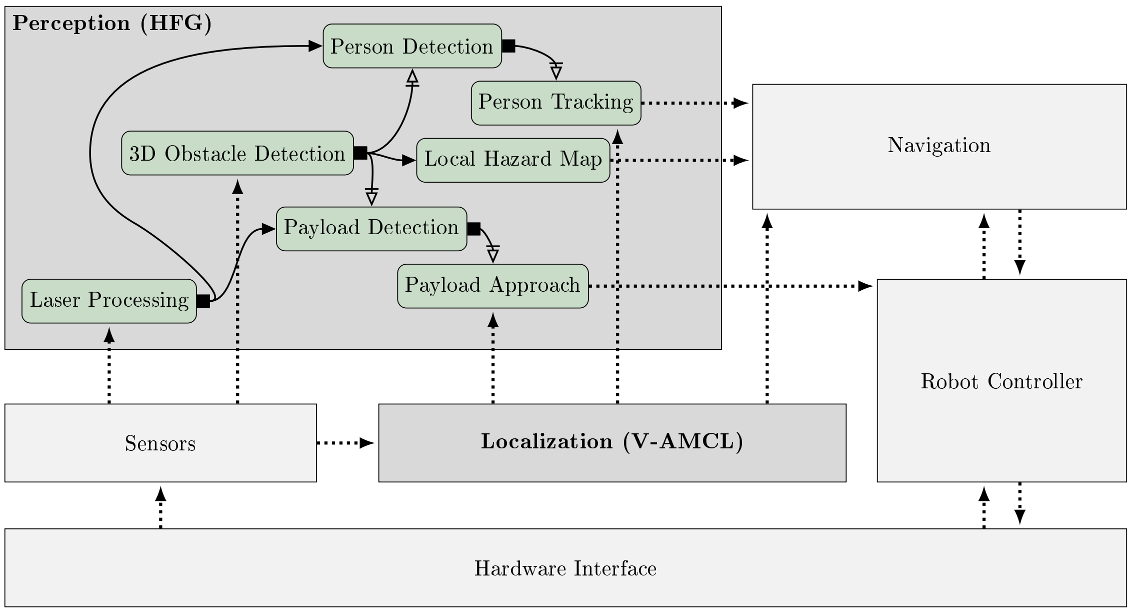


Figure 6.2: The data flow in the complete PATSY software architecture. HFG components and their interface are shown.

The basic component is the detection of obstacles from 3D TOF data, which is necessary for safe navigation and is also used as a pre-processing step for person- and container detection. Furthermore, we implement a person tracking component that maintains hypotheses for multiple people to smooth the detection results and to mitigate false or missing detections. Another node is implemented to approach a payload container, which is only used when the payload has to be collected.

In this chapter we describe the individual modules, mainly focusing on the data flow components. In Chapter 8 we then show how to combine perception and high level control in a common framework using activity flow graphs.

6.3.3 Point Cloud Structure

Depth sensors generate depth images, such as the one shown in Figure 6.3a. This image can be projected into a full, three dimensional point cloud (cf. Figure 6.3b) using the camera model for the sensor.

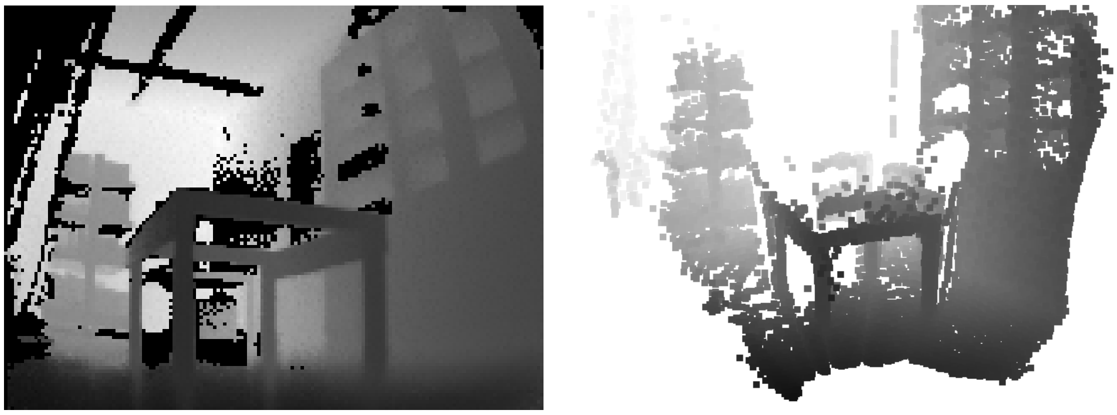

(a) Depth image of a container mock-up. (b) Point cloud of a container mock-up.

Figure 6.3: The point cloud is structured the same as its underlying depth image.

It is computationally expensive to work with unordered point clouds. Instead we assume that every point cloud is structured, i.e. that it was generated from a depth image. With this assumption it is possible to infer a pixel in the depth image for each point in the cloud.

6.3.4 Obstacle Types

Let $PC = \{p_i \mid 0 \leq i \leq n\}$ be a point cloud consisting of n points p_i that represent the environment. The goal of obstacle detection is to decide for each p_i, whether it belongs to an object that is an obstacle to the AGV. We differentiate the following types of obstacles:

1. *Objects* are all the p_i that belong to an obstacle that can also be detected by the 2D scanners. Objects are therefore large and relatively easy to detect.

2. *Overhanging* obstacles are those p_i that lie above the 2D plane.

3. *Small* obstacles are points so low that they completely lie below the laser scanner plane. In our case this means that they are lower than 6 cm. They are especially hard to detect by depth alone due to noise in the depth measurements.

4. *Negative* obstacles are hazards to the AGV that are below the floor level, such as descending stairs or large holes in the floor.

The obstacle type is only relevant for the following discussion. During the operation of the AGV, however, collisions have to be avoided with every obstacle type equally.

6.4 Time-of-Flight Cameras

Different types of 3D sensors have been evaluated for the presented scenario by Rauscher *et al.* (2014). The E70P TOF camera by Fotonic has then been selected as the 3D sensor for our AGV, since it is smaller than 3D laser scanners and can handle stray sunlight better than RGB-D cameras. Stereo cameras have not been considered, since indoor scenarios often entail low textured floors and walls, which cannot reliably be used for disparity estimation. Additionally, TOF cameras are active sensors, which means they also work in bad lighting conditions and even in darkness, where stereo cameras will fail.

6.4.1 Characteristics of the Fotonic E70-P Camera

The E70P sensor (cf. Figure 6.4) has a minimum range of 0.15 m and can measure the floor at a minimum of 0.65 m due to its field of view and the way it is mounted in our case. This is an important factor for safety, because there is always some delay until an obstacle is recognized and until the brakes can halt the vehicle. When an obstacle is detected directly in front of the sensor, this delay can cause the AGV to approach the obstacle so closely, that it cannot be seen any more. For objects this is not a problem, since they can be seen by the 2D sensors. Small and overhanging obstacles, as well as negative obstacles, on the other hand, have to be remembered such that collisions can be avoided. This is the motivation for obstacle mapping, which we will only shortly describe in this chapter.

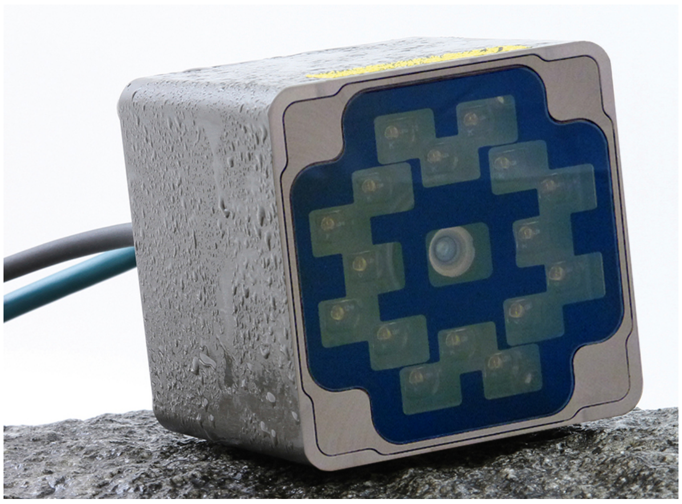
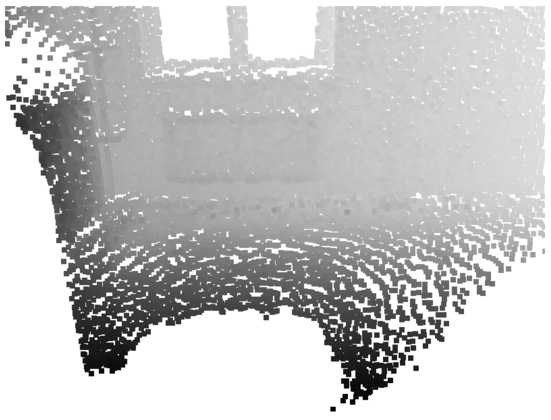


Figure 6.4: The Fotonic E70P sensor. (Source: http://www.fotonic.com/product/fotonic-e-serie-4w/)

The maximum range of the E70P sensor is ca. 10 m in perfect conditions. Due to the low operation height, the angle at which the emitted light hits the floor in front of the vehicle is very sharp. At greater distances, too much of the emitted light is reflected by the floor, which then cannot be detected any more. Hence the effective distance at which the floor can reliably be measured is only about 2 m.

In the vicinity of walls or large obstacles, the effective range is increased since more light is reflected off the walls back to the sensor, which can be seen in Figure 6.5. Large obstacles are therefore easier to detect, which is demonstrated in Figure 6.5a. When facing small or negative obstacles in an otherwise free environment, however, the amount of missing information causes problems. When confronted with a situation such as depicted in Figure 6.5b, where the floor cannot be seen further than ca. 1.5 m, it is difficult to decide whether there is an obstacle ahead.


(a) Facing a wall, the floor is detected up to ca. 7 m.

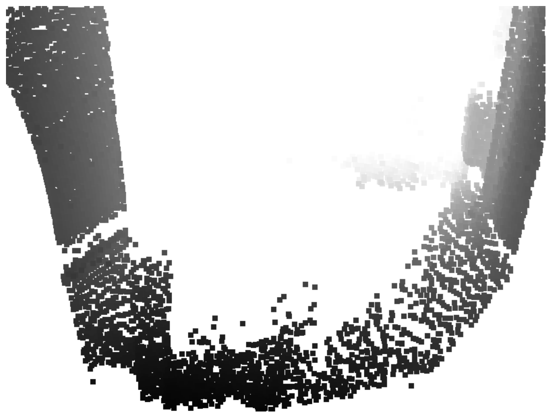
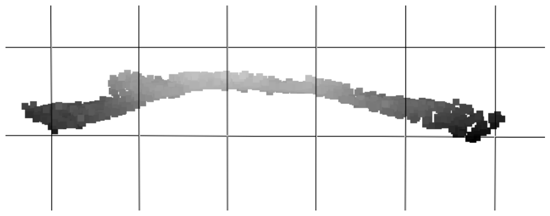

(b) Facing a long corridor, the floor is detected only up to 2.5 m.

Figure 6.5: The range at which the floor is still detected varies depending on the environment. If the sensor is facing a large obstacle, more light is reflected and the effective range is higher. The range at which the floor is still detected varies.

The last challenge is the signal to noise ratio of the sensor. Especially with stray sunlight, the measurement noise is quite high, as can be seen in Rauscher *et al.* (2014). Even with relatively small disturbances, there is more noise to deal with than with laser scanners, for example. In addition to the sensor noise, there are other artifacts in the data that have to be dealt with, before obstacles can reliably be detected.

6.4.2 TOF-Camera Sensor Errors

A major issue with many TOF cameras is a static distortion effect, which is shown in Figure 6.6. Flat surfaces near the sensor are disturbed so severely, that an otherwise flat floor can no longer be measured as a plane. The amount of distortion depends on many factors, such as the distance to the sensor and the distance to the nearest object in the direction of the emitted light. Some of these effects can be corrected by calibration, as shown by May *et al.* (2009), but depending on the environment there are still distortions remaining. Many perception algorithms in the literature, e.g. (Bhatia and Chalup, 2013), assume a flat ground and perform some form of plane fitting. Since the floor is not measured as a plane, such algorithms are not possible in our case.


(a) Side view of the distorted ground plane.

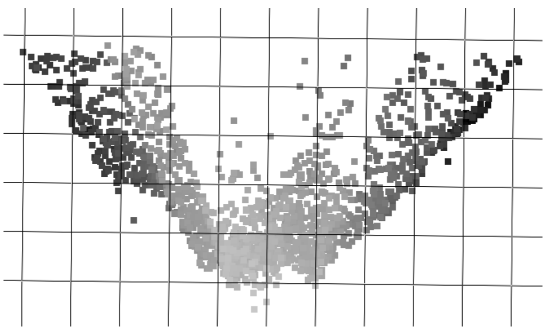
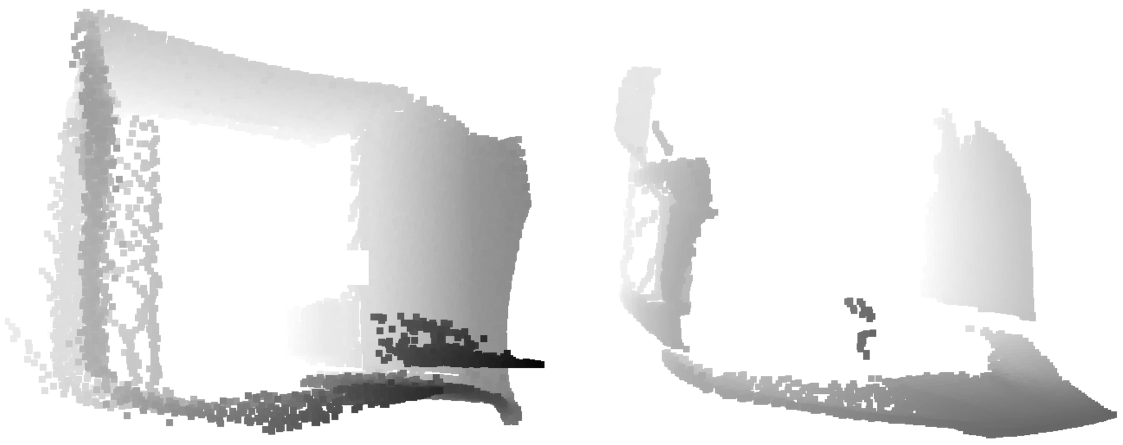

(b) Top down view of the distorted ground plane.

Figure 6.6: Various effects accumulate and cause a distorted image.

The second type of artifacts are interpolation errors on edges of objects. As explained by Reynolds *et al.* (2011), these are caused by large distance discontinuities, when light is reflected both by the foreground object and the more distant background. These errors result in flying pixels that lie between the foreground and the background. Since these measurements do not correspond to any real-world objects, they must neither be classified as obstacle nor as traversable.

The last class of artifacts are so-called wrapping errors which are directly caused by the measurement procedure: A TOF camera emits infrared light and calculates the distance for each pixel using the time it takes for the reflected light to be received again. The emitted light is the carrier of a modulated signal wave that has a wave-length λ of several meters. The round-trip duration is calculated from the phase-shift between the emitted and the received signal. Since the modulated wave is periodic, the determined phase-shift for a measurement z is the same as for $z + i\lambda$ for any integer i. This results in erroneous measurements for objects that are further away than $d_{max} = \frac{\lambda}{2}$, where the distance wraps around and is reported in the ambiguity interval $[0, d_{max}]$ which results in flying pixels.

Normally, there is not enough light received from objects that are further away from the sensor than d_{max}. However, if the camera is oriented towards a large surface that is



(a) The dark points above the floor are wrapping errors caused by a wall that is narrowly out of range. There is no obstacle.

(b) Artifacts caused by a retro reflector that is positioned a little more than three times d_{max} away. The scene is otherwise empty.

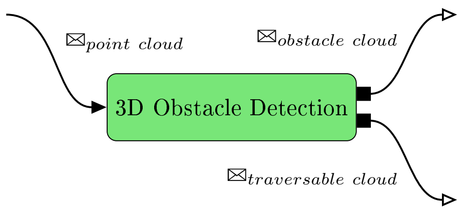
Figure 6.7: Wrapping errors caused by strongly reflecting surfaces.

just outside the ambiguity interval, there can be enough reflected light to cause wrapping errors, as can be seen in Figure 6.7a. Highly reflective surfaces are even worse. Looking directly at glass or mirrors, most of the emitted light can be collected at the sensor, even at large distances.

Retroreflecting materials are reflecting incoming light back to the source, independent of the angle of the incident light. These materials therefore cause wrapping errors at very large distances, many multiples of the wave-length λ, as can be seen in Figure 6.7b. Retro reflectors are common in a variety of settings, because they are required for safety reasons, such as for reflective vests and bicycle wheel reflectors. They are also used as landmarks for some types of AGVs and can therefore not be neglected during obstacle detection.

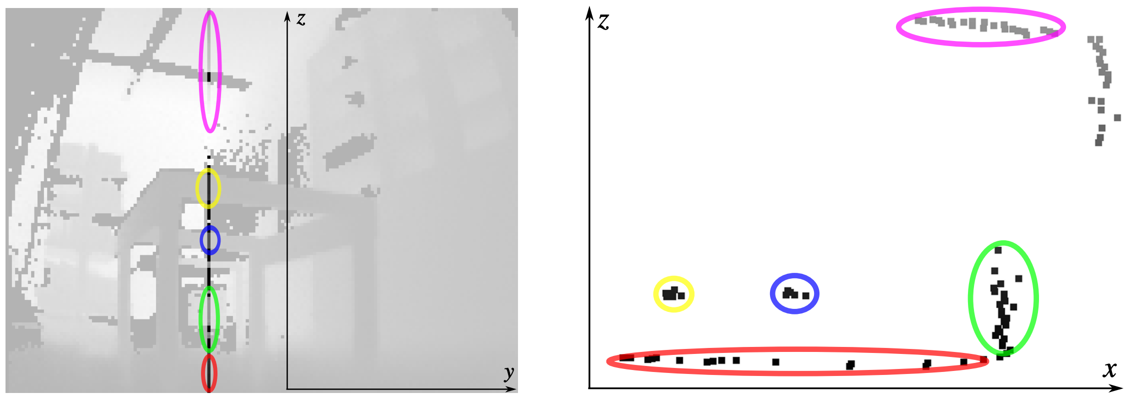
Wrapping errors can be eliminated by using different frequencies for the modulated signal. One can send multiple waves with different frequencies and compare the resulting distances. The measurement is valid if all measurements report the same distance, otherwise there has to be at least one wrapping error. However, a generic obstacle detection approach has to be able to cope with wrapping artifacts since there are TOF cameras, such as the Fotonic E70P, that do not implement multi frequency measurement.

6.5 Generic 3D Obstacle Detection



Our goal is to decide for each point in a given point cloud whether it is an obstacle to the AGV. The *3D Obstacle Detection* module has a simple interface: It receives a point cloud, given in the robot's base coordinate frame. Every point in this cloud is then analyzed, before the node publishes two point clouds of its own: The first cloud contains all obstacle points, the second all traversable points. We require the point cloud to be structured like a depth-image, which is a reasonable assumption since TOF cameras generally output depth and intensity images from which the point clouds are then generated. We work directly with the point clouds, since this way we do not need to know the intrinsic camera parameters. The Fotonic E70P sensor can directly output point clouds with a built-in camera model.

We assume that the TOF camera is mounted horizontally, such that a column in the depth image corresponds to a line of points leading away from the AGV in viewing direction (see Figure 6.8.) We then detect obstacles by iterating each column and searching for different features, such as distance discontinuities.



(a) One column of the depth image is inspected. (b) Side view of the inspected column.

Figure 6.8: One column in the depth image is highlighted. A column in the depth image corresponds to a line of points along the viewing direction of the TOF camera in the metric point cloud. The column is a slice through a container mock-up, with the colors representing corresponding structures.

The input point cloud is filtered before it is analyzed in order to remove as many data artifacts as possible. This step is highly sensor specific, due to the specificity of the artifacts themselves. The point cloud analysis, however, is completely generic and has also been successfully used with many other types of depth sensors.

Figure 6.9 shows an outline of our algorithm. Artifacts, as described in Section 6.4.2, are first removed in a pre-processing step using different heuristics. The remaining points in each column are then classified using depth and intensity information. After the point cloud is classified, detected obstacles and free space are inserted into a local hazard map both to filter false detections and to serve as a memory of obstacles out of view. This map is also used to detect negative information.

```
Input: Point cloud $PC$, Number of rows $R$, Number of columns $C$
Input: Local obstacle map $M$
Output: Updated local obstacle map $M'$
 1: function ObstacleDetection($PC, R, C, M$)
 2:     $D \leftarrow$ Preprocess($PC$)                              ▷ Section 6.5.1
 3:     for $col \leftarrow 0$ to $C$ do                            ▷ Iterate columns
 4:         for $row \leftarrow 0$ to $R$ do                ▷ Iterate current column
 5:             $label(L_{row,col}) \leftarrow$ Classify($D, col, row$)    ▷ Section 6.5.2
 6:         end for
 7:     end for
 8:     $M' \leftarrow$ UpdateMap($M, D$)                           ▷ Section 6.5.3
 9:     FindNegative($M, D$)                                       ▷ Section 6.5.3
10:     return $M'$
11: end function
```

Figure 6.9: Pseudo code of the proposed obstacle detection algorithm. We calculate for each point p_i of the point cloud PC whether it is an obstacle.

6.5.1 Data Representation and Pre-Processing

Camera images, and therefore structured point clouds, are normally represented in row-major form. Since we are iterating the columns, we first convert the point cloud to column-major representation to get an optimized memory layout that results in better cache line sharing. At the same time we transform every point p_i into robot coordinates, such that the x-axis is pointing forward and z-axis upward.

Next we use different heuristics to remove as many data artifacts as possible:

1. We remove a point p_i, if it is closer than the minimum range of the sensor d_{min}, which makes it an obvious wrapping artifact. We also recursively remove direct neighbors p_j of p_i with $||p_i - p_j|| < \delta_{edge}$, which gets rid of artifacts as shown in Figure 6.7a. The threshold δ_{edge} represents the largest allowed distance for the recursion and will be used again for filtering interpolation artifacts in a later step.

   For the Fotonic E70P we have experimentally seen that every wrapping artifact, which is not caused by highly reflective surfaces, falls into this category. This is because non-reflective wrapping errors only occur very close to the ambiguity interval, which wraps around below the minimum range of the sensor.

2. To the remaining points we apply a low-pass filter in the z dimension to reduce sensor noise.

3. We split the volume in front of the AGV in two parts, namely *unknown* with $z \in (-\infty, z_0]$ and *overhanging* $z \in (z_0, \infty)$. Points in the *overhanging* group are either obstacles or artifacts.

   The *unknown* points are either floor points, small obstacles or objects and have to be further classified. Since we cannot assume the floor to be planar, the threshold z_0 has to be set so that all of the points corresponding to the floor and to small objects are in the *unknown* volume. We use an experimentally determined value of $z_0 = 0.12\,\text{m}$.

4. We apply an edge filter to the *overhanging* volume, which removes a point p_i if $||p_i - p_j|| > \delta_{edge}$ for every neighboring point p_j. This eliminates interpolation artifacts at object edges. Edge artifacts are not as common in the *unknown* volume, since there are not many large distance discontinuities so close to the floor. By not filtering the *unknown* points, we avoid further reducing the effective range of the sensor. We experimentally found good results for $\delta_{edge} = 7\,\text{cm}$.

5. As described above, the floor can only be seen up to a fraction of the maximum range d_{max}. Therefore there cannot exist any wrapping artifacts on the floor. This is why we further assume that all remaining wrapping errors emerge in the *overhanging* volume only.

   A more aggressive filter heuristic is only applied to those points to remove retro-reflective wrapping artifacts: We remove a point p_i in the *overhanging* volume, if its intensity $\mathbf{I}(p_i)$ is above a threshold $\delta_{\mathbf{I}}$. We also recursively remove direct neighbors p_j of p_i with $||p_i - p_j|| < \delta_{edge}$, which gets rid of artifacts as shown in Figure 6.7b. Experimentally, we determined $\delta_{\mathbf{I}} = \mathbf{I}_{max} - 3$ where $\mathbf{I}_{max} = 1024$ is the maximally possible intensity value of the sensor.

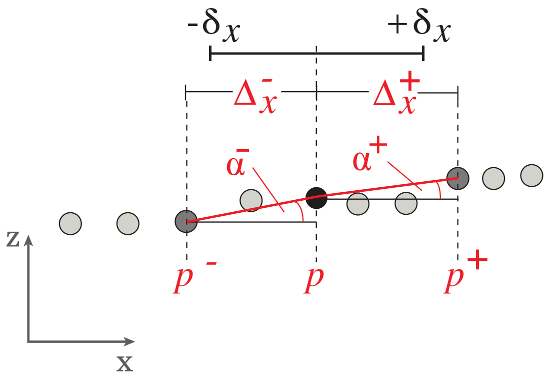
   Importantly, this filter is not applied to the *unknown* volume such that small obstacles cannot be overseen even if they reflect at maximal intensity.

6.5.2 Point Classification

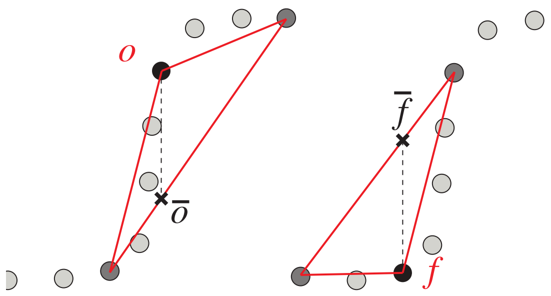
After the pre-processing step, we assume that all artifacts have been eliminated and only valid points are remaining. As a next step, we estimate the probability $P_i := p\,(\text{obstacle}(i) \mid p_i)$ that the i-th point belongs to an obstacle as

$$P_i = \min\left(1.0, \quad \alpha h_D(p_i) + \beta h_I(p_i)\right), \tag{6.1}$$

where $h_D(p_i)$ and $h_I(p_i)$ are functions that calculate the *hazardousness* of p_i, and α, β are used to weight both components. Here $h_D(p_i)$ calculates a score based on depth measurements and $h_I(p_i)$ based on intensity values and both are in the range $[0,1]$.



(a) Predecessor p^- and successor p^+ of p are found with distances Δ_x^- and Δ_x^+ of at least δ_x.



(b) For o we approximate the curvature. For f we use the minimum of the slopes.

Figure 6.10: Neighbors for hazardousness calculation are determined by a local search.

Given a point p_i, we first search for a preceding point p_i^- and a successor point p_i^+ in the same column as sketched in Figure 6.10. Starting at index i, we first search backwards until we find a point $p_i^- := p_j$ with $j < i$ such that the distance in x direction $\Delta_x^- = |p_{i,x} - p_{j,x}|$ is larger than a threshold δ_x. Similarly we search the first successor point p_i^+ with a distance larger than δ_x. The searches are limited to 16 steps which we found to work well in practice. Having found both p_i^- and p_i^+, we calculate the probability P_i, otherwise we classify the point as unknown. Both scores are calculated using p_i, p_i^- and p_i^+.

We define the depth based hazardousness $h_D(p_i)$ using the points seen in Figure 6.10. First, let $\bar{p}$ be the mean of p^+ and p^- and let the z coordinate of p be denoted by p_z. If p_z is larger than $\bar{p}_z$, we approximately calculate the curvature at point p as the difference between the slope towards p^+ and the slope towards p^-. Otherwise, if p_z is lower than the mean, we use the smaller of the two slope values. This second case is defined to better classify floor points directly beside an obstacle. This differentiation prevents obstacles to be grown in relation to the search distance δ_x.

We introduce two parameters that control this function. First we use a threshold

$\varepsilon_z = 1\,\text{cm}$ for the height difference, below which the hazardousness score is 0. For the remaining points we define a maximum curvature κ_{max} above which the hazardousness is 1.0. More formally, we define

$$g_D(p) = \begin{cases} 0.0 & \text{if } \Delta_z < \varepsilon_z, \\ |\alpha^- - \alpha^+| & \text{if } \Delta_z \geq \varepsilon_z \text{ and } p_z \geq \bar{p}_z, , \\ \min(|\alpha^-|, |\alpha^+|) & \text{if } \Delta_z \geq \varepsilon_z \text{ and } p_z < \bar{p}_z \end{cases} \tag{6.2}$$

$$h_D(p) = \min\left(1.0, \kappa_{max}^{-1} \cdot g_D(p)\right) \tag{6.3}$$

with the auxiliary variables

$$\bar{p} = \frac{1}{2}(p^+ + p^-), \tag{6.4}$$

$$\Delta_c^* = |p_c^* - p_c|, \tag{6.5}$$

$$\Delta_c = \Delta_c^- + \Delta_c^+, \tag{6.6}$$

$$\alpha^* = \tan^{-1}\left(\frac{\Delta_z^*}{\Delta_x^*}\right), \tag{6.7}$$

where $c \in \{x, z\}$ and $* \in \{+, -\}$.

For the intensity based hazardousness $h_I(p_i)$ we use a simple approach. Since we are using this approach indoors, we assume the floor to be roughly uniform in intensity and expect to detect large discrepancies in intensity at obstacle points. We then compare the intensity $I(p_i)$ at the point p_i with $I(p_i^+)$ and $I(p_i^-)$:

$$h_I(p) = \begin{cases} 1.0 & \text{if } I(p) > I_{max}, \\ \min\left(1.0, \quad I_{max}^{-1} \max\left(\left|\Delta_I^+\right|, \left|\Delta_I^-\right|\right)\right) & \text{otherwise} \end{cases}, \tag{6.8}$$

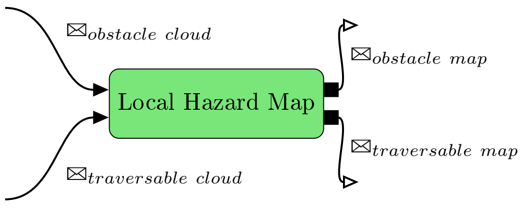
with a free parameter I_{max} for the intensity difference corresponding to a hazardousness of 1 and the differences of intensities

$$\Delta_I^+ = I(p^+) - I(p), \tag{6.9}$$

$$\Delta_I^- = I(p) - I(p^-). \tag{6.10}$$

The measure h_D is thus able to detect obstacles based on changes in curvature, which includes objects and overhanging obstacles. On the other hand, h_I can detect obstacles that are very small and shallow, given that they are made of materials that differ from the floor material. Using evolutionary optimization, we determined the thresholds $\delta_x = 0.12\,\text{cm}$, $P_t = 0.92$ for the probability P_i and $\alpha = 25.57, \beta = 19.18$ for the weighting factors (cf. Section 6.10.1).

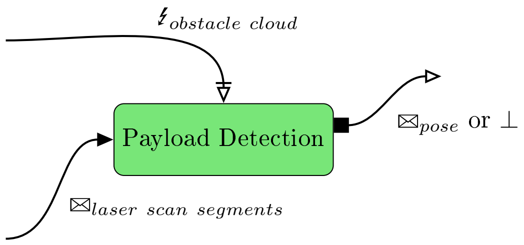
6.5.3 Local Hazard Map



In a last step we perform local obstacle mapping similar to the approach in (Schafer *et al.*, 2008): We keep a rolling window based local grid map to describe the surroundings of the AGV. This way, the map has a constant size and can be efficiently implemented using cyclic buffers. The discretization of the space around the robot smooths out single false detections that are caused by the remaining data artifacts.

The map is split in two parts, namely floor m_f and obstacles m_o. In m_f we remember all points that are classified as non-obstacle and in m_o we remember the obstacle points. Using m_f we can detect negative information by observing an area A in front of the AGV. If a cell of m_f inside A is not mapped, we signal the detection of obstacles caused by missing information. Additionally, we use m_o to signal detected obstacles, if a cell inside A exceeds the threshold P_t defined for the detection itself.

6.6 Payload Detection



After the detection of generic obstacles, we generate payload candidates using 2D information and then verify the presence of a container using this 3D information. The approach requires a 2D laser scanner mounted close to the ground plane that can see the wheels of the containers. The AGV used in this work is equipped with a SICK S300 safety laser scanner with an angular resolution of 0.5°.

In contrast to most 3D sensors, planar laser scanners have a very large field of view of more than 180°. Therefore a detection approach primarily based on laser scans also works out, when the AGV is not directly facing the payload. Furthermore, the minimal measurable distance for a laser scanner is very short, allowing the perception of the wheels of the container while driving beneath it.

By separating detection and verification, we can implement a robust control law to acquire containers. The detection using planar data still provides pose estimates, even when the AGV starts to drive beneath the container. On the other hand, the 3D data is only available during the last few meters of the approach.

Figure 6.11 shows a flow chart of the proposed algorithm pipeline. We do not view the problem of payload detection in isolation, but assume a full perception pipeline for general navigation. As input for the 3D verification we assume a point cloud of detected obstacles to the AGV, as well as a list **S** of connected segments of a laser scan. To generate pose candidates from **S** we use an exhaustive search over all pairs of segments. The verification using 3D data is based on a simple box model. The output is then used to reactively steer the AGV.

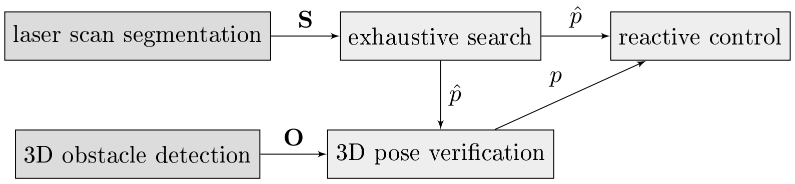
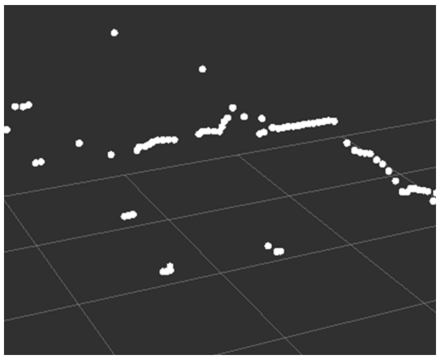


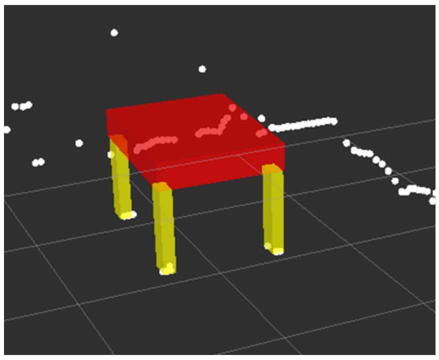
Figure 6.11: Flow chart of the proposed payload detection pipeline.

6.6.1 Container Detection in 2D

The only visible parts of the container in the laser scan are the wheels, as visualized in Figure 6.12a. Therefore, the 2D container detection reduces to detecting laser scan clusters that are appropriately positioned to one another, according to the dimensions of the expected container. Furthermore, no other points are supposed to be seen between the four clusters, because that would either mean that there is an obstacle beneath, or that the clusters do not correspond to a container. Additional difficulty arises from the fact, that the closer two wheels of a container can obscure the ones farther away, such as shown in Figure 6.12a. We therefore cannot assume to see all four wheels at the same time.



(a) Input laser scan.



(b) Partial container detection.

Figure 6.12: Example view of a container in two dimensions.

Laser Scan Segmentation

At first we cluster the laser scan into individual segments using a simple jump-distance-segmentation where two neighboring measurements r_i, r_{i+1} are grouped in the same segment, if their distance $|r_i - r_{i+1}| < \delta_s$ is smaller than a segmentation threshold $\delta_s =$

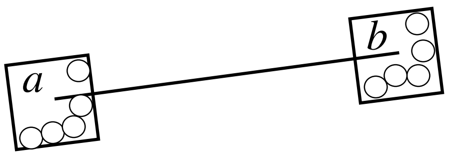
15 cm, which is experimentally selected so that all the laser points on a wheel are grouped together in any orientation. The resulting segments are then filtered using basic heuristics that remove clusters that are too large or too far away.

Exhaustive Search

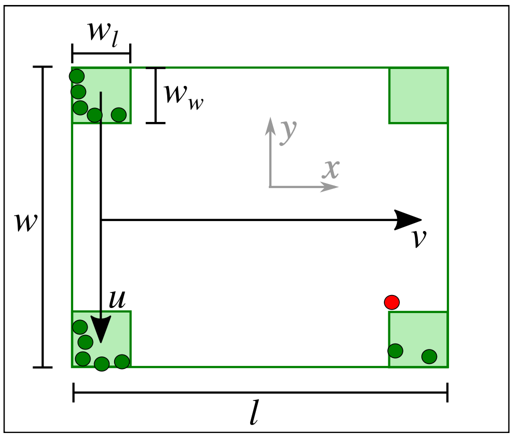
Given a set **S** of laser scan segments, we perform an exhaustive search over all pairs of segments to both determine, whether a container is visible, and to calculate its pose relative to the AGV. We use a two-point model, as visualized in Figure 6.13. Each pair of segments $a, b \in \mathbf{S}$ we assume to be the two forward facing wheels of the container. Let c_a and c_b be the centroids of both segments, where

$$c_s = \frac{1}{2}\left(\max_i(s_{i,x}) - \min_i(s_{i,x}), \max_i(s_{i,y}) - \min_i(s_{i,y})\right)^T \tag{6.11}$$

is the mid-range of segment s, for all points $s_i \in s$.



(a) Two examined segments are assumed to be two wheels.



(b) Container model showing the box model for consensus calculation.

Figure 6.13: Box model of a payload container.

We immediately reject a hypothesis if $||c_a - c_b|| + w_w \notin [w - \delta_w, w + \delta_w]$, i.e. when the distance between the two segments is significantly different from the container width w (cf. Figure 6.13b), with a free parameter $\delta_w = 15\,\text{cm}$ that makes the model more permissive.

Now we calculate the orientation $\hat{\theta}$ of the container as

$$\hat{\theta} = \text{atan2}(\vec{v}_y, \vec{v}_x)^T \tag{6.12}$$

with the base line vector $\vec{u}$

$$\vec{u} = c_b - c_a \tag{6.13}$$

between c_a and c_b and the longitudinal vector $\vec{v}$

$$\vec{v} = (-\vec{u}_y, \vec{u}_x)^T \cdot \frac{l - w_l}{||\vec{u}||} \tag{6.14}$$

along the container.

The center of the container is now given by

$$\vec{C} = c_a + \frac{1}{2}\vec{u} + \frac{1}{2}\vec{v} \tag{6.15}$$

which results in the estimated pose $\hat{p}$ of the container

$$\hat{p} = \left(\vec{C}_x, \vec{C}_y, \hat{\theta}\right)^T \tag{6.16}$$

We interpret $\hat{p}$ as a transformation ${}^{L}T_{O}$ from the coordinate system L of the laser scanner to the local object frame O. We can now calculate the consensus set, which are all points in the laser scan, that agree with a simple axis-aligned box model, which is visualized in Figure 6.13b. For this we transform each point ${}^{L}p$ of the laser scan into the local object frame using transform ${}^{O}T_{L} = \left({}^{L}T_{O}\right)^{-1}$ as

$${}^{O}p = {}^{O}T_{L}\,{}^{L}p. \tag{6.17}$$

Then we can calculate the inliers I by testing for each ${}^{O}p$, whether it is contained in one of the four boxes, representing the wheels (cf. the green boxes in Figure 6.13b). Similarly we find the outliers O, which are all points that are contained in the footprint of the container, but do not belong to any wheel.

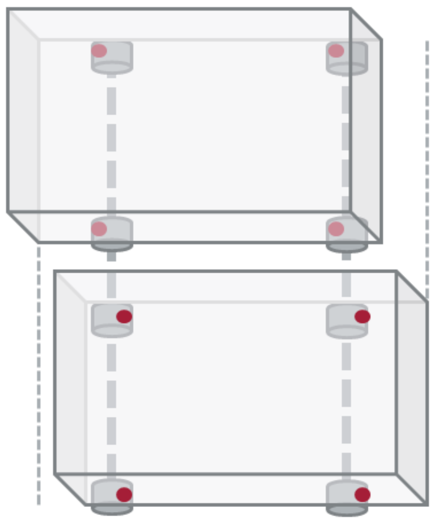
If the consensus set I contains enough points and if there are few enough outliers O, given if $|I| \geq d_{I,2d}$ and $|O| < d_{O,2d}$, the error ε for the hypothesis

$$\varepsilon = (|w - w_w| - ||c_a - c_b||)^2 \tag{6.18}$$

is calculated as the squared difference of the wheel distance from the expected distance $w - w_w$. This way, the procedure will result in the hypothesis which best fits the closest wheels. The farther wheels are not used to calculate ε, since they can be hidden from view. This procedure is iterated N times, where in practice we can observe good results for $N \approx 200$ in cluttered environments.

6.6.2 Container Verification in 3D

The payload pose $\hat{p}$ is determined using data in a plane parallel to the floor, where only the wheels of the container are visible. In a real-world scenario, however, three or four random objects of roughly the size of the wheels can be distributed in such a way, that they become indistinguishable from the payload.


(a) Ambiguity of pose estimation, caster wheel axes are highlighted.

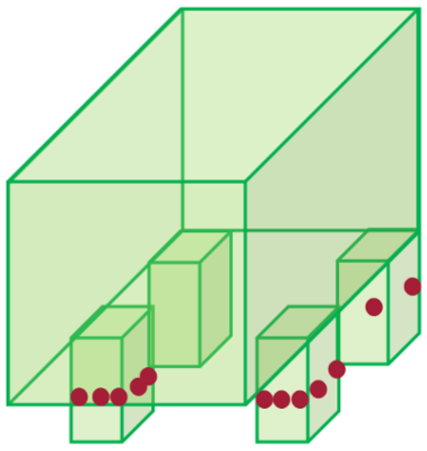

(b) 3D box model of a container.

Figure 6.14: Verification of the 2D detection using 3D data.

Furthermore, typical containers are built using caster wheels, which are passively rotating around an axis which is offset from the center of the wheel. Therefore, we cannot unambiguously recover the 3D container pose from the wheels alone, as demonstrated in Figure 6.14a.

Obstacle Clustering

To solve this problem, we propose to employ the forward looking 3D TOF camera in order to recover the 3D pose. The idea of the verification step resembles the 2D detection: A point cloud captured by the TOF camera is clustered into candidates for the container. First we analyze the point cloud using the approach presented above to separate the points belonging to objects from points belonging to the floor. This approach works well even in environments with uneven ground and with high levels of noise.

This leaves us with a filtered point cloud which only contains points belonging to objects in the environment. For the 3D verification we next apply euclidean clustering similar to (Trevor *et al.*, 2013), using a voxel grid structure for performance reasons. This results in a set **O**, which contains all the detected clusters of 3D obstacle points.

Container Verification

With the set **O** as input, we can recover the pose of the container. First we remove clusters that are too large or to small. Then we use the prior pose $\hat{p}$ in analogy to Section 6.6.1

$$^{O}p = ^{O}T_{PC}\, ^{PC}p \tag{6.19}$$

to transform all cluster points ^{PC}p into the object frame O.

In the second step, we split the points in two sets:

$$\begin{aligned} U &= \left\{ ^{O}p \mid ^{O}p_z \geq h_t \right\}, & (6.20)\\ L &= \left\{ ^{O}p \mid ^{O}p_z < h_t \right\}, & (6.21) \end{aligned}$$

where U contains the points in the upper volume of the container model and L the points in the lower part, i.e. the wheels. Here h_t is the distance between the floor and the body of the container above the wheels.

Now we resolve the ambiguity problem, which mostly consists of a translational error. We iterate all upper points $p \in U$ and estimate the longitudinal offset o_x of the nearest points to the model as

$$\begin{aligned} o_x &= \frac{1}{|F|}\left(\sum_{p\in F} p_x\right) - \frac{l}{2}, & (6.22)\\ F &= \{p \in U \mid |p_x - \tilde{x}| < \delta_o\}, & (6.23)\\ \tilde{x} &= \min\{p_x \mid p_x \in U\}, & (6.24) \end{aligned}$$

where $\tilde{x}$ is the x-coordinate of the closest point $\tilde{p}$ in the upper part U and F is the set of all points in the front of the container whose x-coordinate differs less than δ_o from $\tilde{p}$.

With o_x we can update our prior estimate p of the pose $\hat{p}$ to account for the translational error

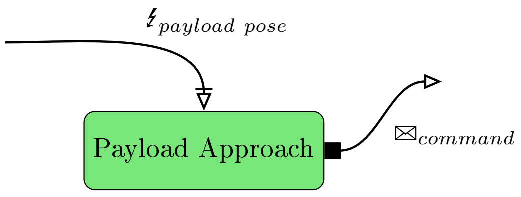
$$p = \begin{pmatrix} \hat{p}_x + \cos(\hat{\theta})o_x \\ \hat{p}_y + \sin(\hat{\theta})o_x \\ \hat{\theta} \end{pmatrix}. \tag{6.25}$$

The inliers I and outliers O are then calculated similarly to the 2D case described in Section 6.6.1. We first apply the offset o_x to all points and then determine two inlier sets I_L and I_U for the lower and upper half of the 3D box model shown in Figure 6.14b. Both sets are again calculated by testing the points for inclusion in an axis-aligned bounding box. We then get

$$\begin{aligned} I &= I_L \cup I_U & (6.26)\\ O &= \{p \in L \cup U \mid p \notin I\}. & (6.27) \end{aligned}$$

We consider a container as verified, if $|I| \geq d_{I,3d}$ and $|O| < d_{O,3d}$ are satisfied, where $d_{I,3d}$ and $d_{O,3d}$ depend on the type of container and are determined experimentally.

6.7 Reactive Control For Payload Acquisition



As a next step, we use a simple control law to steer the AGV beneath the payload. This also demonstrates the real-time capability of the proposed method. This law is implemented using a simple switching control system, where we differentiate between approaching the closest container face and approaching the final pose. The controller is developed for our asymmetric AGV, that consists of a differentially driven base and a passively steered trailer.

Let $\mathbf{b}$ be the pose of the base of the AGV and let $\mathbf{t}$ be the pose of the trailer, expressed in the inertial frame of the odometry. Using a virtual vehicle formulation similar the one described by Egerstedt *et al.* (2001), we search for a desired pose $\tilde{\mathbf{b}}$ for the base on the center line of the container, as shown in Figure 6.15.

Then $\mathbf{e}_b = \mathbf{b} - \tilde{\mathbf{b}}$ is the error of the AGV base and $\mathbf{e}_t = \mathbf{t} - \tilde{\mathbf{t}}$ is the error of the rear-most point on the trailer. To improve numerical stability, we then select $\tilde{\mathbf{b}}$ by minimizing $||\mathbf{e}_b||$ with the constraint $||\mathbf{e}_b|| > \delta_{min}$ for a free minimal distance parameter δ_{min}.

Let $\tilde{\mathbf{b}}_g$ be the final goal pose for the AGV. Depending on the distance the AGV still has to travel, we distinguish two states: In the first state, we position the AGV as closely as possible to the target line while we are approaching the container. In the second state, we position the AGV to the final pose $\tilde{\mathbf{b}}_g$.

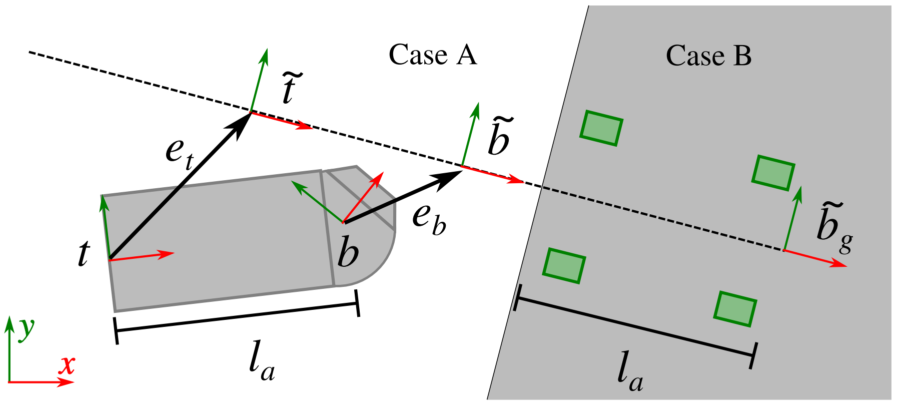


Figure 6.15: The error to the central line of the cart is minimized.

Steering Control

The first of the two states is active, when the distance $||\mathbf{b} - \tilde{\mathbf{b}}_g|| \geq l_a$ is greater than the length l_a of the AGV. In this case, we use the steering angle

$$\psi = \psi_a := k_a \operatorname{atan2}\left(\mathbf{e}_{b,y}, \mathbf{e}_{b,x}\right) + k_b \operatorname{atan2}\left(\mathbf{e}_{t,y}, \mathbf{e}_{t,x}\right), \tag{6.28}$$

where the term dependent on $\mathbf{e}_b$ drives the AGV base towards the line and the term consisting of $\mathbf{e}_t$ the trailer. By enforcing $\delta_{min} \gg 0$, we avoid numerical problems. Experimentally, we determined $k_a = 0.7$ and $k_b = 1.8$.

The second state is reached, when the distance $||\mathbf{b} - \tilde{\mathbf{b}}_g|| < l_a$ is smaller than the length of the AGV. In this case we can assume, that the AGV is already close to the target line and that only small steering commands are necessary. We choose as control input

$$\psi = \psi_b := k_c \mathbf{e}_{b,y} \tag{6.29}$$

the error in y direction relative to the container (cf. Figure 6.13b), scaled by a free parameter k_b. Experimentally, we determined the value $k_c = 0.6$.

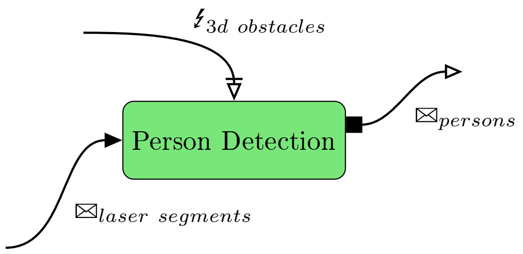
Velocity Control

Finally, we control the linear velocity of the AGV with a simple proportional controller

$$v_x = \max\left(v_{min}, \min\left(v_{max},\ k_v \mathbf{e}_{b,x}\right)\right) \tag{6.30}$$

which gives another control parameter k_v. The velocity falls off linearly with the distance to the resting pose and is constrained to be in the interval (v_{min}, v_{max}).

6.8 Multi-sensor Person Detection

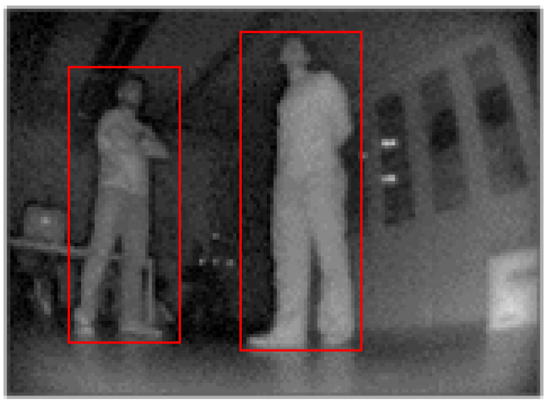


One of the aspects of the PATSY projects is the detection of persons around the AGV. This is done in order to enable the system to be more careful in the vicinity of inexperienced people. As with the detection of payload containers, we make use of the already classified depth information coming from the obstacle detection module. Similarly, we use a segmentation of the planar laser scanners to track people that are not visible in the limited field of view of the TOF camera.

6.8.1 3D Person Detection

As a first step we again apply a clustering step to generate candidates for the presence of persons using the detected obstacles. We do not directly use the same clustering parameters as in the payload detection, so that the parameters can be independently adjusted and optimized. For each cluster we calculate a region of interest (ROI) in the corresponding intensity image (cf. Figure 6.16a). Every ROI is then processed individually. First we extract the corresponding sub-image I_{ROI} from the intensity image I, which is trivial in structured point clouds. We then calculate a feature descriptor for the image, which can be either HOG (Dalal and Triggs, 2005) or ACF (Dollár *et al.*, 2014). These descriptors are then classified by a machine learning algorithm. For this we compare Random Forests (RF), Support Vector Machines (SVM), AdaBoost and Multi-layer Perceptrons (MLP) in Section 6.10.3.

In addition to the 3D information, we use the planar laser scanners to detect persons all around the vehicle. Similarly to the container detection, we segment the laser scans into clusters. A Random Forest is then used to predict for every cluster, whether it might be a leg of a person. Due to the low information density, however, we cannot reliably determine, whether a segment really belongs to a person.


(a) Regions of interest classified as persons.

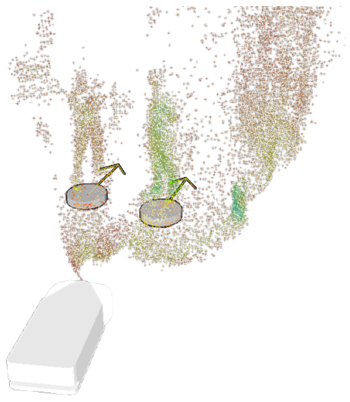
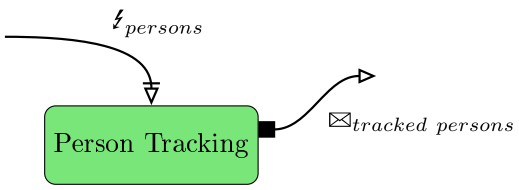

(b) Two people are tracked in 2D and 3D.

Figure 6.16: Person detection and tracking.

6.8.2 Person Tracking



The output of the person detection module are direct measurements of people. Detections from 3D data can be used with high confidence, those from planar laser scanners only with low confidence. In order to fuse both measurement types, we use a

Multi-Hypothesis Tracker (MHT) based on the Kalman Filter, similar to the algorithm proposed by Arras *et al.* (2008). The tracker is implemented as a *buffer* (cf. Section 3.6.3) with asynchronous inputs for detected persons.

For each detected person we find the closest existing hypothesis and update it with the measurement. New hypotheses are created for measurements that cannot be assigned to an existing hypothesis. We use two different update functions, one for 2D detections and one for 3D detections. This way we can weaken the update for the more error-prone leg detection relative to the more accurate camera based detection. The combination of both detectors is beneficial for the performance, because we can tolerate missing detections or false positives of both individual modules. Figure 6.16b shows the same situation as seen before, now displaying two hypotheses for the two persons.

6.9 Vector-based Adaptive Monte Carlo Localization

To localize the AGV we use Monte Carlo Localization (MCL) based on architectural maps in vector format. Particle filters are used to approximate the posterior probability with partially observable Markov chains (Thrun, 2002). This is then used to estimate the pose x_t of a robot at time t from the complete history of measurements $z_1, \dots z_t$ and all control commands $u_1, \dots, u_t$. In the particle filter, this is done by estimating the belief

$$bel(x_t) = p\left(x_t \mid z_1, \dots z_t,\ u_1, \dots u_t\right) \tag{6.31}$$

by keeping a set S_t

$$S_t = \left\{ x_t^{[n]} \mid 1 \leq n \leq N \right\} \tag{6.32}$$

of N particles.

In the case of an AGV, there usually exists a precise plan of the workspace. Architectural plans, such as the one shown in Figure 6.17, are very memory efficient compared to occupancy grid maps, due to the continuous representation. The lack of discretization also makes them more precise, assuming that they accurately represent the actual building.

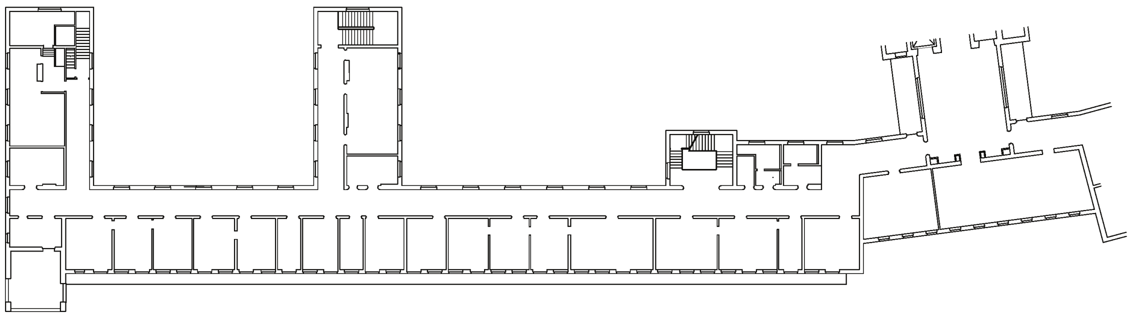
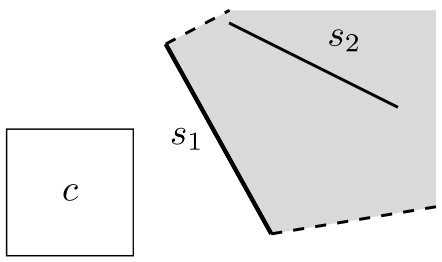


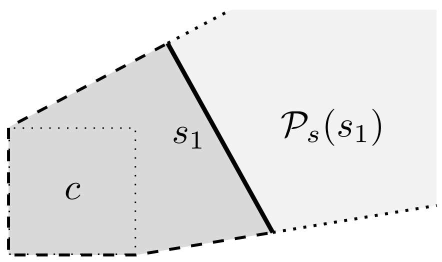
Figure 6.17: An architectural map of the Cognitive Systems chair.

One way to use such an architectural map would be to discretize it into an occupancy grid. This would, however, negate the previously mentioned advantages. A large architectural map $\mathcal{M}$ may contain thousands of line segments. Without pre-processing, each of those lines would have to be intersected with every hypothetical laser beam, in order to calculate the likelihood of each particle.

To reduce the computational effort, we create a lookup table $\mathcal{I}$ for discrete chunks of the environment. $\mathcal{I}$ is three dimensional, using the x-y position and the yaw-orientation θ. When we need to generate a virtual laser beam at pose $p = (x, y, \theta)$, we look up the cell c_p that p is located in. The possibly visible segments are then used for intersection calculation. To determine the visibility, a shadow polygon $\mathcal{P}_s(s_i)$ is calculated for each segment (cf. Figure 6.18a). All segments that do not lie in any shadow polygon of another segment are then stored in $\mathcal{I}$. The shadow $\mathcal{P}_s(s_i)$ is calculated by finding the convex hull of the cell c and both ends of s_i. The newly added segments are extended into infinite lines, as demonstrated in Figure 6.18b.



(a) The segment s_2 is invisible to the cell c because it is shadowed by s_1.



(b) Construction of $\mathcal{P}_s(s_1)$ using the convex hull of c and s_1.

Figure 6.18: Calculation of visible segments relative to cell c.

6.10 Experimental Results

6.10.1 Obstacle Detection

Quantitative values for the obstacle detection pipeline are given on a frame-by-frame basis. The complete system is demonstrated in a video.[1] Table 6.1 shows the average runtime over 10000 frames recorded while driving the AGV through various environments. As can be seen in the table, the average runtime required is 7.5 ms, which corresponds to a frequency of ca. 120 Hz, which is clearly real-time since the employed sensor has a maximum frequency of 60 Hz.

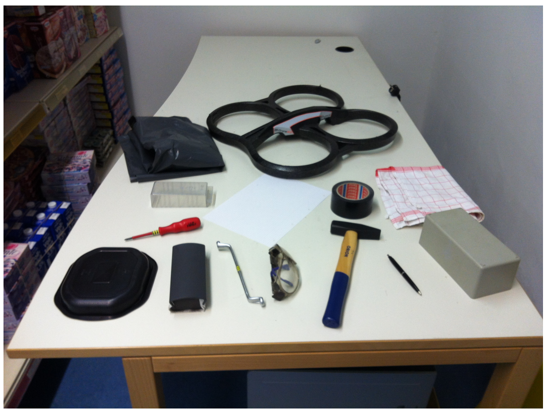
To select the optimal parameters, we created a set of ca. 50 manually labeled point clouds, where we specified for each point whether it is caused by an artifact or whether it

[1]A video demonstration can be seen at https://youtu.be/vn3cyXHYyHQ

Table 6.1
PERFORMANCE RUNNING ON THE AGV

Runtime in ms	Mean	Standard deviation
Conversion	0.56	0.38
Intensity Filter	0.27	0.28
Edge Filter	0.48	0.42
Classification	6.20	2.01
Overall	7.50	3.09

should be classified as an obstacle or as floor. Points that are ambiguous or not relevant have not been not labeled. New scenes were added to the set, if they proved difficult to classify. We focused on small objects, since detection of objects larger than ca. 10 cm was found to be very stable. Using the evolutionary algorithm framework EvA2 developed by Kronfeld *et al.* (2010), we optimized all parameters using the AUC as the fitness function. Finally, we selected the threshold $P_t = 0.92$ using the optimal ROC curve. Included in the data are small objects that we placed in front of the AGV. Also included are different floor types, including stone, PVC, wood and concrete.


(a) Objects included in the manually labeled data set are mostly smaller than 10 cm.

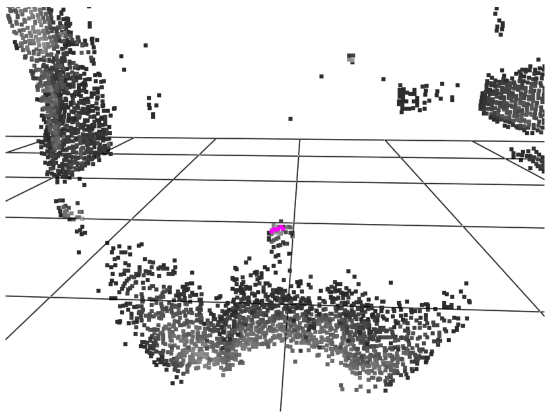

(b) Detection result for a screwdriver at a distance of 2 m. The detected points are highlighted.

Figure 6.19: Small objects contained in the test data set.

Due to the resolution of 160×120 pixels of the Fotonic E70P, there is a limit for how small an object we can detect. To evaluate the effective detection range of the algorithm, we created a second data set by placing small objects in front of the AGV at various distances, as can be seen in Figure 6.20. We measured the number of detected obstacle points and reported an object as detected, if there were at least 10 detected points. A subset of the objects can be seen in Figure 6.19a. The use of the sum of intensity and depth based detection allows the detection of weakly reflecting and transparent objects, although both components are only weakly discernible. We found that these objects, which include dark and clear plastics, as well as glass, can be detected up to a distance of ca. 1.3 m. Detecting tools like screwdrivers (cf. Figure 6.19b), hammers and wrenches proved to be possible up to a distance of 2.3 m. Very small and weakly reflecting objects cannot be detected reliably, since neither the depth nor the intensity data is sufficient to gain information about the objects. Larger objects and overhanging obstacles can reliably be detected up to 5 m.

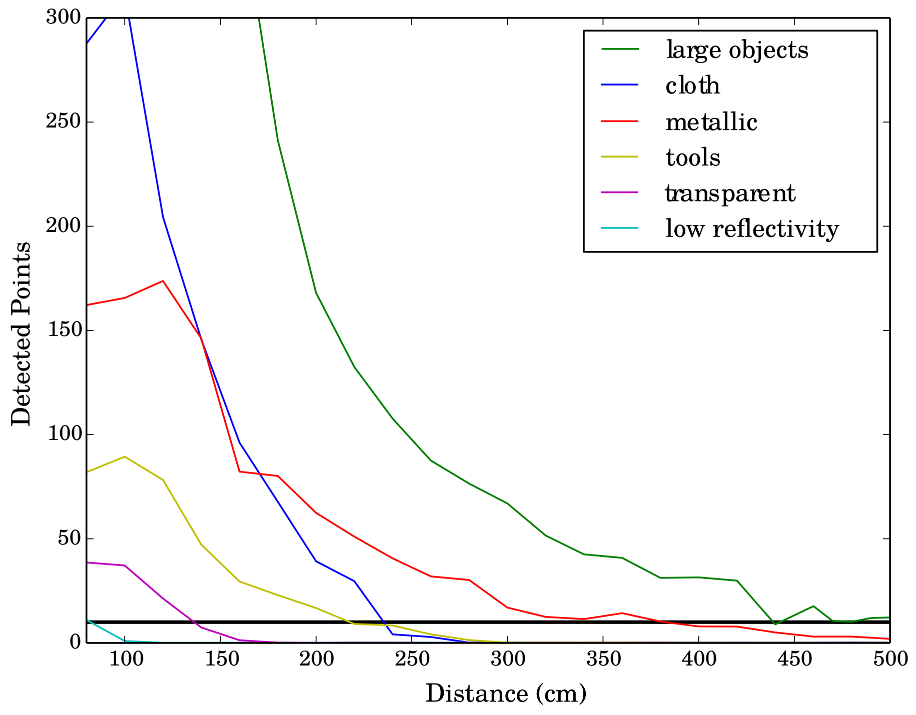


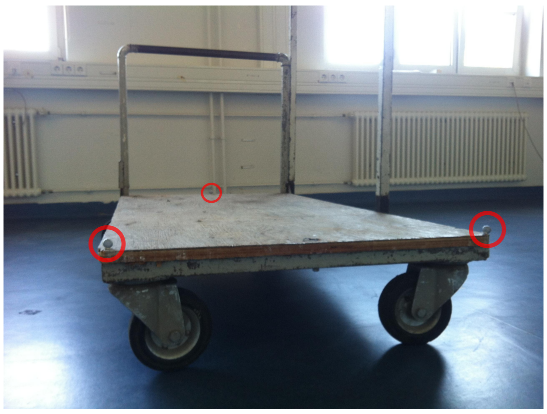
Figure 6.20: The number of detected points for different object classes.

6.10.2 2D Container Detection

We evaluated the container detection algorithm in three experiments: First we determined the 2D detection accuracy using a stationary laser scanner and a four-wheeled cart in a tracking system. In the second experiment we used the full pipeline to evaluate the accuracy while the AGV was autonomously driving beneath a realistic container mock-up. A final experiment was done in simulation to evaluate the approach of a payload cart from many different initial poses.

Stationary 2D Detection

We evaluated the 2D localization accuracy using a four-wheeled cart in place of a real container, which is shown in Figure 6.21a. The 2D sensor was mounted at the same height as in the AGV. Both the pose of the cart and the pose of the laser scanner were tracked using OptiTrack.


(a) A cart equipped with tracking markers (highlighted in red.)

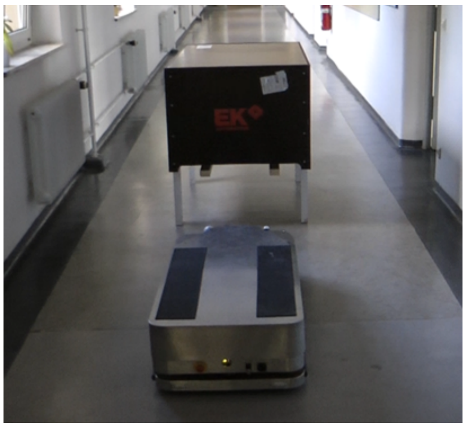

(b) A mock-up of a real container is used for experiments.

Figure 6.21: Experimental setup.

The experiment was performed using a stationary laser scanner and a moving cart. The motion of the cart was varied to simulate different scenarios:

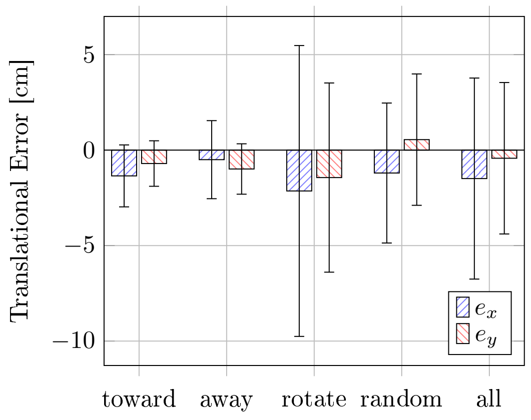
1. Straight motion. The cart's motion leads towards or away from the scanner. From the reference frame of the laser scanner, this is equivalent to the application scenario, where the AGV is driving towards the cart.
2. Rotation. The cart is rotated around its central axis at varying distances.
3. Random. The cart is pushed and rotated randomly with directional changes that cause the wheels to have non-parallel orientations.

Figure 6.22 shows the detection rates of the tracked cart in these scenarios and Table 6.2 shows the detection rates.

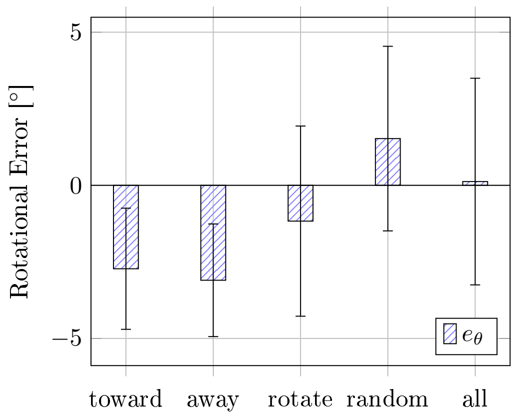
Table 6.2
EXPERIMENT 1: TRUE AND FALSE-POSITIVE-RATE

	toward	away	rotating	random	combined
TPR (%)	93	92	87	85	87
FPR (%)	0	0	1	1	1

The combined translational error is $-1.49\,\text{cm} \pm 5.26\,\text{cm}$ in x direction and $-0.43\,\text{cm} \pm 3.97\,\text{cm}$ in y direction. The average orientation error is $0.126° \pm 3.323°$. The larger uncertainty in x direction is expected, do to the asymmetric caster wheels. This also shows that the 2D detection is not sufficient to localize the payload. In the combined case, 87% of the time the container can be detected, whereas false detections only happen rarely.



(a) Signed translational errors.

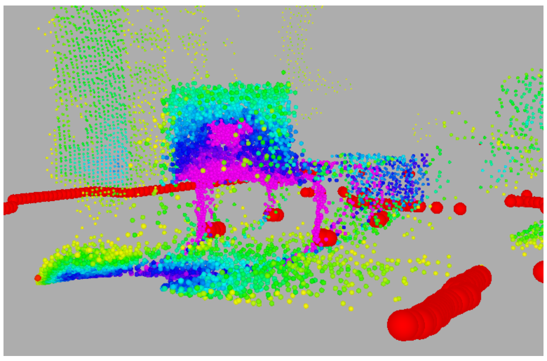


(b) Rotational error (yaw angle.)

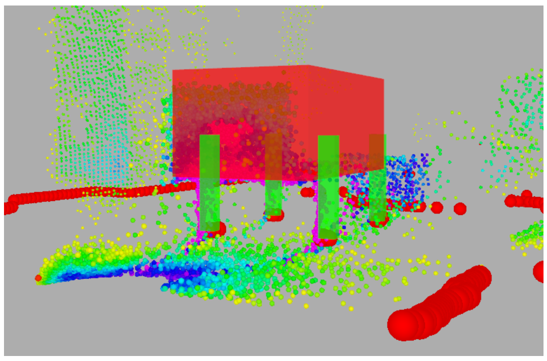
Figure 6.22: Results of 2D tracking of a cart.

Full Container Acquisition

The second experiment shows the combined approach of using both 2D and 3D information. This experiment was performed on the real AGV using a mock-up of a payload cart. The mock-up, which is shown in Figure 6.21b, was built to match the dimensions of a real cart, where the poles result in an equivalent laser range scan as four parallel caster wheels.



(a) 2D and 3D capture of the container mock-up.



(b) Detected Pose and height of the container.

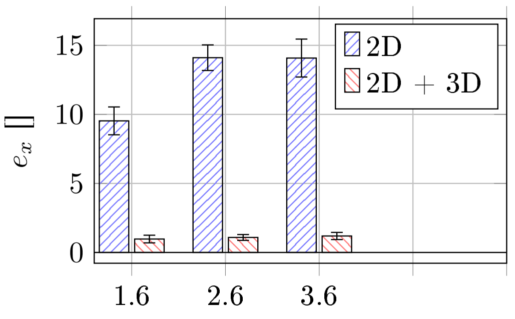
Figure 6.23: Experiment 2: Combined detection and approach.

Figure 6.23 shows the data of both sensors, while the AGV is approaching the container. Figure 6.24a shows the average error along the x-axis for ca. 4000 measurements with different poses at three distances. The error along the y-axis is shown in Figure 6.24b. As desired, the 3D information considerably reduces the error in x direction.

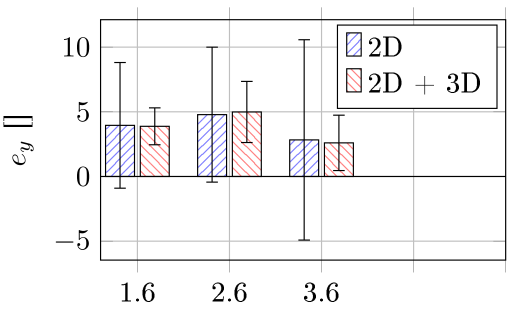
The run times of the individual steps are shown in Table 6.3, evaluated on the AGV. The complete pipeline has a delay of ca. 16 ms, which makes it real-time capable with 60 Hz. Once the 3D verification has been successful, only 2D information is needed for the controller. The 2D sub-system is decoupled and allows for ca. 160 Hz, which is far higher than the 12.5 Hz operating frequency of the employed laser scanner.

Table 6.3
CONTAINER DETECTION RUN TIMES

Step	Mean [ms]	Standard Deviation [ms]
2D Segmentation	0.632	0.023
2D Payload Detection	3.247	0.837
Clustering	0.765	0.235
3D Verification	0.686	1.162



(a) Error e_x in x direction.



(b) Error e_y in y direction.

Figure 6.24: Accuracy using 3D verification depending on the distance to the container.

Simulation

In the last experiment we evaluated the approach towards a container cart from various different poses. This experiment was done in simulation, where we fully simulated the robot's kinematics. The start pose of the AGV was varied on a grid (cf. Figure 6.25) with $x \in [-2\,\mathrm{m}, 2\,\mathrm{m}]$ and $y \in [-4\,\mathrm{m}, 4\,\mathrm{m}]$ with a step size of 0.5 m in both directions. The pose of the container was set to $(4\,\mathrm{m}, 0\,\mathrm{m})$. The orientation was varied as $\theta \in \{-0.5, 0, 0.5\}$. Figure 6.25 shows the results, where every starting pose from which the container can be successfully picked up is visualized.

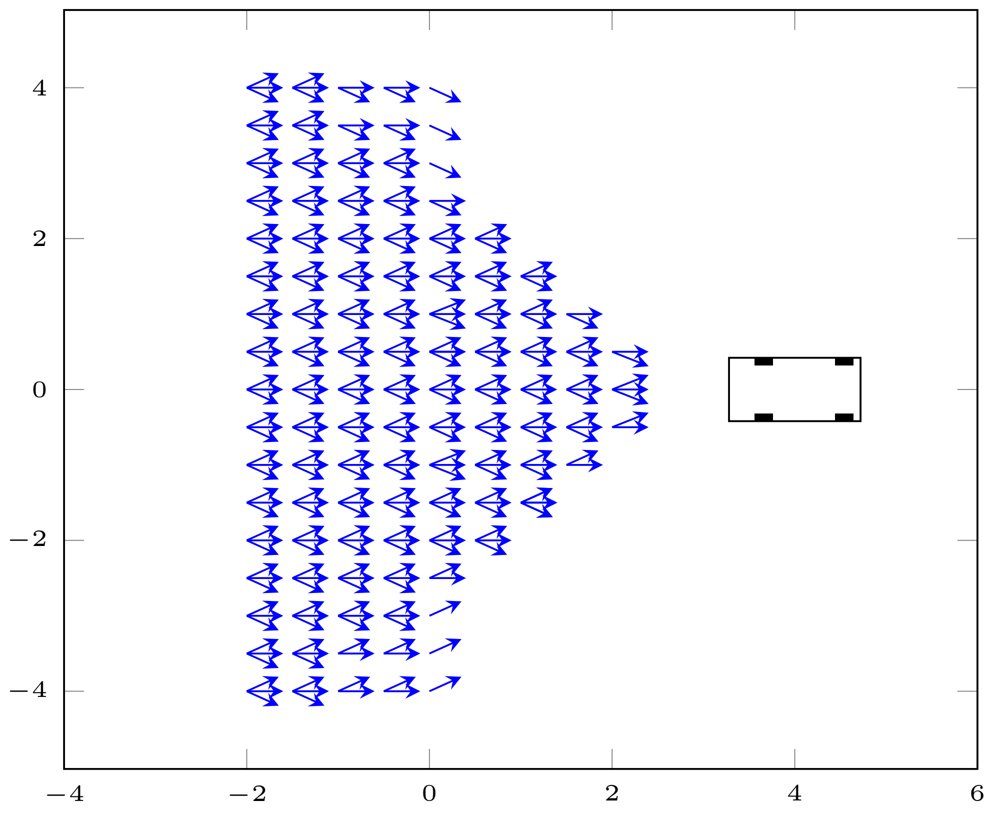


Figure 6.25: Simulated approach from various initial poses. Shown are the container (black) and all poses from which the approach is possible (blue).

6.10.3 Person Detection

The person detection using 3D data was evaluated on a dataset of ca. 6,000 sensor measurements of the Fotonic E70P, which have been manually annotated with ca. 16,000 annotations. The data set is intentionally designed to be challenging.

Table 6.4
PERSON DETECTION STATISTICS

Descriptor	ACF			HOG		
Classifier	SVM	RF	AdaBoost	SVM	RF	MLP
TPR (%)	27	74	75	78	76	78
FPR (%)	5	21	20	22	27	23

A large portion of samples contains more than one person and there are many partially visible people. Table 6.4 contains the statistics of all evaluated features and classifiers. The strongest variant for this data set is the HOG descriptor in combination with an SVM, albeit all variants except for ACF with SVM are comparable. A true positive rate of 78%

can be achieved, which is sufficient for the PATSY project but should be improved in the future. A more detailed evaluation with more recent experiments can be found in (Hanten, Kuhlmann, Buck, Otte, and Zell, 2018).

6.11 Discussion and Conclusions

We have presented an architecture for a person detecting AGV, mainly based on the Hybrid Flow Graph model. The key component is a simple and efficient approach to detect obstacles in 3D point clouds that can deal with various data artifacts caused by the measurement principle of TOF cameras. By using both depth and intensity data we can detect objects of various sizes, ranging from less than 10 cm to several meters. Small objects cannot be detected without using intensity data, because of the high signal to noise ratio, especially in sunlight. The downside of using intensity values is that the floor has to be rather uniform. However, since we are using the gradient of the intensity, this is only problematic if there are very high contrast changes between floor types. Using evolutionary optimization we have found parameters that work with a variety of floor types.

Our obstacle detection algorithm is kept as generic as possible and is directly applicable to any TOF camera system. The approach has also been tested on a 3vistor-T TOF camera, which is developed by the SICK AG. Compared to the E70P, the 3vistor-T camera employs multiple frequency scanning and therefore does not suffer from wrapping artifacts. The floor plane, however, is also not measured completely flat and cannot be measured further than with the E70P. The distortions of floor are largely equivalent for both sensors and interpolation artifacts on object corners have to be filtered in both cases. We classify each column of the point cloud independently, which means that the algorithm can be directly used on spinning LIDAR, RGB-D or stereo systems. However, the artifact filtering step is mostly necessary for TOF cameras. We have also used a variant of the presented algorithm for an outdoor robot in rough terrain by adjusting the hazardousness function.

Transportation of payload carts is the central task for single unit AGVs. We have shown that the location of these carts can efficiently and robustly be determined in real-time on the vehicles themselves. Such an approach could replace the state of the art in real application scenarios, where containers can usually only be collected in sending stations with arrangements to perfectly align the wheels of the carts. The unconstrained payload acquisition allows AGVs to navigate freely, i.e. AGVs can be made independent of predefined paths and specialized stations to send or receive payloads. Instead, any area in the environment can become a (temporary) designated station, without the need for additional hardware.

The results of the generic obstacle detection are also used to detect people around the AGV. We have presented a fast person detection method based on candidate proposals generated from obstacle clusters. In combination with a tracking step, the results are good enough to allow the AGV to safely navigate in populated areas.

It remains a matter of future work to replace the last and most aggressive filtering step of the obstacle detection with more sophisticated models, such as the probabilistic phase unwrapping approach by Droeschel *et al.* (2010). Future work also includes the creation of a benchmark of a variety of objects under different conditions for a more systematic evaluation. Concerning the detection of payload, future work includes the use of more than one container model to differentiate between different types of payloads. Additional work includes the implementation of a least-squares fit to further refine the search results, which could improve the accuracy even further.

In this chapter we have presented the individual modules of the PATSY architecture in detail. We have, however, omitted details about the high level control structures needed to make the system fully autonomous. These will be presented in Chapter 8, once we have introduced the concept of activity.

Chapter 7

Activity Flow Graphs

In the previous chapters we have derived the first two layers of the flow graph hierarchy that pervades this work. At first we have defined the SDF+ model, which separately implements both synchronous and asynchronous data flow. With HFG we have then unified both data flow models into a coherent framework, which is a hierarchical data flow model. In both cases, however, we still need additional structures to model high level state that changes orders of magnitudes less frequently than the data flow.

A common way to model any complex system is using the various forms of UML diagrams. Finite-state machines (FSMs) are often used, both in the design and as the foundation of the implementation. They are, however, not really capable of modeling concurrent or data-driven processes.

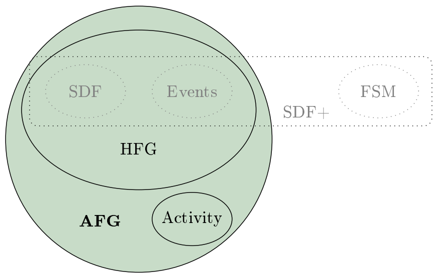


Figure 7.1: Activity Flow Graphs (AFG)

In this chapter we propose a novel way to model high level robot control based on data flow. Many existing approaches (Ethan Rublee *et al.*, 2015; Biggs *et al.*, 2011; Hart *et al.*, 2014) that model perception and mission control in a single framework use a combination of multiple models, such as data flow for perception and finite-state machines for high level mission control. Statemate (Harel *et al.*, 1990) uses three separate model types: Module, state and activity charts. We, however, want a coherent graphical computation model that can be used to model both perception and high level mission control in a single framework.

Other approaches base their unified model on state-machines and extend them by also modeling data flow. RAFCON by Brunner *et al.* (2016), implements state machines that support hierarchies and concurrency. The data flow is used to represent parameters and return values for the states. The system needs a separate solution for the implementation of perception tasks. Sequentially constructive statecharts (von Hanxleden *et al.*, 2014) are another way to represent synchronous computation based on state machines, which are designed for safety-critical applications.

Flow diagrams are also common in workflow specification for businesses. Data flow and control flow are used in business process management to analyze and verify workflow processes (Trcka *et al.*, 2008; Sadiq *et al.*, 2004). As with the similar sounding ActivityFlow (Liu and Pu, 1997), a workflow process schema typically specifies activities that constitute the workflow process and dependencies between these activities. Activities thus represent steps required to complete a business process. In our model, we view activity as an abstract property that flows through a graph, the same as with data and control flow.

We eliminate the need for an FSM by introducing the concept of *activity*. This approach is inspired by UML Activity diagrams (Dumas and Ter Hofstede, 2001), which are capable of modeling both computational and organizational processes. We show that the expressiveness of FSMs can be translated into HFG with minimal changes to the model, resulting in *Activity Flow Graphs* (AFG) . In Chapter 8 we show how to apply AFG to modeling high level robotic mission control.

7.1 Activity Flow Graphs

7.1.1 State Representation

So far we have not formally defined how a flow graph and an FSM can interact. This is due to the fact that flow graphs are focused on data flow and FSMs are focused on control flow and state representation. Informally, we assume that state machines and flow graphs interact asynchronously, which can be implemented using events in both the SDF+ and the HFG models. Every state of an FSM can therefore send and receive signals, as visualized in Figure 7.2.

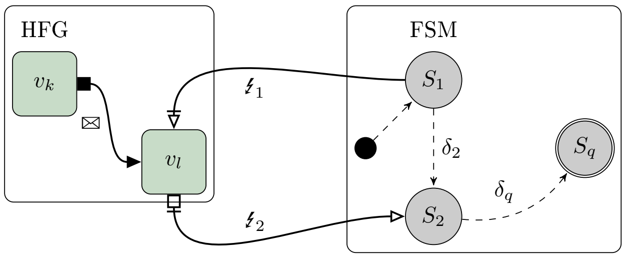


Figure 7.2: Technical example of an interaction between an HFG and an FSM to wait for a message ✉: First state S_1 emits signal $↯_1$ and transitions via δ_2 to state S_2. $↯_1$ is handled by v_l, which sends $↯_2$, once ✉ is received. Only now will the FSM transition to S_q via δ_q.

Figure 7.2 demonstrates a solution to a stateful problem, where the system has to wait for the arrival of a specific message, before it can continue. This could be an AGV that is waiting for a container to be placed in its vicinity, as described in Chapter 6. The example demonstrates one of the main reasons, why HFG alone is not sufficient to model more complex systems: Flow graphs are inherently state-less, with all components representing functional units. A sophisticated robotic system, on the other hand, always needs to consider the global state of the robot and its mission.

We can also see that the vertices of the graph can be split into two disjoint sets, one representing data flow and the other representing the FSM, where interactions between the two sub-graphs are only possible via asynchronous message passing. Instead of giving a more precise definition for FSMs in the context of flow graphs, we directly generalize the concept of states and state machine graphs into a coherent model we call *Activity Flow Graphs* (AFG).

7.1.2 Activity Flow

Let us again consider the exemplary situation from Figure 7.2. If we only observe the state machine, we can interpret the FSM graph as a data flow graph: Each state is a data flow node and the transition edges between states are asynchronous connections. When the FSM transitions from state S_1 to S_2, we can imagine a message being sent via the transition δ_2. The FSM always has a single *active* state, which is why we can say that the message sent via δ_2 transfers the *activity* of the graph. This also holds in general, there is always exactly one active node in an FSM and the activity can only be transferred using a transition.

We now transfer the activity concept to HFG, with the aim to eliminate the need for an FSM altogether. Let $G = (V, E, P_*)$ be a flow graph according to the HFG model with the nodes V, edges E and graph ports P_*. For each $v_k \in V$ we define an *activated* attribute

$$A(v_k) \in \{0,\ 1\}, \tag{7.1}$$

where 1 indicates an activated node. We initially set $A(v_k) \leftarrow 0$ for all $v_k \in V$. The behavior of an activated node is the same as that of an active state in an FSM, i.e. it stays active until the activity is transferred to another node. Importantly, whether a node is activated or not does not influence the behavior of the node in the data flow. If no activity is injected into the graph, we have a pure HFG model. Otherwise, an activated node still participates normally in the data flow, except for specially implemented nodes that behave differently depending on their activation.

For a node to become active, it has to receive the activity via a connection, i.e. via a message passed from another node. We therefore define an *activity modifier* attribute

$$a(\bullet) \in \{0,\ +1,\ -1\} \tag{7.2}$$

for tokens, which can take on one of three values: 0 representing no change, $+1$ representing an activation and -1 a deactivation. Then we can define a simple update rule for the activity of any node v_k upon receiving a token to be

$$A(v_k) \leftarrow \max\left(0, \min\left(1, A(v_k) + a\left(\bullet\right)\right)\right), \tag{7.3}$$

which means that an inactive node receiving a token with $a(\bullet) = 1$ becomes activated. Likewise, if an activated node receives a token with $a(\bullet) = -1$, it will be deactivated.

In order to control the flow of activity, we need an equivalent structure to the transition in FSMs. We therefore define an *active* attribute for connections $e_k \in E$ as

$$A(e_k) \in \{0,\ 1\}, \tag{7.4}$$

such that only *active* connections can transfer the *activated* property of a node to another.

Let $E_A = \{e_k \in E \mid A(e_k) = 1\}$ be all active edges and $E_{\neg A}$ all the inactive edges of G. A second update rule, this time for the tokens sent via connection e_k, is then given by

$$a(\bullet) \leftarrow a(\bullet) \cdot A(e_k), \tag{7.5}$$

which means that all tokens sent via an $e \in E_{\neg A}$ will be assigned a modifier of 0. This way, only active connections can transmit tokens with $a(\bullet) \neq 0$. Whereas the *activated* state of a node changes over time to reflect the state of a higher level system, the *activity* of an edge is a fixed part of the graph structure. Since only active connections can transfer a token with an activity modifier, $G_A = (V, E_A, P_*)$ forms a sub-graph of G that describes the flow of activity between nodes. Note here that $E_A \subseteq E$, so each $e_k \in E_A$ still participates in the regular data flow.

The transfer of activity is then defined as follows: When a node v_k produces a token $\bullet$ via an output, we set

$$a(\bullet) \leftarrow A(v_k), \tag{7.6}$$

so that a node sends active tokens while it is active itself. Sending an active token should, however, deactivate the sending node to achieve a similar behavior to an FSM. Therefore we set

$$A(v_k) \leftarrow 0, \tag{7.7}$$

once v_k has sent a token with $a(\bullet) = +1$.

This way, analogously to an FSM, activity is transferred between nodes via edges, however we do not have to define additional node or port types. Instead, every output ${}^kO$ and every event ${}^kE$ of node v_k is able to relay activity and v_k can be implemented in an activity-agnostic way. This definition results in a more general model than FSM, however, since an active node can send activity to more than one succeeding node at a time. Additionally, nodes can become activated from other sources, e.g. any inactive node can send a signal $\lightning$ with a $a(\lightning) = +1$.

In AFG, more than one node can be active concurrently, which means that not every active sub-graph $G_A = (V, E_A, P_*)$ can be represented as a UML state diagram. Using the familiarly sounding UML *Activity Diagram*, we can, however, represent the activity sub-graph G_A, if we restrict the generation of activity to a single source.

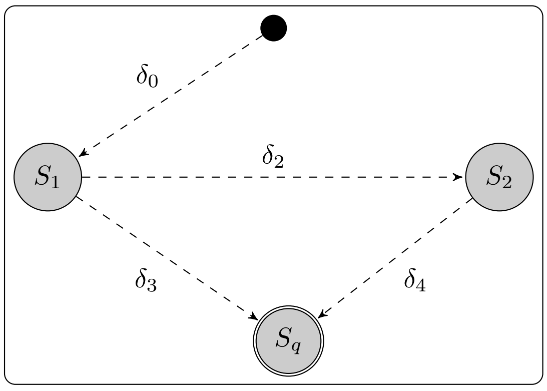
7.2 Exemplary Usage

7.2.1 Finite-State Machine

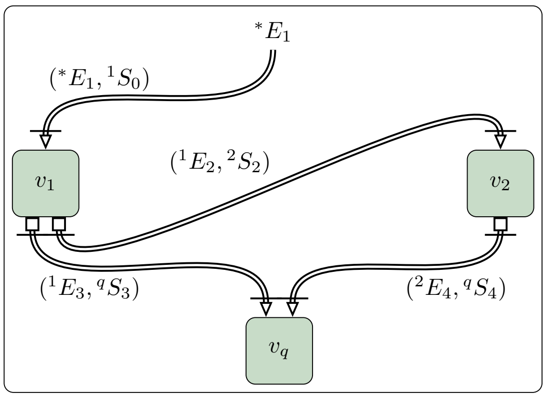
We first show that activity flow graphs are capable of representing any finite-state machine, by translating a general FSM into AFG. Let $(\Sigma, S, s_0, \Delta, F)$ be a finite-state machine where Σ is the input alphabet, $S = \{s_0, \ldots, s_n\}$ is the set of all states, s_0 is the initial state, Δ is the set of all transitions and F is the set of all final states. We construct an AFG $G = (V, E, P_*)$ in the following way:

1. Create a graph event ${}^*E_1 \in P_*$ that emits an active token to initialize the graph and represents the initial state s_0.
2. For each state $s_k \in S$, except for s_0, create a node v_k representing that state.
3. For each transition $\delta_t = (s_i, s_j) \in \Delta$ between state s_i and s_j:
    a) If $s_i \neq s_0$ create an event ${}^iE_t$ for node v_i and a slot ${}^jS_t$ for node v_j and add an active connection $({}^iE_t, {}^jS_t)$ to E.
    b) Otherwise create a slot ${}^jS_t$ for node v_j and add an active connection $({}^*E_1,\ {}^jS_t)$.
4. Implement each node v_k analogously to the implementation of state v_k. Instead of invoking a transition $\delta_t = (s_i, s_j)$, publish an active signal token $\not{t}$ with $a(\not{t}) = +1$ on the event ${}^jS_t$. Ensure for each node, that only one active token is sent at a time, otherwise G implements a non-deterministic finite-state machine.

Figure 7.3 shows an example of this transformation.



(a) Finite-State Machine



(b) Activity Flow Graph

Figure 7.3: Translation of an FSM into the AFG framework. States are replaced by nodes, transitions are modeled as connected pairs of events and slots. The initial state is modeled as a graph port.

7.2.2 Data Flow and High Level State Interaction

We can implement a solution to the problem shown in Figure 7.2 using activity flow. The idea is to wait, until a node v_k has produced a message and only afterwards continue propagating the high level state. This can easily be modeled using AFG, as demonstrated in Figure 7.4. We can use a buffering node v_b, as introduced in Section 3.6.3. A node v_k is sending a message to v_b, which simply relays it in form of an event to S_q.

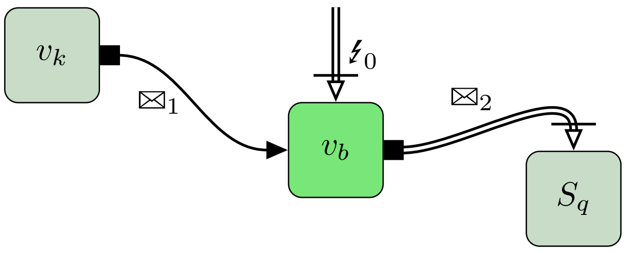


Figure 7.4: A solution to the same problem as in Figure 7.2, with active edges shown doubly lined. The buffer node v_b relays incoming tokens $⊠_1$ to $⊠_2$. Once $↯_0$ activates node v_b, the buffer transfers the activity to S_q with the next $⊠_2$.

As long as v_b is inactive, the signals $⊠_2$ will have a modifier of $a(⊠_2) = 0$, so S_q will not be activated. However, once a $↯_0$ token with $a(↯_0) = +1$ arrives, v_b will be activated. The next relayed $⊠_2$ will then have a modifier of $a(⊠_2) = +1$, deactivating v_b and activating S_q. We therefore have constructed a graph where S_q will be activated only after v_k has produced a message.

Note that we do not have to consider activity in the implementation of v_k, v_b or S_q, all of which are completely activity-agnostic. Only when constructing the graph do we have to consider which edges have to be active for the activity flow to be correct. We have a lower complexity than in the solution shown in Figure 7.2, which needed two different graphs with completely different semantics, as well as many more nodes and edges.

7.2.3 Unified Data Flow and Robot Control

Previous examples have shown how we can use the activity concept to control the current state of a robotic system from within the data flow. Additionally, we can trivially model the reverse situation, where we control the data flow according to the high level state. This can be used to activate parts of a perception pipeline exactly when it is needed and thereby conserve computational power when it is not needed.

With SDF+ and HFG we have used an FSM to deactivate parts of the system, where we needed additional complexity and where we did not have fine-grained control over the data flow. With AFG we can, for example, throttle a currently unimportant sub-graph to a lower frequency instead of completely deactivating it. All of this can be achieved, without any of the involved nodes having to be differently implemented. This is particularly useful for robotic systems with limited computational power, where not all possible pipelines can be executed at the same time. In Chapter 8 we will demonstrate how to model different robotic systems using AFG alone.

Chapter 8

High Level Robot Control Using Activity Flow

In this chapter we show how to model different robotic systems using activity flow graphs. In Section 8.1 we review the scenario described in Chapter 4, this time using AFG instead of the basic SDF+. Afterwards, in Section 8.2, we apply the AFG model to the PATSY project, which completes the perception algorithms described in Chapter 6 into a fully autonomous robotic system. Finally, we apply AFGs to an outdoor robot which is first used as a simulated space exploration platform (Section 8.3) and then used as a person-following robot (Section 8.4). All these exemplary use-cases are used to highlight the advantages of a unified and coherent framework.

Parts of this chapter have been published as an article in (Buck and Zell, 2018). Other parts are based on the article in (Huskić, Buck, Ibargüen González, and Zell, 2017a).

8.1 Fetch-and-Delivery Robot using AFG

In Chapter 4 we have presented a robotic fetch-and-delivery system designed for the SICK robot day 2014 competition. The design, based on SDF+, required the use of an FSM to implement the high level mission planning aspects, including different behaviors for fetching and delivering items. We now present a solution using AFG that requires fewer problem specific implementations.

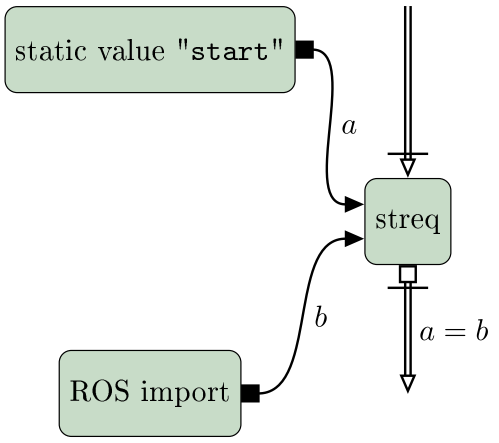
8.1.1 Building Blocks

Except for the FSM, we reuse all the sub-graphs presented in Chapter 4. These include the detection of cross-hair targets and number signs, the detection and evaluation of items in the robot's basket, as well as the analysis of the environmental map. Underlying systems, such as navigation and mapping, are also kept. This leaves the implementation of additional functional units to replace the FSM.

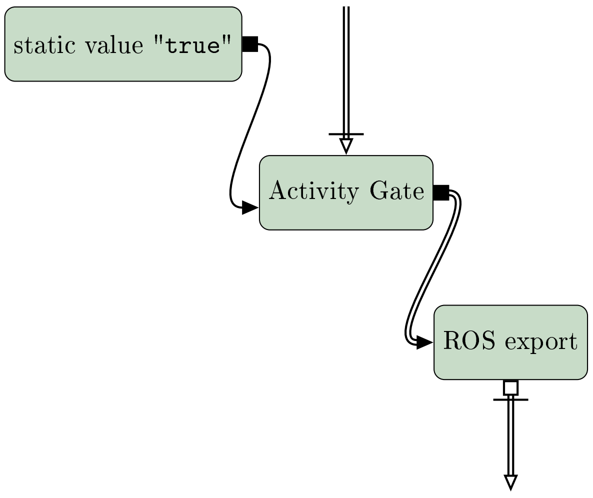
Waiting for a start signal and controlling the signalling light

In the SDF+ model, we require an explicit state that remains active until a starting signal is sent to the robot. With AFG, we do not need an explicit state to wait for a message, as was already demonstrated in Figure 7.4. Instead, we only need a source node that reads on a data channel. Every produced token is then compared to the predefined starting command, and activity is transferred once the incoming token matches. We do not need any problem specific implementation, everything can be achieved using the sub-graph shown in Figure 8.1a.

Similarly, controlling the signaling lamp is achieved by sending a command via an outgoing ROS data channel. As shown in Figure 8.1b, we need to employ an activity gate. The gate only forwards messages when it is active, otherwise it sends a $\perp$ token. This way we ensure that only one message is sent and with this message the activity will be transferred to the exporting node. Once the message is exported, the node triggers an event which will further forward the activity.



(a) The start signal is a string message received on a ROS topic. The *streq* node triggers a signal, when a and b are equal. Only then will activity be transferred.



(b) Setting the signal light is done via a boolean ROS topic. Since the activity gate only forwards messages when it is activated, it forwards exactly one (active) message.

Figure 8.1: Two simple sub-graphs to replace explicit states in an FSM. Activity is only transferred via doubly drawn lines.

In both cases we do not need any application specific nodes, compared to the two specific states necessary in the FSM used in Chapter 4. Rather, every node in the active sub-graphs shown in Figure 8.1 can be interpreted as its own state.

Searching for a delivery station

To deliver an item, the robot has to know the location of the delivery stations. A Gaussian Mixture Model is used to track the positions of all known stations. The map may, however, not yet contain the required station, which needs to be handled robustly. This is done using the graph shown in Figure 8.2: At first the cube detection node becomes active. Once a cube is detected, the imprinted number is sent to a node called *MapLookup*, which also takes the current state of the sign map. The output of the sign map node is a function that maps the number read from a cube to a pose in the world, whereas unknown stations are represented by $\perp$. The *MapLookup* node emits a signal if the lookup fails, i.e. when the result is $\perp$. Otherwise it forwards the result of applying the function to the key. This way, the activity flow branches into two different flows, depending on whether the delivery station is known or not.

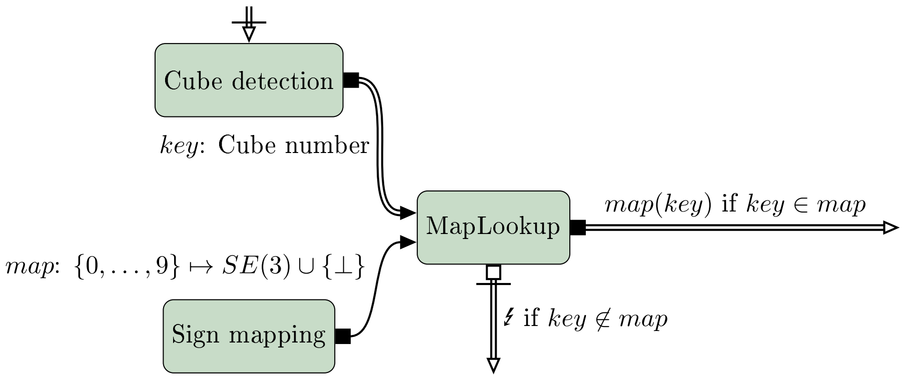

Cube detection
key: Cube number
MapLookup
$map(key)$ if $key \in map$
map: $\{0, \ldots, 9\} \mapsto SE(3) \cup \{\perp\}$
Sign mapping
$\lightning$ if $key \notin map$

Figure 8.2: The sign map node produces a map of known sign poses. Unknown signs are mapped to $\perp$. The cube detection node sends the number printed on detected cubes. The activity flow is routed depending on whether the sign pose is known or not.

In case the station is known, the pose resulting from the map lookup is used as the goal pose for the navigation system. Otherwise, the robot switches into an exploration mode, where it drives counter-clockwise around the arena, examining the outside wall for possible stations. This is again implemented similarly to the FSM case, where a problem specific node is implemented to generate goal poses for the navigation stack, based on the current pose of the robot and the results of map analysis.

Positioning the robot reactively

Positioning the robot is done analogously to the FSM implementation: To navigate the arena we employ a full navigation stack, including path planning and following. Additional movements are done reactively, i.e. not using global positioning but relative measurements. For this, the reactive programming approach achieved by the asynchronous data flow of AFG is a perfect match.

In the original design, the robot first quickly moved about 2 m diagonally forward to populate the occupancy grid map with enough measurements to extract the central filling stations. This center exploration is its own state in the SDF+ model and can be replaced with a more generalized AFG node *MoveRobot*. When activated, the node *MoveRobot* generates motion commands that move the robot for a given distance in a predefined direction. Additionally, the FSM has a state to back up the robot by about 1 m after collecting or delivering an item, for which we can immediately reuse the *MoveRobot* node.

In order to precisely place the robot below either a filling or a delivery station, another node called *PositionToTarget* is implemented. The new node is essentially equivalent to the state implemented for the FSM im Chapter 4. Whereas the FSM state handled the complete control of the robot, the AFG node is implemented in a more functional way: As input it takes the pose of the robot, the detected target signs and the estimated pose of the target sign, and as output it generates a motion command. The AFG approach has the advantage that additional processing nodes can be used before and after the generation of the movement command. For example, a Bayesian filter can be used to improve the localization accuracy of the target without changing the implementation of *PositionToTarget*. Another possibility would be an obstacle avoidance node that post-processes the generated movement command to avoid collisions.

After the robot is positioned relative to the target sign, it is commanded to drive towards the wall in front of it until a critical distance is reached. This is achieved using a general motion node *MoveToObstacle*, which is also mostly equivalent to the FSM case. *MoveToObstacle* has as input the current laser scanner measurement and as output a motion command. Again, the advantage of the AFG approach is that we can arbitrarily pre-process the input data using a data flow graph, e.g. filtering the laser scan. With a single framework it is also easier to find good parameters to tweak the performance of the robot, especially when using a graphical user interface.

Hierarchical Graph

We combine multiple nodes into a sub-graph node to reduce complexity and to make it easier to reuse existing solutions. Keeping to the model implemented in the FSM case, we combine selecting a filling station, navigating to the station, positioning to the target, and evaluating the cube into a *Fetch Cube* node. Similarly, we group the sub-graphs shown in Figure 8.1 into *Wait for Start* and *Explore*, respectively.

Delivering a collected cube is also grouped into *Deliver Cube*, which strongly resembles the *Fetch Cube* node. The only difference between the two is the determination of the target pose and whether to wait for a cube number or for a $\perp$ token at the output of the cube detection node.

8.1.2 Resulting AFG Graph

Combining the individual building blocks, we get the AFG shown in Figure 8.3, which completely solves the challenge without the use of any additional high level control structure.

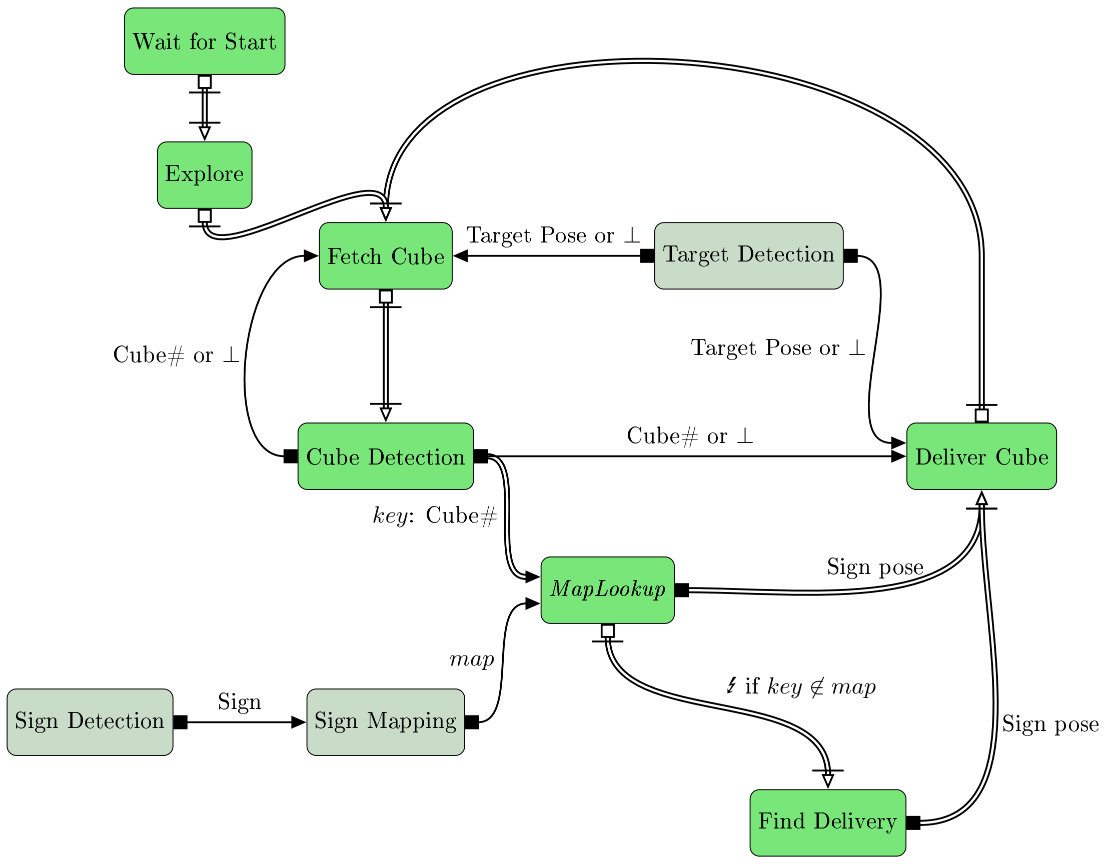


Figure 8.3: A mostly complete activity flow graph that solves the SICK robot day 2014 challenge. The highlighted nodes form the active sub-graph, which also includes the cube detection, one of the perception graphs shown earlier. Sensor source nodes and command sink nodes are not shown to reduce complexity.

8.2 Freely Navigating AGV

As a second example, we give an AFG model that realizes the same architecture for the PATSY AGV as seen in Section 6.3. Figure 8.4 shows the part of the model responsible for the collection of a payload container. In Chapter 6 we did not go into detail about the navigational aspects, as well as high level control, because we need AFG features to implement them.

First of all, we need to implement additional nodes, which are highlighted in Figure 8.4. The central node for high level control is the *Mission Control* node, which is responsible for making decisions about what the AGV should do. We only show the sub-graph necessary for the collection of payload containers, which is initiated by a signal $\lightning_{get\ container}$. Once this signal is sent, *Mission Control* loses activity while *Path Planner* gains it.

Path Planner and *Path Follower* are two generic navigation nodes that can also be used with other robots. While the path planner plans a path to a given input pose, the node stays active. Only when a path is published is the activity transferred to *Follow Path*. There are also signals for error handling which transfer activity. However, they are not shown here in order to reduce the complexity.

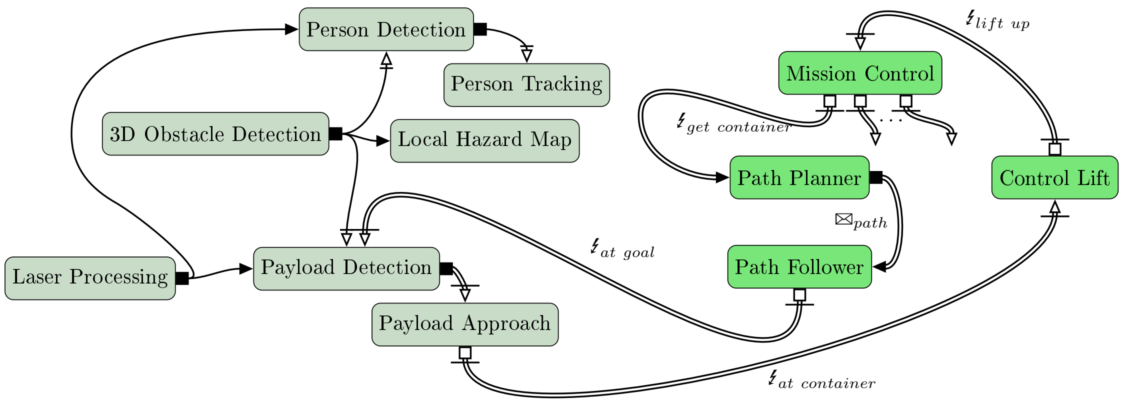


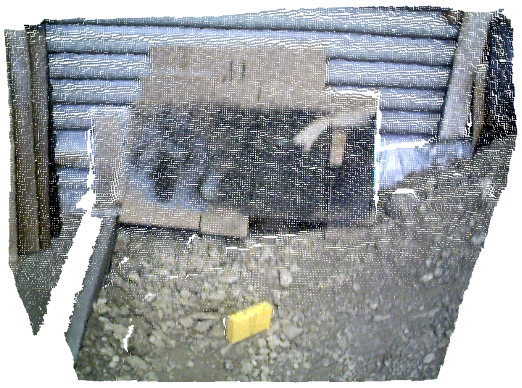
Figure 8.4: AFG model for the PATSY architecture that also includes high level control. The *Mission Control* node can select between different tasks by emitting signals. When it emits $\lightning_{get\ container}$, the payload approach is eventually used to collect a container.

When the path has been followed to the end, the signal $\lightning_{at\ goal}$ transfers the activity to the payload detection node. The node stays active until the next payload pose is successfully estimated and published to the node responsible for approaching the payload. When the container is reached, another signal $\lightning_{at\ container}$ is emitted, which activates *Control Lift*. This special node, which controls the lifting unit of the AGV, is used to pick up the payload cart. The node stays active until the lift is completely in the upward position and then forwards activity back to *Mission Control*.

The active sub-graph in Figure 8.4 strongly resembles an ordinary FSM, however it overlaps with the pure data flow graph. At least two more states would have to be implemented in an FSM, which correspond to the overlapping nodes. An additional advantage is the fact that we only have a single graph, which speeds up development and debugging, as well as allowing the use of a common graphical user interface, which we will present in Chapter 9.

8.3 Planet Exploration Robot

To illustrate the re-usability benefits of AFG, we briefly present a part of our high level design for the SpaceBot Camp 2015. Here the outdoor robot introduced in Chapter 1 was used to simulate a planetary exploration mission. One of the simulated tasks was to detect the object shown in Figure 8.5a and map its pose (cf. Figure 8.5b.)


(a) One of the objects to be found.

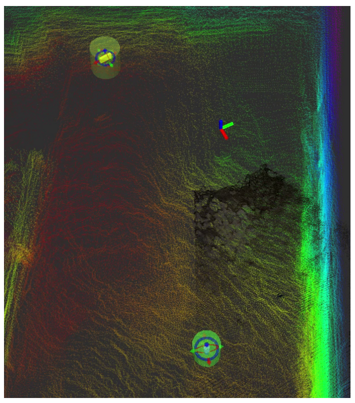

(b) A map showing two detected objects and the robot's current pose.

Figure 8.5: The object detection task in the SpaceBot Camp.

A similar model to the one presented in Section 8.2 is shown in Figure 8.6, whereas differences are highlighted. Many modules are completely reused, such as the generic navigation nodes and the obstacle detection and mapping.

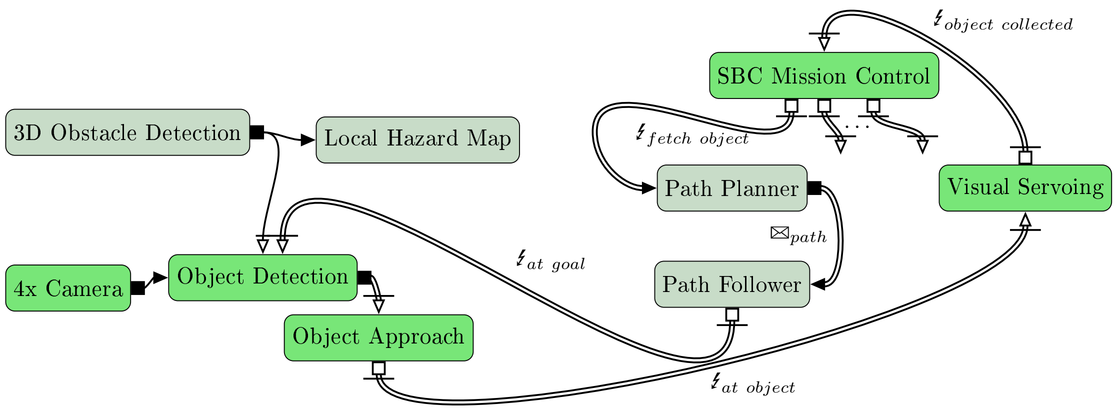


Figure 8.6: The AFG for the SpaceBot Camp is in part very similar to the PATSY graph.

8.4 Human-following Outdoor Robot

As a final example, we model a high level control system for the Summit XL robot with a stereo camera setup. The resulting robotic system is capable of following a person at higher speeds in outdoor environments, as is visualized in Figure 8.7.

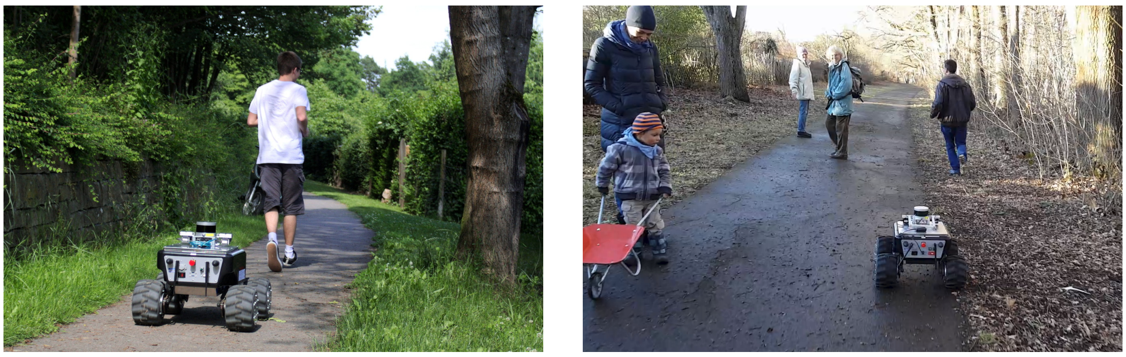

(a) An example scenario. (b) Challenging situations are handled.

Figure 8.7: The robot's task is to accompany a person at higher speeds in outdoor environments. High level control is needed due to unknown environments and obstacles.

8.4.1 Challenges

Person following is well understood. Simple, reactive controllers can achieve acceptable results at lower speeds. The maximum velocities for these algorithms range from $0.75\,\mathrm{m\,s^{-1}}$ in (Chen and Birchfield, 2007) and $1\,\mathrm{m\,s^{-1}}$ in (Kobilarov *et al.*, 2006) to $1.2\,\mathrm{m\,s^{-1}}$ in (Bohlmann, Beck-Greinwald, Buck, Marks, and Zell, 2012). At higher speeds, deliberate methods based on kinematic or dynamic path planning are needed to implement safe and stable following behavior. This is especially the case for skid-steered vehicles, such as the Summit XL used as the basis for our system, which quickly gets unstable at higher velocities.

Our system combines multiple novel approaches that form a robust navigation solution in any environment. The perception pipeline is based on the PATSY project and does not pose any requirements on the environment. Dynamic obstacles are explicitly handled and there is no need for a planarity assumption. Using the kinematic controller published by Huskić, Buck, and Zell (2017b), our robot is able to follow a runner in uneven terrain with a maximum velocity of $2.5\,\mathrm{m\,s^{-1}}$.

8.4.2 Perception

We use a single AFG to model perception and high level control. The sub-graph responsible for perception is shown in Figure 8.8.

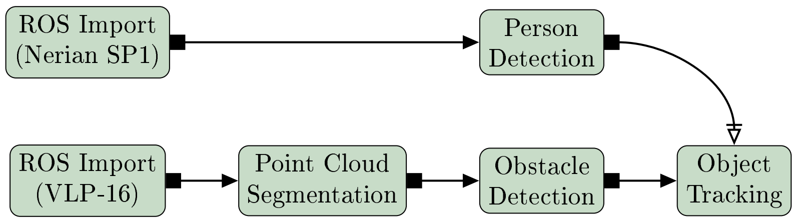


Figure 8.8: Obstacles are detected and tracked using 3D LIDAR data at the acquisition rate of the sensor. Person detection uses a stereo camera. Detected persons are asynchronously handled by the tracking.

We use a 3D Velodyne VLP-16 LIDAR to detect obstacles in the environment and to track the person. The data is first imported using a generic ROS importing node. Every scan is then segmented using a variant of the generic obstacle detection algorithm described in Chapter 6. Detected obstacles are then clustered and tracked to distinguish dynamic from static obstacles. The person detection pipeline from Chapter 6 is used with a Nerian SP1 stereo camera to further distinguish persons from other dynamic obstacles.

The result of the perception part of the graph is visualized in Figure 8.9, where detected obstacles are shown overlaid on the LIDAR scan. The currently tracked person is visualized as an arrow, where the length of the arrow depicts the person's velocity.

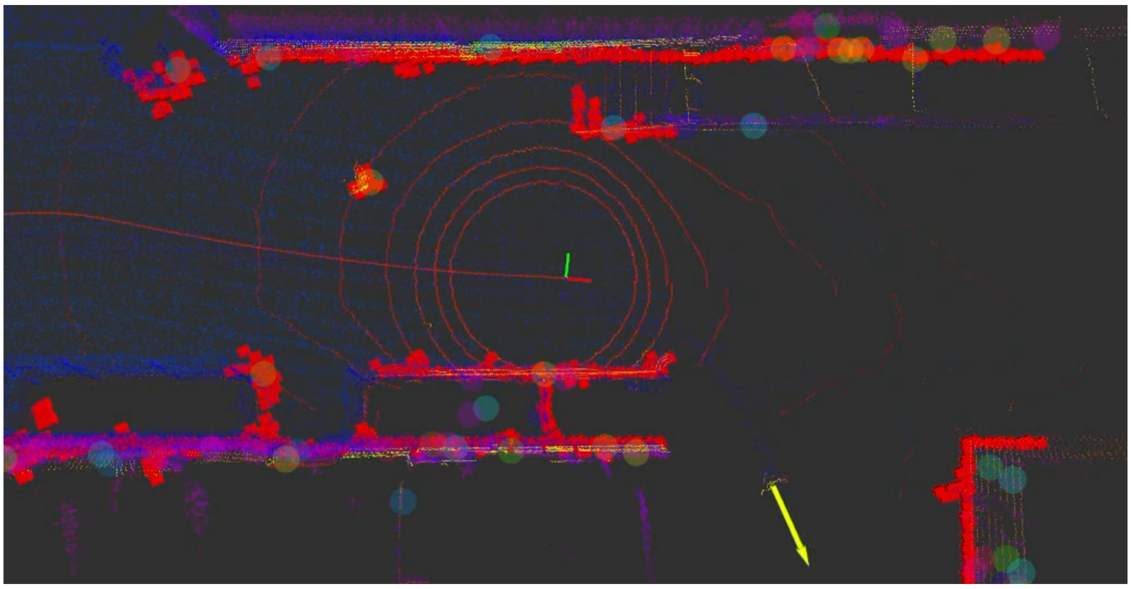
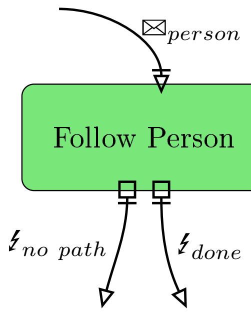


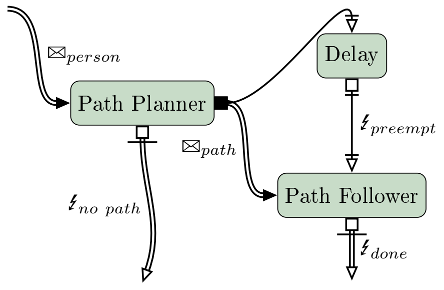
Figure 8.9: Perceived obstacles in the environment (red) and the person's velocity (arrow).

8.4.3 High Level Control

Figure 8.10 shows the central graph component needed to model the high level control of this robot. The *Follow Person* graph takes a person's location and tries to navigate the robot there.



(a) The nested graph takes a person and either emits $\lightning_{done}$ or $\lightning_{no\ path}$ depending on whether a path was found.



(b) Implementation of the nested graph: A path to the person is planned. If a path was found, the follower controls the robot until the $\lightning_{preempt}$ token is received.

Figure 8.10: The *Follow Person* tries to navigate the robot towards the person, when it gets activated. A *Delay* node is used to stop the path following after a few seconds.

The implementation of the graph (cf. Figure 8.10b) uses a predefined navigation strategy to try to reach the person. If a path can be found using this strategy, the path is sent to the *Path Follower* node, which steers the robot along the path. In addition, a *Delay* node receives the same path and emits a $\lightning_{preempt}$ token after a few seconds. This token stops the path following process and returns the activity to the outer graph via $\lightning_{done}$. This way, *Follow Person* steers the robot for a predefined amount of time towards the person.

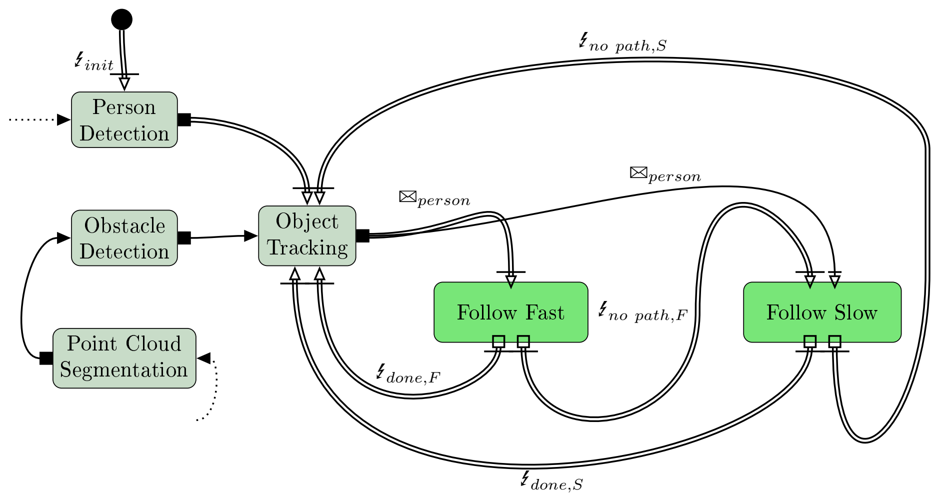


Figure 8.11: Using two instances of *Follow Person* we achieve a simple high level controller, which can recover from navigation errors.

By combining differently parametrized instances of *Follow Person*, we can handle challenging situations. Figure 8.11 shows a graph using two instances: *Follow Fast* and *Follow Slow*. Initially, the *Person Detection* node is activated. The activity is then transferred to the *Object Tracking* node with the first message produced by *Person Detection*. When a person is being tracked by the *Object Tracking* node, the fast following node gets activated via the $\boxtimes_{person}$ token. While it is activated, the robot follows a path towards the person that only contains forward motion. This way the robot can drive at its maximum speed without braking.

When *Follow Fast* is done, either the end of the planned path is reached or a time-out was signaled. The activity then flows back to the object tracking via $\lightning_{done,F}$. This way, we achieve a circular flow of control that regularly activates the *Follow Fast* node. In case no forward path can be found in *Follow Fast*, the $\lightning_{no\ path,F}$ token transfers the activity to *Follow Slow*, another instance of *Follow Person*. In this state the navigation is configured to slower velocities and the path planner is allowed to plan both forward and backward motion. This way, the robot can robustly follow a person at the maximum speed possible. It will only slow down if no forward path can be found, which rarely happens in real experiments. One such experiment is shown in Figure 8.12, where the robot followed a person for ca. 1.7 km.

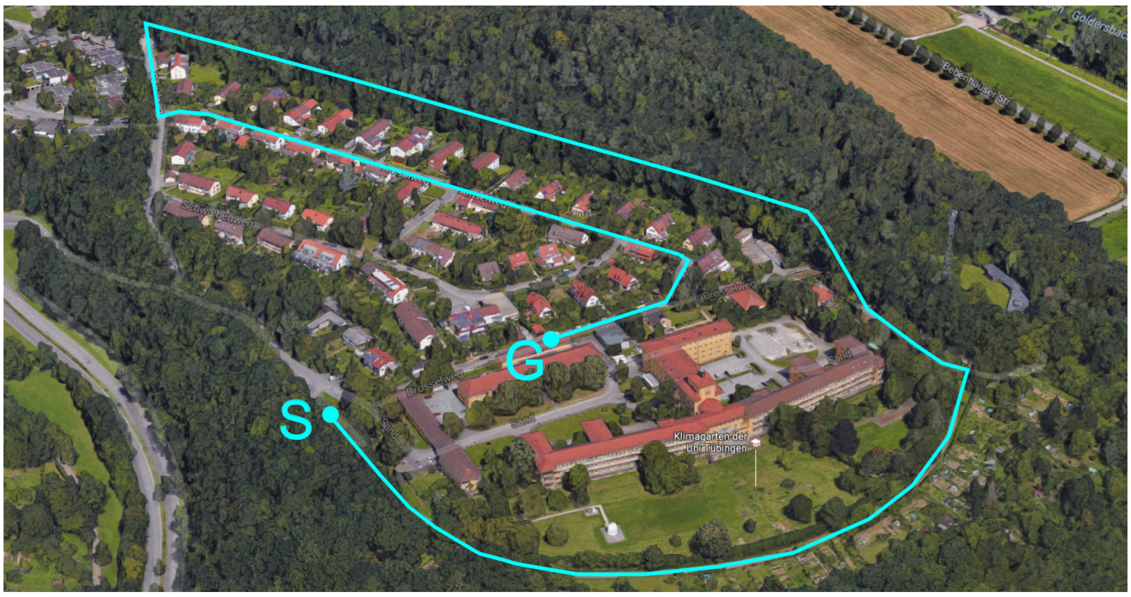


Figure 8.12: Trajectory of the robot following a runner for ca. 1.7 km.

Using AFG, only a single computation graph is needed to implement this complex behavior. Figure 8.13 displays a screen shot of the complete activity flow graph in CS::APEX. Among smaller tasks, this graph models obstacle detection, person tracking, pose estimation and high level mission planning.

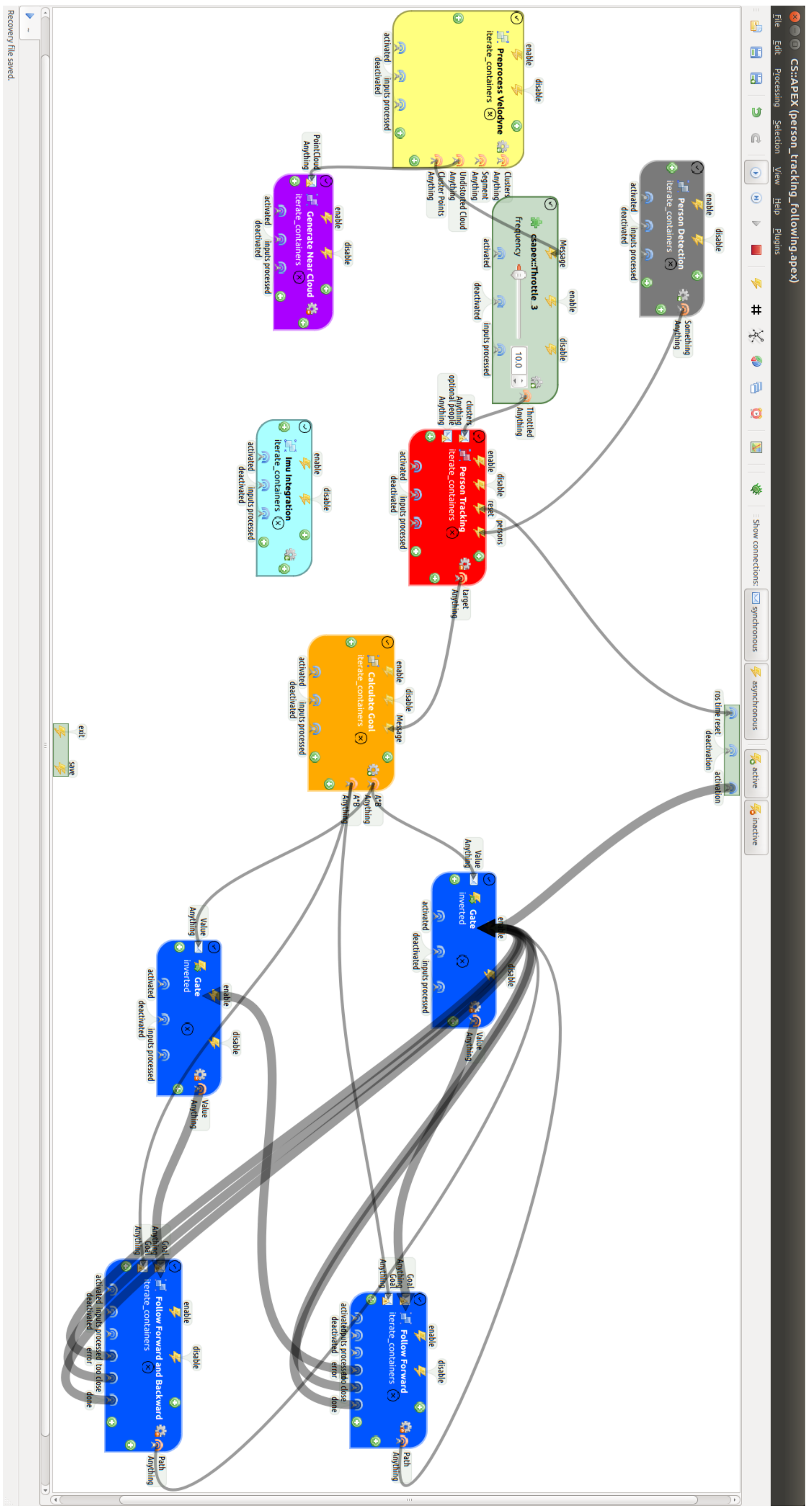


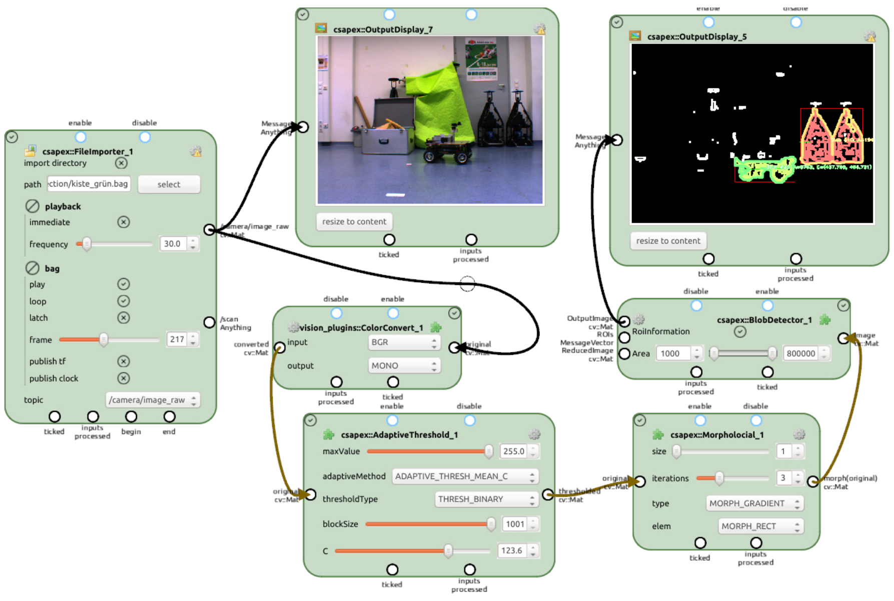
Figure 8.13: Screen shot of a fully functional, hierarchical AFG graph in CS::APEX. This graph models obstacle detection (yellow), person tracking (red), pose estimation (turquoise) and high level mission planning (blue). The (optional) person detection is shown in gray. Detected persons are asynchronous inputs to the person tracking, which can also be used as pure dynamic object tracking. The orange node projects the tracked person's future position as input for the navigation system. Active edges are visualized thicker than normal edges.

Chapter 9

Prototyping Robotic Algorithms Using Visual Programming

9.1 Cognitive Systems Algorithm Prototyper and EXperimenter

In previous chapters we have described activity flow graphs in detail and we have shown multiple exemplary applications. All these robotic systems have been developed using a common framework, called the *Cognitive Systems Algorithm Prototyper and EXperimenter* (CS::APEX), which consists of a graphical user interface and an execution back-end. The user interface (shown in Figure 9.1) allows direct interaction with the structure of the data flow graph and introspection into the flow of data. This enables rapid prototyping of algorithms and also encourages reusable code and modular designs.


csapex::FileImporter_1
import directory
path ction/kiste_grün.bag
playback
immediate
frequency 30.0
bag
play
loop
latch
frame 217
publish tf
publish clock
topic /camera/image_raw

csapex::OutputDisplay_7

csapex::OutputDisplay_5

vision_plugins::ColorConvert_1
input BGR
output MONO

csapex::AdaptiveThreshold_1
maxValue 255.0
adaptiveMethod ADAPTIVE_THRESH_MEAN_C
thresholdType THRESH_BINARY
blockSize 1001
C 123.6

csapex::Morpholocial_1
size 1
iterations 3
type MORPH_GRADIENT
elem MORPH_RECT

csapex::BlobDetector_1
RoiInformation
Area 1000 800000

Figure 9.1: Exemplary workflow that imports a ROS bag, converts the camera image to gray values, performs adaptive thresholding, morphological operations and blob detection.

In this chapter we briefly describe how CS::APEX implements the activity flow graph hierarchy. Our implementation of AFG is tailored to allow interactive data flow graph manipulation. We also show how computation parameters are explicitly represented in the user interface and introduce aspects of control flow. Finally, we show how the scheduling of such graphs can be controlled by the user.

The following requirements were the foundation for the development of the framework:

- Data flow graphs can be created and modified at runtime.
- Parameter values can be changed through the data flow and by the user.
- Data in the flow can be inspected at any step in the process.
- The user can influence the scheduling policy of the processing nodes.
- Irregular events are made visible to the user and can be handled coherently.
- The flow of activity can be controlled by the user.

This chapter is based on the previously published article (Buck *et al.*, 2016b).

9.2 Implementation of AFGs in CS::APEX

The aim of CS::APEX is to provide a user-centerd platform for developing and experimenting with flow-based algorithms for robots and other cognitive systems, encouraging modularity, extensibility and accessibility. We follow a pragmatic approach to data flow programming, focused on providing a user-friendly interface, concentrating on speeding up the prototyping experience, providing useful user feedback and making parameters of the system more easily accessible. Resulting data flow networks can be directly deployed on a robot (Section 9.3).

The framework consists of two components: A graphical user interface (see Figure 9.1) based on Qt5 and a computation back-end library for scheduling and maintenance[1]. We achieve modularity by implementing the flow-based graph structure defined by the AFG model hierarchy, presented in Chapter 3, Chapter 5 and Chapter 7. Flow graphs automatically encourage users to implement component-based solutions that only depend on message types and can thus be easily reused. Extensibility is accomplished by a plug-in system, which makes modification of the main components unnecessary and simplifies the distribution of implemented computing nodes among collaborators.

The user interface allows the user to dynamically add and delete computation nodes at runtime. Nodes can also be disabled and enabled, moved and copied. Furthermore, the user can add and delete connections between nodes and inspect the transmitted values. No scripting or manual configuration file editing is required. The user interface is used

[1]An overview video can be seen at `http://youtu.be/weFZZrQ1BeE`

to generate a network and to provide feedback during the prototyping process. Once the configuration is done, the UI is no longer needed and the graph can be executed in a headless fashion. This way, a prototype configuration can be used on a robot, without a screen attached.

9.2.1 Nodes and Messages

Adding custom functionality is possible by implementing new node types and providing parameters to allow fine tuning. Computation nodes are written in C++11 and dynamically linked once they are needed. We provide multiple ways to add new processing nodes: Nodes can be derived from a base class `Node`, or from specialized base classes, like image filters. Furthermore, we provide a utility class that can automatically generate nodes from a given C++ function by analyzing the function signature using template meta programming techniques.

When a new node is needed, three functions have to be implemented: `setup` tells the system about the required input and output ports, as well as event triggers and slots. The function `setupParameters` declares the parameters of the new node. Every parameter will automatically be wrapped into a UI widget so that the user can manipulate it easily (cf. Figure 9.2). Finally, `process` implements the new functionality, i. e. messages from inputs are read and processed, before output messages are generated and published on the outputs. Asynchronous processing of slots is done using callback functions specified in the setup stage.

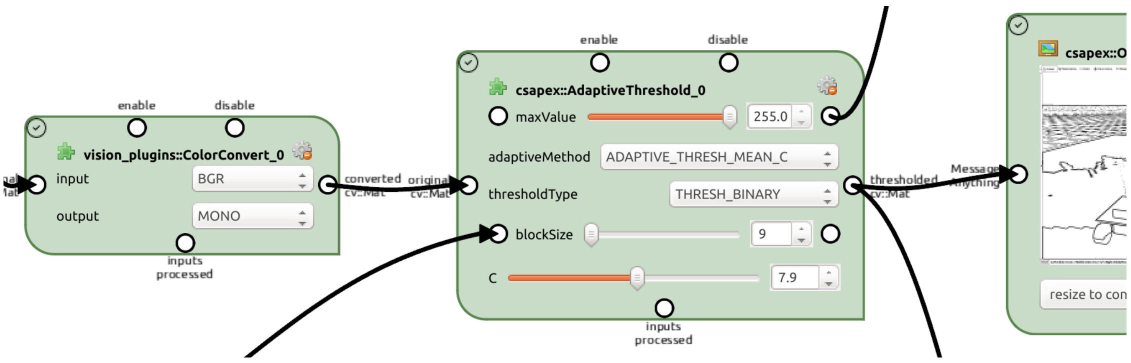
Nodes are communicating by sending tokens, which contain data of specific message types. The tokens can easily be copied and distributed among nodes without copying of the attached message data. Packages can also provide custom message types. Other plug-ins can use these messages without having to worry how to do I/O with them. Authors of a custom message can provide functions to serialize the message using YAML, to read them from a file, to visualize them and publish them via ROS. We have implemented messages for integral types, strings, images, laser scans and more. Among others, we have also implemented support for OpenCV[2] and the point cloud library (PCL)[3] via independent libraries.

9.2.2 Parameters

Parameters are central to our approach, since the direct feedback from changing parameters can speed up rapid prototyping. As described in Section 3.2, we create a pair of one input and one output per parameter. Since this would lead to a lot of ports for nodes with many parameters, we only add inputs and outputs for parameters that are marked as *interactive* by the user.

[2]OpenCV is available at `http://www.opencv.org/`

[3]PCL is available at `http://www.pointclouds.org/`



input BGR
output MONO
maxValue 255.0
adaptiveMethod ADAPTIVE_THRESH_MEAN_C
thresholdType THRESH_BINARY
blockSize 9
C 7.9

Figure 9.2: A node that performs adaptive thresholding on an image with one image input and one output. There are five parameters: two floating point ranges (`maxValue`, `C`), one integer range (`blockSize`) and two sets (`adaptiveMethod`, `thresholdType`). `maxValue` and `blockSize` are connected to the data flow. The node has automatically generated slots to enable and to disable it, as well as an event that fires once the input is processed.

We provide different types of parameters: Boolean, integral and floating point values, ranges, intervals, pairs and more specialized variants like angles, file paths and color values. To enable fast changes to parameter values, we wrap each parameter into a specialized UI widget that allows the user to quickly modify the parameter's value. UI elements and the parameters themselves are strictly separated, so that a graph can be used on a robot completely without a graphical interface. We render these controls directly into the graphical representation of the node, as can be seen in Figure 9.2. This way, the controls are physically located where they are used, which makes it easy to find the right parameter to change and allows us to provide helpful information for parameters via tool-tips.

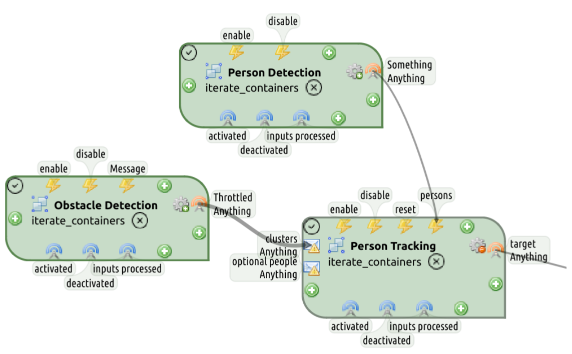
9.2.3 Hybrid Flow

In Chapter 5 we introduced hierarchical computation graphs, which are fully supported in the graphical user interface. Groups of nodes can be combined into a nested sub-graph, whereby existing connections will be automatically transformed to use graph-global ports. Sub-graphs can optionally be used to iterate the entries of list-like messages one at a time, which can be used to implement a *map-reduce* approach (Dean and Ghemawat, 2008).

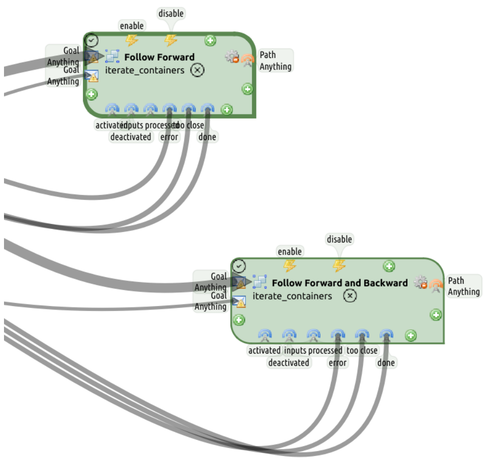
Hybrid connections are also implemented, which means that any output port can be connected to any input port. The only constraint is that synchronous connections may not result in a cycle. The user can, for example, connect a synchronous output o to an asynchronous slot, which causes the slot to be triggered for every produced message from o (cf. Figure 9.3a).

9.2.4 Activity

Active nodes are highlighted in the interface (cf. Figure 9.3b), which gives feedback about the current state of the system. To control the activity of the system, the user can mark connections to be active, as defined in Chapter 7. Marking a connection e_k as active corresponds to setting $A(e_k) \leftarrow 1$ (Equation 7.4). Furthermore, the activity can be reset to to initial state without resetting the whole system, which is especially beneficial during prototyping and debugging.



(a) The node *Person Tracking* uses both synchronous and asynchronous inputs: *Person Detection* sends messages using a hybrid, asynchronous connection. *Obstacle Detection* sends messages using a synchronous connection.



(b) Active nodes are highlighted with a bold frame.

Figure 9.3: The user interface allows the use of hybrid connections and activity.

9.2.5 Scheduling and concurrency

Executing a node does not have any side effects on other nodes in the graph. For this reason, concurrency can be achieved without any effort on the client side, merely the scheduler has to deal with the details of concurrent programming. This takes away the burden on inexperienced programmers, who can make full use of parallel architectures with thread-agnostic modules. Parallel execution is not necessary though, all nodes can be executed sequentially as well.

Even though users should not have to be concerned about matters of concurrency, we want to enable them to take control of the scheduling process. For this reason, we employ a distributed scheduling scheme using thread pools. Every node is assigned to a thread group and all the nodes in a thread group are managed by one scheduler. By default, every node is managed by a single scheduler, as to not fully utilize every CPU core, if that is not desired. The interface allows users to define their own thread groups and assign nodes to them.

9.3 Applications

Since the development of CS::APEX began in 2012, we have developed more than 500 plug-ins to solve a variety of perception problems for research projects and robotics competitions: We achieved the second place in the SICK Robot Day 2014, which was described in Chapter 4. We also deployed the framework for the perceptions tasks at the SpaceBot Camp 2015[4], which was hosted by the national aeronautics and space research center of Germany (DLR). Additionally, we are using the framework in research projects: As shown in Chapter 6, we developed a person-recognizing autonomous transportation system in the BMBF-founded project PATSY, using data flow graphs to detect objects and people in point clouds. In the project IZST IOC 104[5], founded by the state of Baden-Württemberg, we developed vision algorithms for laparoscopic surgery. The framework is also used by many students to simplify the development of robotic prototypes. Some examples have been shown in previous chapters in Figure 3.24, Figure 5.9 and Figure 8.13.

9.3.1 Evolutionary optimization ROS nodes

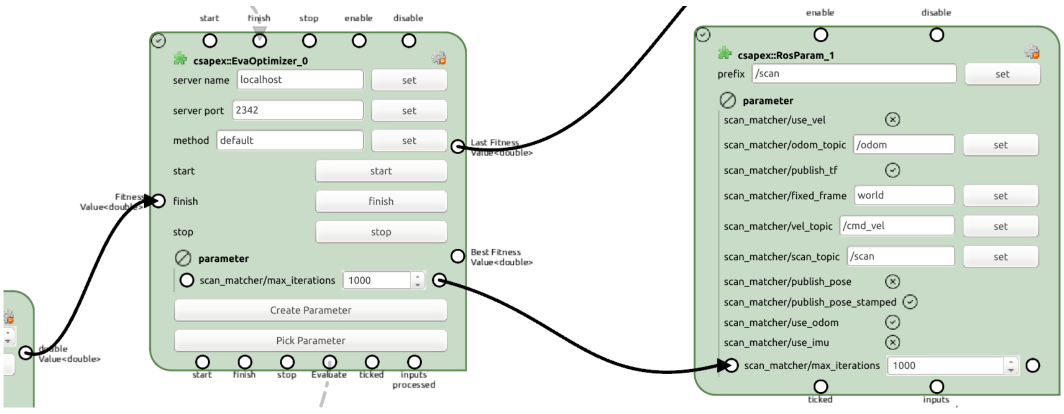
In Chapter 6 we said that we used evolutionary optimization to select values for all free parameters of the obstacle detection. Here we demonstrate this optimization process in CS::APEX using a simplified problem: The application of evolutionary algorithms to optimize the parameters of a ROS node called `laser_scan_matcher` using the *Differential Evolution* algorithm (Figure 9.4). Using a plug-in to communicate with the optimization framework Eva2 developed by Kronfeld *et al.* (2010), we are able to find parameters that minimized the trajectory error on our dataset.

The scan matching process is running in a separate ROS node, whose parameters are set using a generic ROS parameter interface, that can be seen in Figure 9.4. The values of these parameters are determined by the optimization framework for each iteration and are propagated through the data flow. Events and slots are used to asynchronously control the optimization process: An event is executed every time a run of the scan matcher was complete. This event is connected to a slot in the optimizer, which caused a new parameter set to be generated.

The communication with the optimization framework is generic and can be applied to other problems. The user selects an arbitrary set of parameters using the user interface, which adds it to the optimization process. This is possible due to the generic integration of the parameters into the data flow and can be implemented purely on a plug-in basis, since it does not require any modifications to the framework. The only additional functionality, that has to be implemented, is a node to calculate a fitness value for the current parameter set. This value is needed by the optimizer to assess different parameter combinations.

Furthermore, this example demonstrates the usefulness of the additional event-based

[4]`http://www.dlr.de/dlr/desktopdefault.aspx/tabid-10081/151_read-15747`
[5]`http://www.ra.cs.uni-tuebingen.de/forschung/Chirurgische_Navigation`



server name localhost
server port 2342
method default
scan_matcher/max_iterations 1000

prefix /scan
scan_matcher/use_vel
scan_matcher/odom_topic /odom
scan_matcher/publish_tf
scan_matcher/fixed_frame world
scan_matcher/vel_topic /cmd_vel
scan_matcher/scan_topic /scan
scan_matcher/publish_pose
scan_matcher/publish_pose_stamped
scan_matcher/use_odom
scan_matcher/use_imu
scan_matcher/max_iterations 1000

Figure 9.4: Evolutionary optimization of a ROS node for localization. A node for calculating a fitness value has to be implemented. For each newly generated population, the `Evaluate` event is fired. A `finish` slot is triggered when the current population of parameters is fully evaluated and the fitness is finalized. Eva2's and ROS parameter setter's parameters are connected. (To simplify we only show one connection.)

mechanism. Events can be used to control the execution of different data flow sub-graphs. This way, complex processes like these can be automated and controlled based on the data flow. To the best of our knowledge, such an approach to parameter optimization is not possible in related frameworks.

9.3.2 Rapid Prototyping and Experimentation

The main motivation for the CS::APEX graphical user interface is to aid the user in rapid prototyping of new algorithms. As an example we show some of the available functionality in Figure 9.5. The depicted graph is taken from the SICK robot day model, which was presented in Chapter 4. This graph, which was created after the competition, combines all other graphs presented before in a single instance.

First of all, the example demonstrates the profiling options available. Five nodes are profiled and for each of them a bar plot of the past execution durations is shown. On the bottom of the window there is an additional profiling panel that shows the execution start and end time points for all profiled nodes. This enables rapid prototyping since the user can directly observe the effects of parameter changes on the performance of any node and on the complete system.

The second feature shown is data inspection. The user highlights a connection with the mouse cursor, which opens a second window that shows the data currently sent via this connection. Many message types that are available can be visualized this way.

The final feature shown is the assignment of nodes to thread groups, which is here additionally indicated by the color of the nodes. This way nodes from different groups can be executed at the same time, as can be observed in the profiling panel.

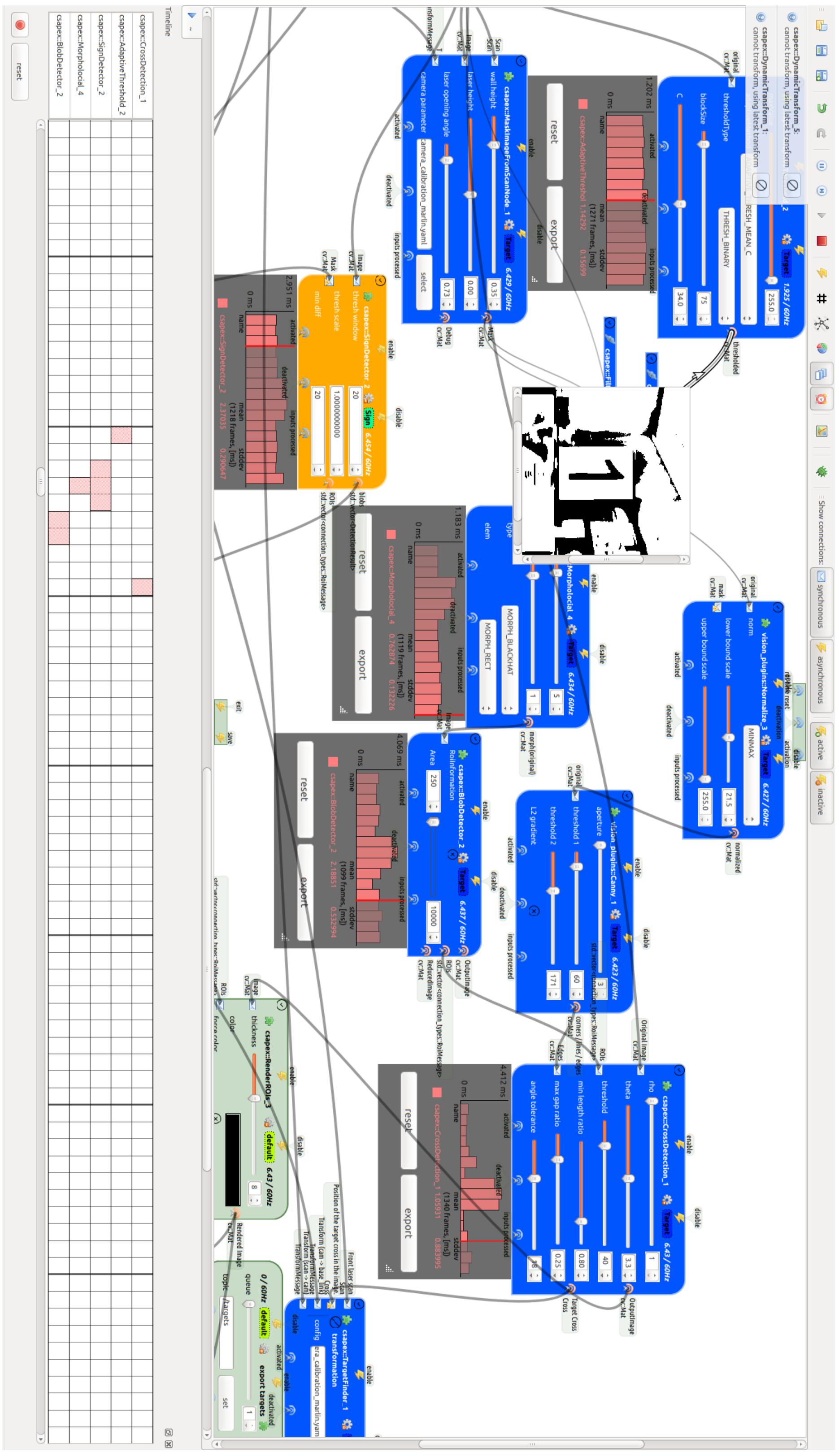


Figure 9.5: Screen shot of some of the prototyping and profiling capabilities in CS::APEX. The user can inspect the data flow at any point, here the output of a morphological operator node is shown. Also shown are different thread groups that are allocated and the average rate of execution for each node. Here five nodes are being profiled, next to each of them we can see a bar plot of the durations of the last executions. A global profiling plot on the bottom of the screen gives more detailed information, accurate to 1 ms. In this example, two of the nodes are executed in parallel, indicated by overlapping red boxes.

9.4 Comparison to Related Work

Many tools and frameworks utilizing flow-based programming have been published in related domains, such as Ptolemy II by Eker *et al.* (2003) and the Ptolemy-based Kepler by Ludäscher *et al.* (2006). Special purpose frameworks include the Robot Task Commander by Hart *et al.* (2014), the Konstanz Information Miner (Knime) (Berthold *et al.*, 2007) for data mining, the Waikato Environment for Knowledge Analysis (WEKA , Hall *et al.* (2009)) and Orange (Demšar *et al.*, 2013) for machine learning and MeVisLab (Bitter *et al.*, 2007) for medical image processing. There are also commercial products based on data flow processing, for example LabVIEW and MATLAB Simulink. A cognitive architecture for artificial vision is described by Chella *et al.* (1997) using a more symbolic approach than the one presented here, whereas the approach presented by in Hochgeschwender *et al.* (2014) is purely declarative.

Biggs *et al.* (2011) provide a pipeline based approach, and show its potential with an example for point cloud processing. They impose requirements similar to the ones presented here, yet focus more on inter-process communication aspects and less on interaction. Their framework implements purely asynchronous data flow, meaning that there is a need for message queues between components. They provide a user interface with which the graph can be modified at runtime and parameters can be adjusted, but they do not model parameters in the data flow and don't seem to feature event-based functionality in their user interface.

Although our implementation can be used independently from ROS (Quigley *et al.*, 2009), we support ROS interaction, such as data import and export. We provide our user interface as a single ROS node, in which ROS topics can be subscribed and published to. ROS itself can be seen as a flow-based framework, where different nodes are separate processes and can run on different machines in a local network. ROS is implemented using the publish-subscribe pattern, where each node is publishing messages onto topics that are subscribed to by other nodes, which is another case of asynchronous data flow. Our visual programming approach allows users to construct processing graphs on the fly via a graphical user interface instead of using text based configuration files. Our implementation can be compared to ROS nodelets, in that all processing nodes are running in the same process, yet ROS nodelets do not allow any interaction or scheduling control.

More relevant for our work is ecto (Ethan Rublee *et al.*, 2015), which grew out of the ROS scene and also represents computer vision and perception tasks as a directed acyclic graph. In contrast to ecto, our approach explicitly handles node parameters, allowing them to be used as data sources or sinks. The ecto framework also provides some form of event-handling, yet these are not part of the interface of a processing node as in our proposed solution. Graphs in ecto are meant to be specified and configured using python programs. There exists a web-based graphical user interface for ecto which allows users to create nodes and connect them. This approach has the advantage that the graph is accessible via the network, yet the implementation does not allow for a high level of interactivity. In our approach, the user interface is the key part of the framework and we focus on interactive graph manipulation and immediate feedback.

Another, recently published framework is RAFCON by Brunner *et al.* (2016), which similarly allows developers to graphically design complex robotic systems. Their approach is based on hierarchical finite-state machines with asynchronous data flow. The framework is more suitable for developing high-level mission control than general data flow algorithms.

9.5 Discussion

In other fields, graphical prototyping tools have become commonplace, yet in robotics there do not exist such standard tools. We think this is partly due to the fact that robotics is a broad and multidisciplinary field. We aim to provide a user-friendly graphical interface that lowers the barrier to entry into robotics, especially robotic perception. The interface provides immediate feedback, allowing to visually construct new data flow graphs at runtime, to get insight into the data flow in real-time and to learn the effects of different parameters on the behavior of the algorithm. Although we provide a library of reusable nodes, custom algorithms have to be implemented eventually. In contrast to fine-grained visual programming, where individual instructions are composed in a graphical interface, we rely on a plug-in based system which allows users to program custom processing nodes in C++ and then compose them visually. This approach simplifies using unknown modules, since inputs, outputs and parameters are directly displayed and can be connected visually.

Many perception problems naturally show synchronous characteristics: If an online image classification algorithm, for example, cannot operate in real-time, images have to be dropped, synchronizing the update rate of the pipeline. On its own, however, homogeneous synchronous data flow is a limiting factor, due to a uniform execution of the individual nodes, whereas robotic systems are composed of many subsystems requiring different update rates. By extending SDF+ into HFG we can solve this problem with only little additional complexity.

Our framework can be fully applied to problems that can be separated into modules. Highly optimized algorithms that cannot be subdivided, however, have to be implemented as single nodes. This is not unusual in a coarse-grain data flow framework such as the presented approach. Even though large nodes are less reusable, they can still profit from the parameter system and other UI features such as execution profiling and data visualization.

Chapter 10
Conclusions

10.1 Summary

The main idea of this dissertation was to develop a coherent data flow framework that can be used to implement any robotic system. We focused on robot perception and high level robot control, though other components, such as localization and navigation, are also realizable The graphical framework was derived in three stages, all of which are useful on their own.

The lowest layer model, which we called SDF+, is based on synchronous data flow, reactive programming and finite-state machines, and was introduced in Chapter 3. This model consists of multiple, disjoint concepts that can be used to model any robot, as we have shown for a fetch-and-delivery robot in Chapter 4. The main disadvantage of this model is its complexity and variety: Synchronous data flow lends itself to modeling perception pipelines, reactive programming is suitable for event-based algorithms. Additionally, finite-state machines are needed for high level robot control and mission planning. These different concepts cannot easily be represented in a single, coherent framework.

The first extension to SDF+ was the addition of a hierarchical graph structure and hybrid connections between the synchronous and asynchronous aspects in Chapter 5. This resulted in the definition of *Hybrid Flow Graphs* (HFG), which unify the synchronous data flow and the reactive programming based ideas of SDF+. These extensions give a more precise model, allowing a seamless blending of synchronous and event-based data processing. The application of the model was demonstrated in Chapter 6, where we have implemented the complete perception sub-system of an AGV using HFG.

We observed that using HFG, we still need to employ two disjoint graphical structures to model complex robotic behavior: hybrid data flow graphs and finite-state machines. FSMs are needed, because data flow graphs are inherently state-less, yet complex behavior requires a form of high level state management. In Chapter 7 we have transferred the concept of activity from FSMs into HFG, which can be used to eliminate the need for FSMs altogether. The resulting *Activity Flow Graphs* (AFG) model is a coherent way to model a complete robotic system without needing any additional frameworks. In many cases, AFG models require fewer states than equivalent HFG models with a state machine. Additionally, the activity concept is unobtrusive to the implementation of individual nodes, allowing the use of any HFG node in an AFG graph. These advantages were demonstrated in Chapter 8, where we have presented multiple AFG models of different, high level robotic systems.

The three models form a hierarchy, where SDF+ is a real subset of HFG, which in turn is superseded by AFG. This means that an implementation of AFG can be used in a restricted way to achieve the other two models. In Chapter 9 we have presented our open-source implementation of the AFG model, which is called CS::APEX. The implementation consists of a graphical user interface and a separate computation back-end. Developers can quickly and easily generate AFG graphs, experiment with different graphs and settings, and debug existing solutions using a single user interface. Resulting graphs can be directly used on any robot that supports the robot middleware ROS.

10.2 Discussion

An important aspect of any complex system is an architecture that is well defined and that allows all collaborating users to develop their required functionality as easily as possible. Developers in a team have different programming styles and different levels of experience, but their implemented functionality should nevertheless be as reusable and interchangeable as possible in order to maximize efficiency. This can best be achieved by a consensus on common interfaces between modules. The computation graph abstraction, which is the foundation of the AFG hierarchy, is very promising in this aspect. The interfaces of individual nodes in the graph are defined by the types of messages the nodes read and write. With the explicit introduction of synchronous and asynchronous input ports in our approach, this interface description is even more precise.

The AFG model was developed with two main goals: On one hand it serves as a reference for the real-world implementation as a theoretical model based on the well-known Petri net formulation. On the other hand it is explicitly designed to be easily adapted into a graphical user interface, which enables rapid prototyping and intuitive optimization, profiling and debugging. The distinction between the two types of data flow intuitively communicates two different meanings to users. Synchronous inputs are processed at once and are therefore very similar to a regular function call with multiple arguments. The synchronous inputs can only be processed again once all processing is finished down-stream. This can also be interpreted in a functional way, since the messages sent by a node are arguments to other functions, and so on. The functional analogy is very natural and useful in understanding complex graphs. Slots, i.e. asynchronous inputs, are processed individually. This gives them the character of an event- or interrupt-handler, which are common in many programming environments. They are expected to perform quickly and can be called in fast succession.

Given any graph constructed using AFG, the model exactly determines the behavior of the system. It is easy to see for a given node which other nodes have to be executed first, while parallel execution of unrelated nodes is automatically possible. During the implementation of individual nodes the user does not have to consider any of these factors. Each node can be implemented completely thread agnostic and is still safe to use in parallel with other nodes. This will become more important as CPUs are providing an ever growing amount of available processing cores.

Another important aspect of large, complex systems is modularity. Individual modules should have as few dependencies on each other as possible. This means that only the modules needed to solve the task at hand have to be available. Using AFG we naturally achieve modularity, due to the foundation on pure message passing interfaces. Dependencies exist only on the types of messages sent between nodes and not on the nodes themselves. Up to this point we have implemented more than 500 individual processing node types and more than 20 messages types. Due to this modularity the framework has successfully been used in many projects, competitions and student theses.

Another design goal for AFG was to provide a coherent approach, which means that every aspect of the computation graph should be the same for all types of nodes. This is where the lower level models (SDF+ and HFG) fall short, because they still require the use of a finite-state machine in order to implement higher level state. It is possible to use the lower level models to implement the perception part and a separate FSM to model high level control, where both graphs can communicate with each other. We have demonstrated this once with SDF+ and then with HFG. However, this requires the use of two completely different concepts.

Of course, we could implement a graphical user interface that combines the construction of hybrid flow graphs and finite-state machines, but then we would still need to precisely specify the ways of interaction between the two. The AFG approach instead generalizes the relevant aspects of finite-state machines and combines them with hybrid flow graphs. This combined approach is at first not as intuitive as the well-known finite-state machines, because the separation between state and data flow is lost. However, the approach also has multiple advantages: First of all, there is no need to explicitly implement different states in the system. Every node can be directly used as a state that is active for the duration of the node's execution. This is a common case for FSMs, where we switch between states after specific computations are performed. Second, the interactions between "states" and normal nodes are formally defined and behave deterministically. Using the Petri net model we can precisely understand the interaction between the collaborating nodes. Third, since there is no distinction between different types of nodes, the model becomes simpler and easier to adapt into a graphical user interface with which users can manipulate these graphs. Lastly, we can use a common scheduling algorithm for all nodes, which ensures that resources are properly managed.

There are two reasons why we chose to use a dynamic scheduler in the implementation of CS::APEX . First, it is possible to adapt the execution of larger graphs with potential for concurrent processing at run-time. Depending on the available resources, scheduling algorithms are capable of reacting to the current system load. Second, while static scheduling is more efficient in principal, it is not possible to statically schedule asynchronous data flow and so only the synchronous flow could be scheduled offline. This would introduce complications, since synchronous and asynchronous data handling in the same node must not happen at the same time.

There is a small overhead of a few percent in execution performance due to the dynamic scheduling. This is not a problem in the development and testing phase, but using AFG models in performance-critical systems that operate at the limit of the hardware is not advised. Instead the framework is meant for rapid prototyping and experimentation. It even allows users with little programming experience to implement algorithms quickly, to improve their performance by inspecting the data flow and profiling the execution, and then deploy the resulting program on any robot easily.

10.3 Future Work

We have implemented a few robotic systems using AFG, which serves to demonstrate the validity of the model. In future work we intend to apply the model to even more systems with different requirements and tasks.

One interesting opportunity is modeling swarms of robots, each having their own AFG model. This requires inter-robot communication, which can be implemented using AFG alone. However, an extension of the model to explicitly manage unstable connections between nodes would be beneficial. Furthermore, there are other aspects of the original synchronous data flow as described by Lee *et al.* (1987) that could be transferred into AFG. This includes recursive synchronous connections and inhomogeneous processing nodes.

Another possible extension is to explicitly model time in the flow graph, allowing the formulation of constraints to guide the execution of more complex graphs. This would take AFGs closer to real-time systems, even on regular architectures. Additionally, time could be introduced into the control of the activity of nodes, to realize time-out behaviors on a model level, instead of in the implementation of individual nodes.

Future releases of CS::APEX could include further features to increase the productivity of developers: The core implementation should be implemented using a client-server architecture, such that the user interface can be used on a different machine from the actual robot. This would greatly improve inspection and debugging tasks.

Nomenclature

ACF	Aggregate Channel Features, Page 63
AFG	Activity Flow Graph, Page 96
AGV	Automated Guided Vehicle, Page 1
AMCL	Adaptive Monte Carlo Localization, Page 60
CS::APEX	Algorithm Prototyper and Experimenter for Cognitive Systems, Page 4
DP	Dataflow Programming, Page 11
EA	Evolutionary Algorithm, Page 32
FBP	Flow-based Programming, Page 11
FPDW	Fastest Person Detector in the West, Page 63
FSM	Finite-State Machine, Page 15
HFG	Hybrid Flow Graph, Page 51
HOG	Histogram of Oriented Gradients, Page 63
LIDAR	Light Detection And Ranging, Page 5
MHT	Multi-Hypothesis Tracker, Page 85
MLP	Multi-Layer Perceptron, Page 84
PCL	Point Cloud Library, Page 115
RF	Random Forest, Page 84
ROC	Receiver Operating Characteristic, Page 87
ROI	Region of Interest, Page 84
ROS	Robot Operating System, Page 12
SDF	Synchronous Data Flow, Page 13
SLAM	Simultaneous Localisation and Mapping, Page 41
SVM	Support Vector Machine, Page 84
TOF	Time-of-Flight, Page 7
UML	Unified Modeling Language, Page 15
YAML	YAML Ain't Markup Language, Page 115

Bibliography

Alenyà, G., Foix, S., and Torras, C. (2014). Tof cameras for active vision in robotics. *Sensors and Actuators A: Physical*, **218**, 10–22.

Arras, K. O., Grzonka, S., Luber, M., and Burgard, W. (2008). Efficient people tracking in laser range data using a multi-hypothesis leg-tracker with adaptive occlusion probabilities. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 1710–1715. IEEE.

Badino, H., Franke, U., and Pfeiffer, D. (2009). The stixel world-a compact medium level representation of the 3d-world. In *Pattern Recognition*, pages 51–60. Springer.

Behnke, S. (2006). Robot competitions-ideal benchmarks for robotics research. In *Proc. of IROS-2006 Workshop on Benchmarks in Robotics Research*.

Berthold, M. R., Cebron, N., Dill, F., Gabriel, T. R., Kötter, T., Meinl, T., Ohl, P., Sieb, C., Thiel, K., and Wiswedel, B. (2007). KNIME: The Konstanz Information Miner. In *Studies in Classification, Data Analysis, and Knowledge Organization (GfKL 2007)*. Springer.

Bhatia, S. and Chalup, S. K. (2013). Segmenting salient objects in 3d point clouds of indoor scenes using geodesic distances. *Journal of Signal and Information Processing*, **4**(03), 102.

Biggs, G., Ando, N., and Kotoku, T. (2011). Rapid data processing pipeline development using openrtm-aist. In *System Integration (SII), 2011 IEEE/SICE International Symposium on*, pages 312–317.

Bishop, C. M. *et al.* (2006). *Pattern recognition and machine learning*, volume 1. Springer New York.

Bitter, I., Van Uitert, R., Wolf, I., Ibanez, L., and Kuhnigk, J.-M. (2007). Comparison of four freely available frameworks for image processing and visualization that use itk. *Visualization and Computer Graphics, IEEE Transactions on*, **13**(3), 483–493.

Bohlmann, K., Beck-Greinwald, A., Buck, S., Marks, H., and Zell, A. (2012). Autonomous person following with 3d lidar in outdoor environments. In *1st International Workshop on Perception for Mobile Robots Autonomy (PEMRA 2012)*, Poznan, Poland.

Bostelman, R., Russo, P., Albus, J., Hong, T., and Madhavan, R. (2006). Applications of a 3d range camera towards healthcare mobility aids. In *Networking, Sensing and Control, 2006. ICNSC'06. Proceedings of the 2006 IEEE International Conference on*, pages 416–421. IEEE.

Brady, M. (1985). Artificial intelligence and robotics. *Artificial Intelligence*, **26**(1), 79–121.

Broggi, A., Cattani, S., Patander, M., Sabbatelli, M., and Zani, P. (2013). A full-3d voxel-based dynamic obstacle detection for urban scenario using stereo vision. In *Intelligent Transportation Systems-(ITSC), 2013 16th International IEEE Conference on*, pages 71–76. IEEE.

Brown, J. (2013). Zbar bar code reader.

Brunner, S. G., Steinmetz, F., Belder, R., and Dömel, A. (2016). Rafcon: A graphical tool for engineering complex, robotic tasks. In *Intelligent Robots and Systems(IROS), 2016 IEEE International Conference on*. IEEE.

Buck, S. and Zell, A. (2018). CS::APEX: A framework for algorithm prototyping and experimentation with robotic systems. *Journal of Intelligent & Robotic Systems*.

Buck, S., Hanten, R., Huskić, G., Rauscher, G., Kloss, A., Leininger, J., Ruff, E., Widmaier, F., and Zell, A. (2015). Conclusions from an object-delivery robotic competition: Sick robot day 2014. In *Advanced Robotics (ICAR), The 17th International Conference on*, pages 137–143, Istanbul, TR.

Buck, S., Hanten, R., Bohlmann, K., and Zell, A. (2016a). Generic 3d obstacle detection for agvs using time-of-flight cameras. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 4119 – 4124, Daejeon, Korea.

Buck, S., Hanten, R., Pech, C. R., and Zell, A. (2016b). Synchronous dataflow and visual programming for prototyping robotic algorithms. In *Intelligent Autonomous Systems (IAS), The 14th International Conference on*, pages 911–923, Shanghai, CN.

Buck, S., Hanten, R., Bohlmann, K., and Zell, A. (2017). Multi-sensor payload detection and acquisition for truck-trailer agvs. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, Singapore.

Canny, J. (1986). A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, **8**(6), 679–698.

Cardarelli, E., Sabattini, L., Secchi, C., and Fantuzzi, C. (2014). Multisensor data fusion for obstacle detection in automated factory logistics. In *Intelligent Computer Communication and Processing (ICCP), 2014 IEEE International Conference on*, pages 221–226. IEEE.

Chella, A., Frixione, M., and Gaglio, S. (1997). A cognitive architecture for artificial vision. *Artificial Intelligence*, **89**(1), 73–111.

Chen, Z. and Birchfield, S. T. (2007). Person following with a mobile robot using binocular feature-based tracking. In *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, pages 815–820. IEEE.

Cigolini, M., Costalunga, A., Parisi, F., Patander, M., Salsi, I., Signifredi, A., Valeriani, D., Lodi Rizzini, D., and Caselli, S. (2013). Lessons learned in a ball fetch-and-carry robotic competition. In *4th International Conference on Robotics in Education 2013*, pages 169–176. Faculty of Electrical, Electronic, Computer and Control Engineering.

Cox, I. J. (1991). Blanche-an experiment in guidance and navigation of an autonomous robot vehicle. *Robotics and Automation, IEEE Transactions on*, **7**(2), 193–204.

Dalal, N. and Triggs, B. (2005). Histograms of oriented gradients for human detection. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1 - Volume 01*, CVPR '05, pages 886–893, Washington, DC, USA. IEEE Computer Society.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, **51**(1), 107–113.

Demšar, J., Curk, T., Erjavec, A., Črt Gorup, Hočevar, T., Milutinovič, M., Možina, M., Polajnar, M., Toplak, M., Starič, A., Štajdohar, M., Umek, L., Žagar, L., Žbontar, J., Žitnik, M., and Zupan, B. (2013). Orange: Data mining toolbox in python. *Journal of Machine Learning Research*, **14**, 2349–2353.

Dollár, P., Appel, R., Belongie, S., and Perona, P. (2014). Fast feature pyramids for object detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Droeschel, D., Holz, D., and Behnke, S. (2010). Probabilistic phase unwrapping for time-of-flight cameras. In *Robotics (ISR), 2010 41st International Symposium on and 2010 6th German Conference on Robotics (ROBOTIK)*, pages 1–7. VDE.

Dumas, M. and Ter Hofstede, A. H. (2001). Uml activity diagrams as a workflow specification language. In *International Conference on the Unified Modeling Language*, pages 76–90. Springer.

Egerstedt, M., Hu, X., and Stotsky, A. (2001). Control of mobile platforms using a virtual vehicle approach. *IEEE Transactions on Automatic Control*, **46**(11), 1777–1782.

Eker, J., Janneck, J. W., Lee, E. A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., and Xiong, Y. (2003). Taming heterogeneity-the ptolemy approach. *Proceedings of the IEEE*, **91**(1), 127–144.

Elseberg, J., Creed, R. T., and Lakaemper, R. (2010). A line segment based system for 2D global mapping. *2010 IEEE International Conference on Robotics and Automation*, pages 3924–3931.

Enzweiler, M., Eigenstetter, A., Schiele, B., and Gavrila, D. M. (2010). Multi-cue pedestrian classification with partial occlusion handling. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 990–997.

Ethan Rublee, V. R. *et al.* (2015). Ecto - A C++/Python Computation Graph Framework.

Evans, J., Krishnamurthy, B., Pong, W., Croston, R., Weiman, C., and Engelberger, G. (1989). Helpmate™: A robotic materials transport system. *Robotics and Autonomous Systems*, **5**(3), 251–256.

Fejfar, P. and Obdržálek, D. (2014). Smart and easy object tracking. *Vˇera Kurková, Lukáš Bajer (Eds.)*, page 28.

Fox, D., Burgard, W., Dellaert, F., and Thrun, S. (1999). Monte carlo localization: Efficient position estimation for mobile robots. *AAAI/IAAI*, **1999**, 343–349.

Frank, M., Plaue, M., Rapp, H., Köthe, U., Jähne, B., and Hamprecht, F. A. (2009). Theoretical and experimental error analysis of continuous-wave time-of-flight range cameras. *Optical Engineering*, **48**(1), 013602–013602.

Fredriksson, S. R. H., Rosendahl, D., and Wernersson, K. H. A. (2007). An autonomous vehicle for a robotday. In *Mekatronikmöte 2007*.

Fung, W., Leung, Y., Chow, M., Liu, Y., Xu, Y., Chan, W., Law, T., Tso, S., and Wang, C. (2003). Development of a hospital service robot for transporting task. In *Robotics, Intelligent Systems and Signal Processing, 2003. Proceedings. 2003 IEEE International Conference on*, volume 1, pages 628–633. IEEE.

Garulli, A., Giannitrapani, A., Rossi, A., and Vicino, A. (2005). Mobile robot slam for line-based environment representation. In *Decision and Control, 2005 and 2005 European Control Conference. CDC-ECC '05. 44th IEEE Conference on*, pages 2041–2046.

Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. (1991). The synchronous data flow programming language lustre. *Proceedings of the IEEE*, **79**(9), 1305–1320.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The weka data mining software: An update. *SIGKDD Explor. Newsl.*, **11**(1), 10–18.

Hanten, R., Buck, S., Otte, S., and Zell, A. (2016). Vector-amcl: Vector based adaptive monte carlo localization for indoor maps. In *Intelligent Autonomous Systems (IAS), The 14th International Conference on*, Shanghai, CN.

Hanten, R., Kuhlmann, P., Buck, S., Otte, S., and Zell, A. (2018). Robust real-time 3d person detection for indoor and outdoor applications. (submitted).

Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Trauring, A., and Trakhtenbrot, M. (1990). Statemate: A working environment for the development of complex reactive systems. *IEEE transactions on software engineering*, **16**(4), 403–414.

Hart, S., Dinh, P., Yamokoski, J., Wightman, B., and Radford, N. (2014). Robot task commander: A framework and ide for robot application development. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 1547–1554.

Hochgeschwender, N., Schneider, S., Voos, H., and Kraetzschmar, G. K. (2014). Declarative specification of robot perception architectures. In *Simulation, Modeling, and Programming for Autonomous Robots*, pages 291–302. Springer.

Huskić, G., Buck, S., and Zell, A. (2016). A simple and efficient path following algorithm for wheeled mobile robots. In *Intelligent Autonomous Systems (IAS), The 14th International Conference on*, Shanghai, CN.

Huskić, G., Buck, S., Ibargüen González, L. A., and Zell, A. (2017a). Outdoor person following at higher speeds using a skid-steered mobile robot. In *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*, Vancouver, Canada.

Huskić, G., Buck, S., and Zell, A. (2017b). Path following control of skid-steered wheeled mobile robots at higher speeds on different terrain types. In *IEEE International Conference on Robotics and Automation (ICRA)*, Singapore.

Jafari, O., Mitzel, D., and Leibe, B. (2014). Real-time rgb-d based people detection and tracking for mobile robots and head-worn cameras. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 5636–5643.

Kanjanawanishkul, K. and Zell, A. (2009). Path following for an omnidirectional mobile robot based on model predictive control. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 3341–3346. IEEE.

Karabegović, I., Karabegović, E., Mahmić, M., and Husak, E. (2015). The application of service robots for logistics in manufacturing processes. *Advances in Production Engineering and Management*, **10**(4).

Kobilarov, M., Sukhatme, G., Hyams, J., and Batavia, P. (2006). People tracking and following with mobile robot using an omnidirectional camera and a laser. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 557–562. IEEE.

Kohlbrecher, S., Meyer, J., Graber, T., Petersen, K., Klingauf, U., and von Stryk, O. (2014). Hector open source modules for autonomous mapping and navigation with rescue robots. In *RoboCup 2013: Robot World Cup XVII*, pages 624–631. Springer.

Kronfeld, M., Planatscher, H., and Zell, A. (2010). The eva2 optimization framework. In *Learning and Intelligent Optimization*, pages 247–250. Springer.

Lai, K., Bo, L., Ren, X., and Fox, D. (2012). Detection-based object labeling in 3d scenes. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 1330–1337. IEEE.

Lee, C.-H., Su, Y.-C., and Chen, L.-G. (2012). An intelligent depth-based obstacle detection system for visually-impaired aid applications. In *Image Analysis for Multimedia Interactive Services (WIAMIS), 2012 13th International Workshop on*, pages 1–4. IEEE.

Lee, E., Messerschmitt, D. G., *et al.* (1987). Synchronous data flow. *Proceedings of the IEEE*, **75**(9), 1235–1245.

Li, X. (2009). *Dribbling Control of an Omnidirectional Soccer Robot*. Ph.D. thesis, Cognitive Science Department, University of Tuebingen.

Liebelt, J. and Schmid, C. (2010). Multi-view object class detection with a 3d geometric model. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 1688–1695. IEEE.

Linán, C. C. (2014). cvBlob.

Liu, L. and Pu, C. (1997). Activity flow: Towards incremental specification and flexible coordination of workflow activities. In *International Conference on Conceptual Modeling*, pages 169–182. Springer.

Luber, M., Spinello, L., and Arras, K. O. (2011). People tracking in rgb-d data with on-line boosted target models. In *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*.

Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E. A., Tao, J., and Zhao, Y. (2006). Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, **18**(10), 1039–1065.

Luo, R., Li, J.-X., and Chen, C.-T. (2008). Indoor localization using line based map for autonomous mobile robot. In *Advanced robotics and Its Social Impacts, 2008. ARSO 2008. IEEE Workshop on*, pages 1–6.

Martinez-Barbera, H. and Herrero-Perez, D. (2010). Development of a flexible agv for flexible manufacturing systems. *Industrial robot: An international journal*, **37**(5), 459–468.

Masselli, A., Hanten, R., and Zell, A. (2013). Robust real-time detection of multiple balls on a mobile robot. In *Mobile Robots (ECMR), 2013 European Conference on*, pages 355–360. IEEE.

Matas, J., Galambos, C., and Kittler, J. (2000). Robust detection of lines using the progressive probabilistic hough transform. *Computer Vision and Image Understanding*, **78**(1), 119–137.

Maček, K., Petrović, I., and Siegwart, R. (2005). A control method for stable and smooth path following of mobile robots. In *Proceedings of the European Conference on Mobile Robots*.

May, S., Droeschel, D., Holz, D., Fuchs, S., Malis, E., Nüchter, A., and Hertzberg, J. (2009). Three-dimensional mapping with time-of-flight cameras. *Journal of Field Robotics*, **26**(11-12), 934–965.

Mojaev, A. and Zell, A. (2004). Tracking control and adaptive local navigation for nonholonomic mobile robot. In *Intelligent Autonomous Systems (IAS-8)*, pages 521–528, Amsterdam, Netherlands. IOS Press.

Morton, R. D. and Olson, E. (2011). Positive and negative obstacle detection using the hld classifier. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 1579–1584. IEEE.

Niechwiadowicz, K. and Khan, Z. (2008). Robot based logistics system for hospitals-survey. In *IDT Workshop on interesting results in computer science and engineering*. Citeseer.

Nüchter, A., Lingemann, K., and Hertzberg, J. (2006). Extracting drivable surfaces in outdoor 6d slam. *VDI BERICHTE*, **1956**, 189.

Otte, S., Krechel, D., and Liwicki, M. (2013). Jannlab neural network framework for java. In P. Perner, editor, *MLDM Posters*, pages 39–46. IBaI Publishing.

Ozkil, A. G., Fan, Z., Dawids, S., Aanes, H., Kristensen, J. K., and Christensen, K. H. (2009). Service robots for hospitals: A case study of transportation tasks in a hospital. In *2009 IEEE International Conference on Automation and Logistics*, pages 289–294. IEEE.

Piotr Dollár, P., Belongie, S., and Perona, P. (2010). The fastest pedestrian detector in the west. In *Proceedings of the British Machine Vision Conference*, pages 68.1–68.11. BMVA Press. doi:10.5244/C.24.68.

Proetzsch, M., Luksch, T., and Berns, K. (2007). The behaviour-based control architecture ib2c for complex robotic systems (extended version).

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5.

Rauscher, G., Dube, D., and Zell, A. (2014). A comparison of 3d sensors for wheeled mobile robots. In *2014 International Conference on Intelligent Autonomous Systems (IAS-13)*, Padova, Italy.

Reynolds, M., Doboš, J., Peel, L., Weyrich, T., and Brostow, G. J. (2011). Capturing time-of-flight data with confidence. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 945–952. IEEE.

Ronzoni, D., Olmi, R., Secchi, C., and Fantuzzi, C. (2011). Agv global localization using indistinguishable artificial landmarks. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 287–292. IEEE.

Rusu, R. B., Marton, Z. C., Blodow, N., Dolha, M., and Beetz, M. (2008). Towards 3d point cloud based object maps for household environments. *Robotics and Autonomous Systems*, **56**(11), 927–941.

Sabattini, L., Cardarelli, E., Digani, V., Secchi, C., Fantuzzi, C., and Fuerstenberg, K. (2015). Advanced sensing and control techniques for multi agv systems in shared industrial environments. In *Emerging Technologies & Factory Automation (ETFA), 2015 IEEE 20th Conference on*, pages 1–7. IEEE.

Sadiq, S., Orlowska, M., Sadiq, W., and Foulger, C. (2004). Data flow and validation in workflow modelling. In *Proceedings of the 15th Australasian database conference-Volume 27*, pages 207–214. Australian Computer Society, Inc.

Salas, J. and Tomasi, C. (2011). People detection using color and depth images. In *Proceedings of the Third Mexican Conference on Pattern Recognition*, MCPR'11, pages 127–135, Berlin, Heidelberg. Springer-Verlag.

Schafer, H., Hach, A., Proetzsch, M., and Berns, K. (2008). 3d obstacle detection and avoidance in vegetated off-road terrain. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 923–928. IEEE.

Scherer, S. A., Dube, D., Komma, P., Masselli, A., and Zell, A. (2011). Robust Real-Time Number Sign Detection on a Mobile Outdoor Robot. In *Proceedings of the 6th European Conference on Mobile Robots (ECMR 2011)*, Örebro, Sweden.

Sermanet, P., Kavukcuoglu, K., Chintala, S., and LeCun, Y. (2013). Pedestrian detection with unsupervised multi-stage feature learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3626–3633.

Sohn, H. and Kim, B. (2008). An Efficient Localization Algorithm Based on Vector Matching for Mobile Robots Using Laser Range Finders. *Journal of Intelligent and Robotic Systems*, **51**(4), 461–488.

Sohn, H. J. and Kim, B. K. (2009). VecSLAM: An Efficient Vector-Based SLAM Algorithm for Indoor Environments. *Journal of Intelligent and Robotic Systems*, **56**(3), 301–318.

Spinello, L. and Arras, K. O. (2011). People detection in rgb-d data. In *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*.

Stimming, C., Krengel, A., Boehning, M., Vatavu, A., Mandici, S., and Nedevschi, S. (2015). Multi-level on-board data fusion for 2d safety enhanced by 3d perception for agvs. In *Intelligent Computer Communication and Processing (ICCP), 2015 IEEE International Conference on*, pages 239–244. IEEE.

Thamer, H., Kost, H., Weimer, D., and Scholz-Reiter, B. (2013). A 3d-robot vision system for automatic unloading of containers. In *2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*, pages 1–7. IEEE.

Thrun, S. (2002). Particle filters in robotics. In *Proceedings of the Eighteenth conference on Uncertainty in artificial intelligence*, pages 511–518. Morgan Kaufmann Publishers Inc.

Thrun, S., Burgard, W., and Fox, D. (2005). *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press.

Toshev, A. and Szegedy, C. (2014). Deeppose: Human pose estimation via deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1653–1660.

Trcka, N., van der Aalst, W., and Sidorova, N. (2008). Analyzing control-flow and data-flow in workflow processes in a unified way. *Computer science report*, (08-31).

Trevor, A. J., Gedikli, S., Rusu, R. B., and Christensen, H. I. (2013). Efficient organized point cloud segmentation with connected components. *Semantic Perception Mapping and Exploration (SPME)*.

Varga, R. and Nedevschi, S. (2014). Vision-based autonomous load handling for automated guided vehicles. In *Intelligent Computer Communication and Processing (ICCP), 2014 IEEE International Conference on*, pages 239–244. IEEE.

Vincent, R., Limketkai, B., and Eriksen, M. (2010). Comparison of indoor robot localization techniques in the absence of gps. *Detection and Sensing of Mines, Explosive Objects, and Obscured Targets XV*, **76641**, 76641Z–76641Z–5.

Viola, P. and Jones, M. (2001). Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–511–I–518 vol.1.

von Hanxleden, R., Duderstadt, B., Motika, C., Smyth, S., Mendler, M., Aguado, J., Mercer, S., and O'Brien, O. (2014). Sccharts: Sequentially constructive statecharts for safety-critical applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 372–383.

Wang, H., Xu, J., Guzman, J. I., Jarvis, R. A., Goh, T., and Chan, C. W. (2001). Real time obstacle detection for agv navigation using multi-baseline stereo. In *Experimental Robotics VII*, pages 561–568. Springer.

Wang, Q., Gong, D., Wang, S., and Lei, Y. (2014). Range clusters based time-of-flight 3d imaging obstacle detection in manifold space. *Optics express*, **22**(8), 8880–8892.

Weichert, F., Skibinski, S., Stenzel, J., Prasse, C., Kamagaew, A., Rudak, B., and Ten Hompel, M. (2013). Automated detection of euro pallet loads by interpreting pmd camera depth images. *Logistics Research*, **6**(2-3), 99–118.

Wu, S., Yu, S., and Chen, W. (2011). An attempt to pedestrian detection in depth images. In *Intelligent Visual Surveillance (IVS), 2011 Third Chinese Conference on*, pages 97–100.

Zhu, J., Wang, Y., Yu, H., Xu, H., and Shi, Y. (2010). Obstacle detection and recognition in natural terrain for field mobile robot navigation. In *Intelligent Control and Automation (WCICA), 2010 8th World Congress on*, pages 6567–6572. IEEE.