UNIVERSITY OF BREMEN
INSTITUTE FOR
ARTIFICIAL INTELLIGENCE

A Framework for Analyzable, Resource-Aware and Self-Optimizing Robot Longterm Autonomy

Sebastian Georg Brunner

Vollständiger Abdruck der vom Fachbereich 3 (Mathematik und Informatik) der Universität Bremen zur Erlangung des akademischen Grades eines

Doktor der Ingenieurwissenschaften (Dr. Ing.)

genehmigten Dissertation.

1. Prüfer:	Prof. Dr. h.c. Michael Beetz, PhD *Universität Bremen*
2. Prüfer:	Prof. Dr. Christian Schlegel *Technische Hochschule Ulm*

Die Dissertation wurde am 19.5.2020 bei der Universität Bremen eingereicht und vor dem Prüfungsausschuss am 23.9.2020 verteidigt und angenommen.

Abstract

There is a high demand for autonomous robots in various application scenarios. National space agencies want to employ autonomous rovers to examine the geological and atmospheric conditions of foreign planets. In industry, robots are taking over tedious and dangerous jobs from humans in order that they can compensate the lack of specialized staff in other areas. Agriculture and household robotics are more application scenarios where autonomous robots will play a vital role.

However, there does not exist any autonomous robot able to operate in real use cases without human supervision or fences yet, except for robots with limited capabilities such as lawn mowers or vacuum cleaners. Next to open hardware challenges, and software limitations in the areas of computer vision, manipulation and navigation, the creation of software architectures for programming autonomous behavior poses one of the biggest challenges. Creating robotic behavior for complex tasks is an open research field and has to combine the creation of reactive behavior with deliberative, goal-directed planning. Robust and efficient solutions that scale to general behavior suitable for a multitude of use cases are required.

To master these challenges I present a sophisticated, well-tested task control framework for analyzable and resource-aware longterm autonomy, which is composed of three main contributions:

- a novel, graphical task control language for creating robotic plans that are able to solve complex tasks in a robust manner
- an analysis framework for identifying robustness and performance bottlenecks in the created robotic plans
- a methodology to autonomously optimize the analyzed plans

Next to specifying the theoretical concepts, they are evaluated in various application scenarios in both the space and the industrial domain. The most prominent space use case was the *Robotic Exploration of Extreme Environments* (ROBEX) Moon-analogue

demonstration mission, in which our *Lightweight Rover Unit (LRU)* robot performed several scientific relevant missions on Mt. Etna, Sicily, Italy. In one of these missions, the LRU deployed several scientific instruments in order to infer the geological composition of the target area and to measure the origin of seismic events.
One major application scenario in the industrial context was the autonomous supply chain maintained by our *Advanced Industrial Mobile Manipulator (AIMM)* robot, which we presented on the international trade fair Automatica in 2018. In the context of this task, AIMM delivered various parts from several storage locations to different workplaces for further processing.

I finally present and discuss *RAFCON* (short for *RMC*[1] *Advanced Flow Control*), the software framework implementing the theoretical concepts. RAFCON enables a developer to create complex robotic behavior using a graphical *Domain Specific Language* (DSL). The language is based on *Hierarchical, Parallel, Finite State Machines with Data Flows (HPFDs)*, a state machine dialect developed in the course of this work. Next to allowing the modeling of the control flow between modular, reusable skills, RAFCON supports the definition of the data flow between robotic actions. This empowers the developer to create behavior with the ability to perform well informed choices based on system and user parameters, which leads to an eminent diagnosability and transparency of robotic behavior. The data provided by the data flows carry parameters and results of critical decisions and can be reliably processed and communicated to other control flow nodes performing more complex decisions. More properties of the language are its clarity, conciseness and its ability to provide deep insights into the contexts of robotic behavior execution. Furthermore, RAFCON offers powerful analysis tools for investigating the efficiency and the robustness of the created behavior. One of the main considerations behind RAFCON was to create a system able to handle the conflicting concerns of expressiveness, analyzability and implementability. The total number of downloads (more than 25 thousand) shows the popularity of the software framework, which we[2] made open source on Jul 5, 2018.

In all evaluated application scenarios, the behavior framework was responsible for orchestrating all high level software modules in such a way that all the required tasks could be executed by the robot platform autonomously. This orchestration includes the creation of reactive and goal-driven behavior enabling the robot to robustly model and manipulate its environment, also in the presence of non-deterministic events.

[1] RMC is the acronym for the *Robot and Mechatronic Center* of the German Aerospace Center
[2] "We" refers to my colleagues Franz Steinmetz and Rico Belder at the RMC and me.

Zusammenfassung

In verschiedenen Anwendungsfeldern besteht ein hoher Bedarf an autonomen Robotern. Nationale Raumfahrtagenturen wollen mit autonomen Rovern die geologischen und atmosphärischen Bedingungen fremder Himmelskörper erkunden. In der Industrie übernehmen Roboter anstrengende und gefährliche Aufgaben des Menschen, um ihm zu ermöglichen sich selbst weiterzubilden und den Mangel an Fachkräften in anderen Bereichen auszugleichen. Auch die Landwirtschaft und der Haushalt sind Anwendungsszenarien, in denen autonome Roboter eine wichtige Rolle spielen werden.

Bisher gibt es jedoch noch keine autonomen Roboter, die in realen Anwendungsfällen ohne menschliche Aufsicht oder künstliche Beschränkungen (z.B. Zäune) arbeiten können, mit Ausnahme von Robotern, die für sehr spezielle Aufgaben gemacht wurden, wie Rasenmäher- oder Staubsauger-Roboter. Neben ungelösten Problemen in der Hard- und Software (v.a. im Bereich der Bildverarbeitung, Manipulation und Navigation), stellt die Erstellung von Softwarearchitekturen zur Programmierung autonomen Verhaltens eine der größten Herausforderungen dar. Die Schaffung von Roboterverhalten für komplexe Aufgaben ist ein offenes Forschungsfeld und muss die Erzeugung von reaktivem Verhalten mit einer zielorientierten Planung verbinden. Robuste und effiziente Lösungen sind gefragt, die mit den Anforderungen komplexer Aufgabenstellungen skalieren und sich für verschiedene Anwendungsfälle eignen.

In dieser Arbeit präsentiere ich ein ausgereiftes, vielfach getestetes Framework für:

- die effiziente, grafische Programmierung komplexen Roboterverhaltens
- das Profiling und die Analyse des erstellten Verhaltens um Verhaltensfehler und Leistungsengpässe zu identifizieren
- die autonome Optimierung des analysierten Verhaltens

Neben der Spezifizierung der theoretischen Konzepte werden diese in verschiedenen Anwendungsszenarien sowohl im Raumfahrt- als auch im Industriebereich evaluiert.

Der prominenteste Anwendungsfall aus dem Raumfahrtbereich war die Mond-Analog Mission des ROBEX Projektes, bei der unser *Lightweight Rover Unit (LRU)* Roboter mehrere wissenschaftlich relevante Missionen auf dem Ätna in Sizilien, Italien, durchführte. Bei dieser Mission setzte die LRU mehrere wissenschaftliche Instrumente ein, um Informationen über die geologische Zusammensetzung des Zielgebiets zu gewinnen und den Ursprungsort von seismischen Aktivitäten zu bestimmen.
Ein wichtiges Anwendungsszenario aud dem industriellen Kontext war eine vollständig autonome Versorgungskette, die von unserem *Advanced Industrial Mobile Manipulator (AIMM)* Roboter aufrecht erhalten wurde und die wir auf der internationalen Fachmesse Automatica 2018 vorgestellt haben. Im Rahmen dieser Aufgabe lieferte AIMM verschiedene Bauteile aus mehreren Lagerorten zur Weiterverarbeitung an verschiedene Arbeitsplätze.

Zum Schluss stelle ich *RAFCON* (*RMC*[3] *Advanced Flow Control*) vor, das Software-Framework in welchem ich meine theoretischen Konzepte umgesetzt habe. RAFCON ermöglicht es dem Entwickler komplexes Roboterverhalten mit Hilfe einer grafischen, domänenspezifischen Sprache zu erstellen. Neben der Modellierung des Kontrollflusses zwischen modularen, wiederverwendbaren Fähigkeiten unterstützt es die Definition des Datenflusses zwischen verschiedenen Roboteraktionen. Dies befähigt den Entwickler Verhalten zu programmieren, welches in der Lage ist wohl überlegte Entscheidungen auf der Grundlage von System- und Benutzerparametern zu treffen, was zu einer hervorragenden Diagnosefähigkeit und Transparenz des Roboterverhaltens führt. Die von den Datenflüssen gelieferten Daten beinhalten Parameter und Ergebnisse kritischer Entscheidungen und können zuverlässig weiterverarbeitet und an andere Kontrollflussknotenpunkte übermittelt werden. Darüber hinaus bietet RAFCON leistungsfähige Analysewerkzeuge zur Untersuchung der Effizienz und der Robustheit des erzeugten Verhaltens. Einer der Hauptgedanken hinter RAFCON war es, ein System zu schaffen, das in der Lage ist mit dem Spannungsfeld zwischen Aussagekraft, Analysierbarkeit und Umsetzbarkeit umzugehen. Die Gesamtzahl der Downloads (mehr als 25 Tausend) zeigt die Beliebtheit unseres am 5. Juli 2018 Open Source veröffentlichten Software-Frameworks.

In den evaluierten Anwendungsszenarien war das Software-Framework dafür verantwortlich, alle Software-Module so zu orchestrieren, dass alle erforderlichen Aufgaben von der Roboterplattform autonom ausgeführt werden konnten. Diese Orchestrierung beinhaltet die Erzeugung von reaktivem und zielorientiertem Verhalten, das den Roboter in die Lage versetzt, seine Umgebung robust zu modellieren und zu manipulieren, auch unter dem Einfluss von nicht-deterministischen Ereignissen.

[3] RMC ist die Abkürzung für *Robot and Mechatronic Center*, ein Forschungsinstituts des Deutschen Zentrums für Luft- und Raumfahrt

Acknowledgments

First and foremost, many thanks go to my supervisor Prof. Michael Beetz, who enabled me to write my thesis at the *Institute for Artificial Intelligence* (IAI) of the University Bremen as an external PhD student. He could always give me some valuable advice for my publications and my thesis, and could provide me with precious related work of various topics in the academic fields of robotics and artificial intelligence. I also want to thank him for his critical questions regarding all major topics of my contributions.

Moreover, I want to thank Dr. Freek Stulp, Dr. Daniel Leidner and Dr. Simon Kriegel for all their organizational support and fruitful discussions about my work. Without their expertise and experience it would have been much more difficult for me to cope with the challenges of writing my thesis and being supervised by an external professor.

Furthermore, I want to thank Dr. Armin Wedler for bringing into life these wonderful and interesting projects ROBEX and ARCHES and delegating me the responsibilities for high level task and mission control of the LRU rovers.

Of course, I also want to thank all my colleagues and co-authors of my papers, especially Franz Steinmetz, Dr. Martin Schuster, Peter Lehner, Andreas Dömel, Sebastian Riedel and Rico Belder.

Finally, I want to thank my parents, my daughter Philomena and especially my wife Stephanie for all their support and love.

October 5, 2020,

Sebastian Brunner

This work has received funding from the Helmholtz Association project alliance ROBEX (HA-304), the Helmholtz Association projects *Autonomous Robotic Networks to Help Modern Societies* (ARCHES)(ZT-0033) and RACELab, the European Commission under contract numbers FP7-ICT-608849-EUROC and H2020-ICT-645403-ROBDREAM, and the Collaborative Research Center EASE (SFB 1320), funded by the German Research Foundation (DFG).

Contents

List of Figures

List of Tables

Chapter one

Introduction

1.1 Motivation

In space science, the main goals for mankind are to investigate the origins of life and to discover new living space for humanity. In order to reach these goals several space missions have already been executed and many more are planned. One type of such missions is the geological exploration of foreign celestial bodies, such as those performed by the *National Aeronautics and Space Administration* (NASA) in which their Opportunity (see Li et al. (2007)) and Curiosity (see Welch et al. (2013)) rovers were deployed. Planetary exploration missions consist of deploying robots to create accurate maps of the surface, to investigate extraterrestrial material, to measure and analyze the geologic composition of a target object, and to collect soil samples for further analysis on Earth.

Next to planetary exploration missions, the setup of scientific instruments such as powerful antennas and satellites allow for the investigation of space as well. Especially LOFAR antennas (see Heald et al. (2018)) are set up to get insights into the formation of the universe during the time shortly after the Big Bang. This is achieved by searching and investigating signals created billions of years ago in different frequency bands of the cosmic radiation and by examining cosmic magnet fields. LOFAR antennas have already been set up on Earth. However, the recorded signals suffer from the distortion created by the Earth's atmosphere. Thus, a future goal is to set up such antenna arrays on foreign celestial bodies to get noise-less signal recordings. An interesting project investigating the technological challenges of such an antenna setup is the ARCHES project (see Helmholtz Center Future Topic Projects (2019)), in which my work plays a central role.

The information gathered by those operations is not only usable to detect new materials and their origination but also essential to find celestial bodies with a high *Earth Similarity Index* (ESI), i.e., planets which are located in the habitable zone in the orbit of their star and are thus offering living conditions similar to Earth. There have been found many planets with a high ESI, one of the closest outside of our solar system being *Proxima Centauri C*. However, traveling a distance of approximately four light years exceeds the current technology level of humanity. Thus, humanity currently focuses on habitable planets inside the solar system. Possible future scenarios are lunar villages, i.e., village on the surface of the Moon as proposed by Foing (2016), or Mars habitats as proposed by Cohen (2015). Setting up such residences requires a vast effort both technologically as well as economically.

As the environmental conditions of space exploration, instrument setup and habitat setup are often too hazardous for mankind, and proper spacecraft obtaining stable living conditions for humans is very expensive, robots are sent to perform such missions. To some extent, some of these missions have already been performed by using real space robots such as NASA's Opportunity (see Landis et al. (2004)) and Curiosity (see Grotzinger et al. (2012)) rovers. This shows that the hardware of these robotic systems is already mature enough to conduct many of such missions and is thus not the constraining part. However, the decision making of most space rovers (see Strategic Missions and Advanced Concepts Office (2017)) is remotely performed by humans, as the robots lack the needed autonomy capabilities to master those tasks efficiently on their own. As a result, the performance of space rovers is greatly reduced. For example, during its fourteen years of operation the Opportunity rover could only travel a distance of 45 km - a stretch many humans can easily cover in one day on foot. The reasons for this are, on the one hand, that each time the commands for just a few hours operation only were sent to the rover before execution was resumed, and that the round-trip communication latency between Mars and Earth is between eight to forty minutes (see Schuster et al. (2017)). On the other hand, the safety of the robot (which can be very expensive and can take years to reach to target celestial body) is of uttermost importance. Thus, only actions are performed that avoid all possible damages or at least reduce their risk to a minimum.

This work focuses on this problem of non-integrated decision-making: it provides concepts to enhance the autonomy capabilities of robots and enables them to execute complex missions self-reliantly in a robust and efficient manner.

Fig. 1.1 shows a possible future scenario of a planetary exploration endeavor, which guides the reader through the whole thesis. It shows various, challenging objectives for future missions that are currently infeasible for space robots using state of the

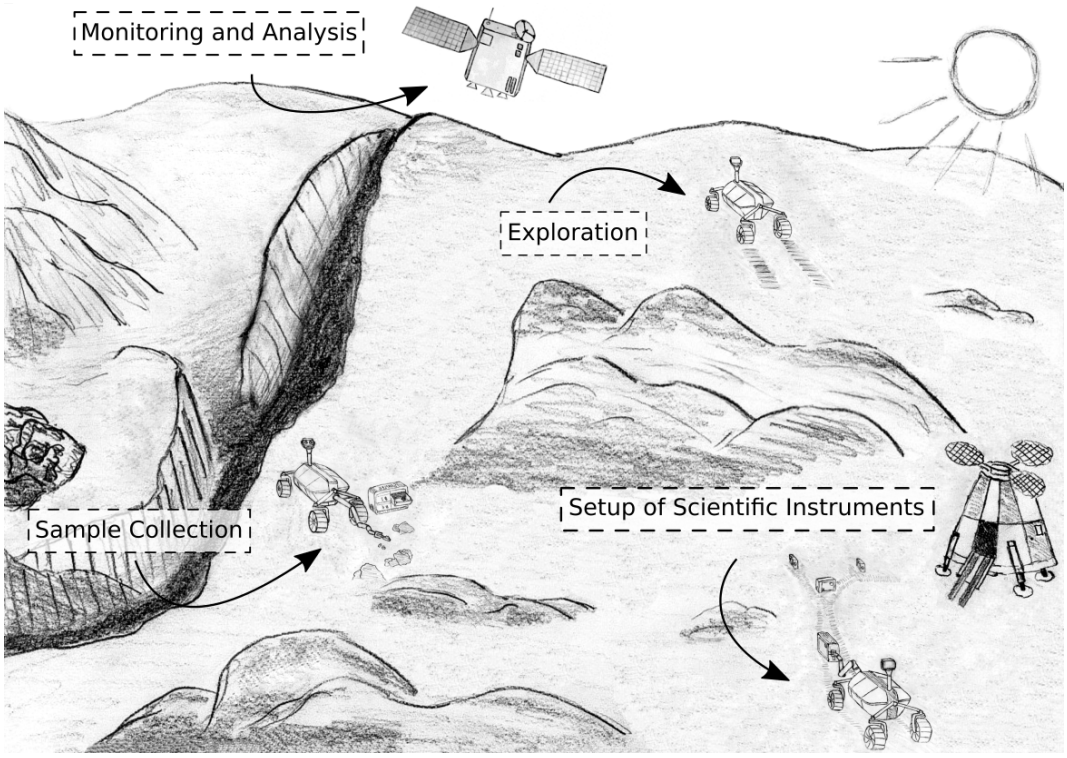


Figure 1.1: *A scene of a possible, future space exploration mission. Several rovers are deployed for different tasks, including exploration and mapping, and the setup of scientific instruments.*

art technology. In the bottom right of the figure a lander is shown, which contains the technical infrastructure for planetary exploration such as rovers and scientific instruments. Some robots already started their tasks: One rover is exploring unknown terrain to decide which areas of the foreign celestial body have to be investigated further with the scientific instruments. Another rover is collecting a soil sample for further investigation of the foreign material directly on board the rover, the lander or (in the context of a sample return mission) on Earth. A third rover is setting up scientific instruments, such as seismometers for inferring the geological composition of the target area, or LOFAR antennas for radio astronomical observation. A manned satellite, which orbits the celestial body, is monitoring and analyzing the progress of the rovers. In case of problems, the astronauts can interfere, guide the robots and thus contribute to the success of the mission. On top of that, gathering precious experience data is another highly important objective in order to be able to optimize the robot's behavior for future endeavors.

As a matter of fact, the described scene is not fictitious but is inspired by the ROBEX Moon-analogue demonstration mission (see Wedler et al. (2017)) and the current ARCHES project (see Helmholtz Center Future Topic Projects (2019)). Some of

Figure 1.2: *The LRU robot on Mt. Etna during the collection of a seismic instrument in front of a lander. In the back, one of the crater is currently erupting, producing a considerable amount of dust and seismic events.*

the proposed tasks have already been executed and evaluated during the analogue-mission. Fig. 1.2, for example, shows the *Lightweight Rover Unit* (LRU) robot (see Sec. 2.5) during the autonomous pick-up of one of several seismic measurement devices currently attached to a lander. After pick-up, the robot placed the seismometers at specific points of interest (POIs) in order to estimate the geologic composition of the target region. Our robot recorded the map needed for the estimation of the POIs completely autonomously during exploration.

My concepts and their implementation in the RAFCON framework were fundamental for realizing this mission. The framework (see Fig. 1.3 for a screenshot) enables to create complex behavior for autonomous agents, enabling them to sense their surrounding, reason about their achieved information, and manipulate their environment in order to fulfill their task objectives. Furthermore, I used RAFCON for behavior execution, mission monitoring and profiling, and online behavior updates during the mission.

In fact, my approach is not only applicable for space scenarios but also for industrial automation in the context of Industry 4.0, service robotics and human-robot interaction. However, the majority of my experiments were performed in the context of space science. This is the reason, why I focus on the space domain (and especially the ROBEX scenario) when highlighting challenges and solutions for robot autonomy. In the evaluation Chpt. 7, I will also present results for other domains, which have similar requirements and challenges.

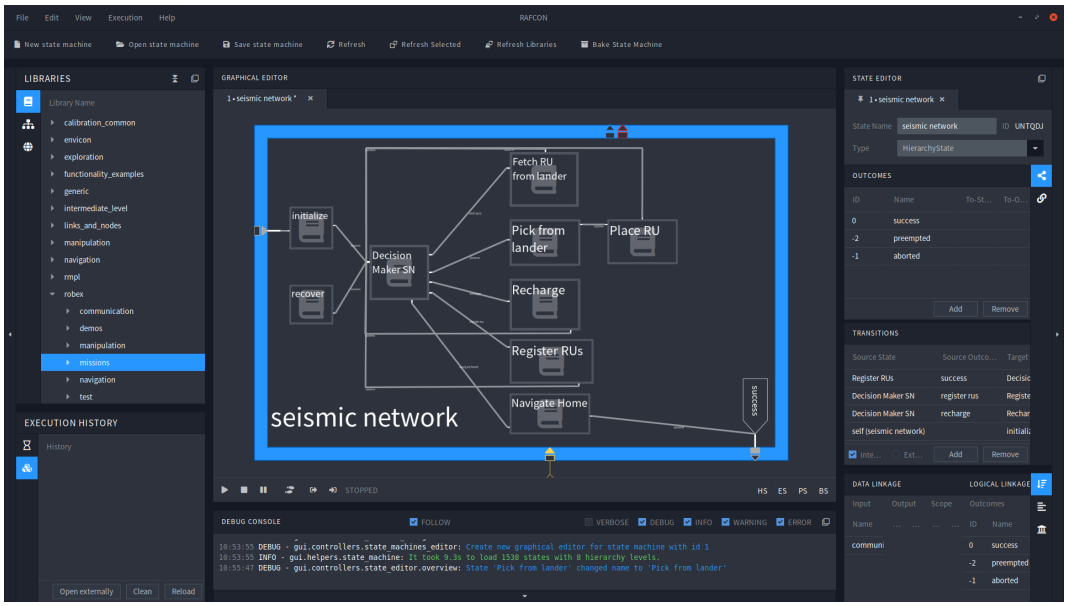


Figure 1.3: *Screenshot of the task control framework RAFCON, responsible for autonomous decision making on the LRU in the context of the ROBEX missions*

1.2 Challenges

Planetary, robotic exploration endeavors, such as those shown in Fig. 1.1, offer many challenges for planning, preparation and execution. Next to the enormous mission complexity, the analysis of the ongoing mission and the optimization of robotic tasks pose many difficulties.

1.2.1 Complexities Imposed by the Environment, the Robot and the Task Constraints

This section describes the main challenges for robot autonomy in the space context, i.e., mission complexity, task constraints, robot complexity (both hardware- and software-wise), challenges for longterm autonomy, non-determinism, unforeseen events and error handling.

Sample return missions are currently one of the hardest missions in the space domain as the sample vehicle has to explore the target area, find a suitable sample, collect it and transport it back to the lander or the space shuttle. The fact that not a single sample return mission since 2000 was completed successfully to the full extent (see Murchie et al. (2014), Williams et al. (2011) and Burnett (2013)) shows that

such endeavors are highly complex. Two major reasons for this are that the target environment is previously unknown to the robot, and that long signal delays (or even no communication possibility at all) prevent a proper remote control of the robotic agent.

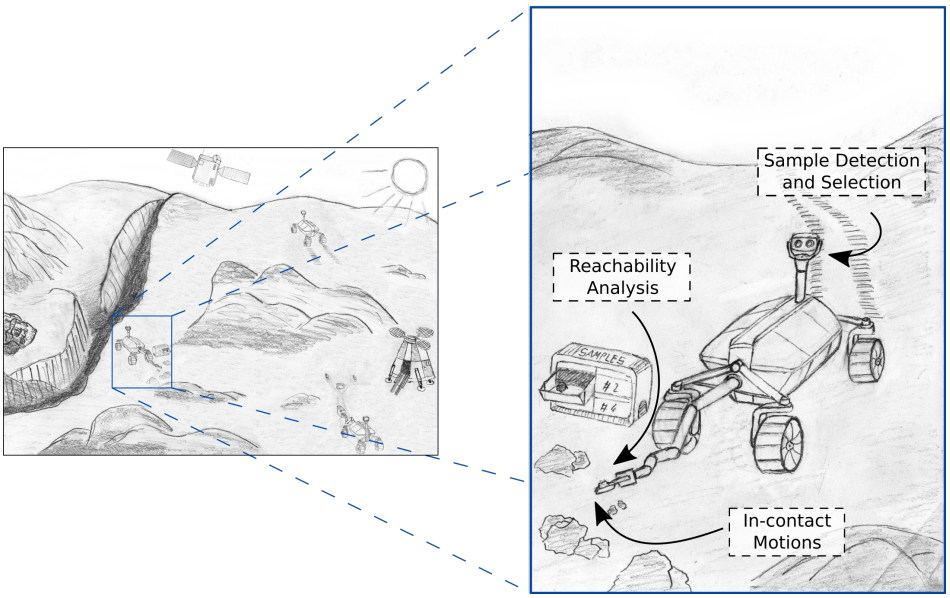


Figure 1.4: *Sketch of a sample return mission, in which the LRU collects a rock sample*

In Fig. 1.4, an example sample return task performed by a space rover is shown. The depicted rover is the LRU, a space rover prototype built by our research group (see Wedler et al. (2015)) at the *Robotics and Mechatronic Center* (RMC) of the *German Aerospace Center* (DLR). A sample return task requires the rover to be equipped with a wide range of hardware and software modules, as shown in Fig. 1.5.

A multitude of actuators is needed in order for the rover to navigate securely to a target area. Additionally, the robot is equipped with a force-torque controlled manipulator to take up the samples from ground. This adds up to many *Degrees of Freedom* (DOF), which can be controlled by the robotic system (nineteen in case of the LRU). Various perception systems are necessary to perform the navigation task, to create a map of the environment and to locate the target sample. For high quality 3D maps, cameras with decent resolutions are required. Estimating the material composition of the target sample (such as small rocks or sand) is performed with infrared cameras. Several camera devices are also needed at the manipulator to guide the in-contact motions while taking up the sample. Even more sensors are necessary to check the power state of the battery, the outside temperature and the grasp forces etc. To address all the actuators, sensors and other hardware components a sophisticated electronic design is required. In order to perform all

internal computations such as environment modeling, object detection, reachability analysis and motion planning powerful computing units are needed onboard the robotic system. Speaking in terms of cognition, the rover has to be aware of this complexity in order to use its capabilities to effectively solve the given tasks (e.g., see Fig. 1.1). This interaction between cognition, sensory and motor functions is called embodied cognition (see Shapiro (2014)).

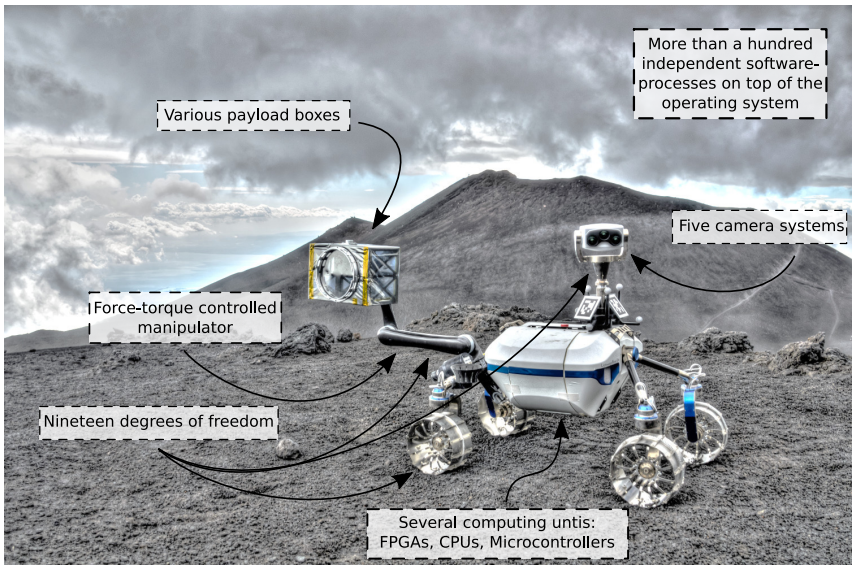


Figure 1.5: *Selected characteristics of the LRU robot*

Another enormous challenge is based on the fact that real space missions require a space rover to act in a robust manner for a long period of time. The Curiosity rover for example (see Strategic Missions and Advanced Concepts Office (2017)), was planned to act for several months on the surface of Mars. Thus, a rover's cognitive capabilities have to be designed in a way that it can fulfill this requirement, taking the challenges of unknown, harsh environments into account. The corridor of possible actions the rover is allowed to carry out must be narrow enough in order not the jeopardize the mission. A transparent specification of all actions, including their preconditions and effects, must be ensured in the face of such high robot complexity, mission criticality and environment uncertainty.

As a matter of fact, space is one of the most challenging domains robots can act in. Robotic systems have to face uncontrolled outdoor environments in space. Uncontrolled means that the robot does not know what it will encounter, i.e., the environment is previously unknown to the rover. In such environments, events can occur, which cannot be controlled by the robot or humans and can alter the capabilities or the goals of the robot. In the space domain this would relate to, e.g., the change of lightening conditions, which makes object detection very hard or even

impossible for certain time spans, or stellar flares (or other solar cosmic ray events), which can cause lasting damage to sensors or actuators. Specifically in space, the environment a rover acts in might be very harsh and thus impassable terrain might be anywhere, which prevents the rover from reaching a target region.

To guarantee robustness in the face of such unforeseen events and uncertainties, the robot needs to be able to detect errors efficiently and react to them appropriately. However, this is a great challenge as the source and number of errors are vast. The correct decision sometimes cannot be derived by the robot itself. Thus, the ability of asking the mission control for help must be possible and deeply rooted into the autonomy design of a space rover. This includes a proper diagnosability and predictability of the robotic behavior.

1.2.2 Task Analysis and Update

Usually, before sending technical equipment to a planet's surface a satellite is placed into the celestial body's orbit (see schematic scenario in Fig. 1.6). The main reason for this is to get a coarse map of the target and to select appropriate landing sites and regions of interest. Furthermore, it is used as communication link between the technical infrastructure on the foreign planet and Earth or among the communication nodes on the surface themselves (i.e., the rovers and the lander).

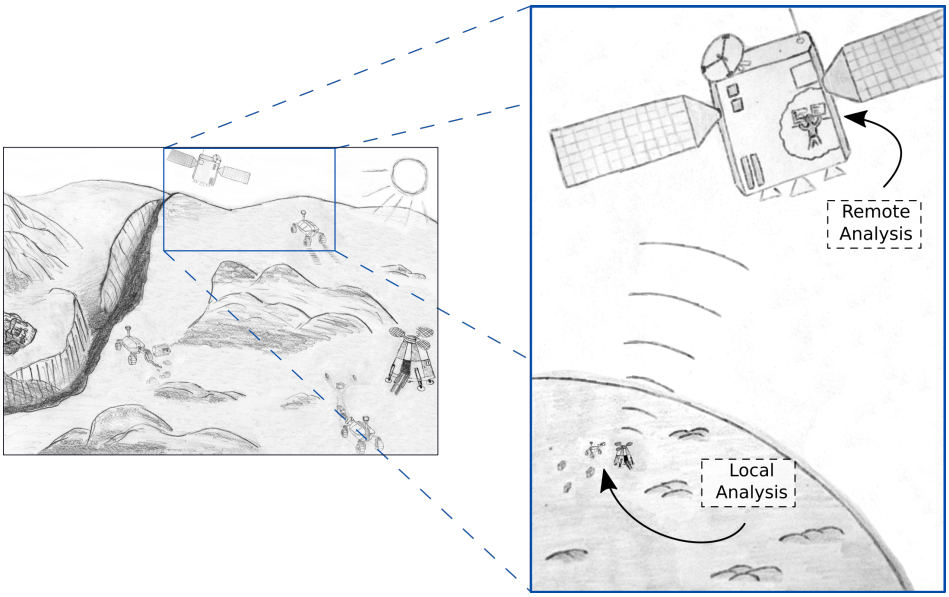


Figure 1.6: *Sketch of the monitoring and analysis of a planetary exploration from an orbiting, manned space shuttle*

Once the link is setup it can be used in three ways to support the autonomous operation on the planet's surface:

1. Mission monitoring: Based on the telemetry data of each active node, exhaustive health checks can be performed and the mission's progress can be tracked. By thoroughly analyzing the robotic tasks, performance bottlenecks can be identified and possible error sources can be detected.

2. Mission updates: Considering the analysis results, mission updates can be sent to the different agents on the planet's surface. Those updates can either consist of completely new goals or of slight changes of the capabilities and objectives of the robot.

3. Error recovery: In opposite to the first two use cases, the robot itself is now the communication initiator. In the case of errors, in which the robot cannot recover autonomously, it may ask for help via the communication channel of the orbiter. Hereupon, the mission control can, in close cooperation with technical and scientific staff, decide upon a possible error recovery procedure and send it back to the rover.

1.2.3 Task Optimization

In planetary exploration missions, there is a manifold of different tasks with attractive scientific output: next to the exploration of unknown terrain and sample return missions, especially the deployment of scientific instruments is of specific interest. For example, seismometers can be used to infer the geological composition of a target region or to estimate the properties of the different crust layers of a foreign planet. Another example are LOFAR antenna arrays, which allow to search and investigate signals created billions of years ago in different frequency bands in order to get insights on the time shortly after the Big Bang - the origin of a vast amount of various electromagnetic signals. Fig. 1.7 shows an exemplary scenario of the setup of four scientific instruments by the LRU rover.

Those tasks are often very repetitive and long-lasting as many instruments of the same kind have to be deployed in several hundred meters distance. Thus optimizing the deployment of a single unit can lead to strong economic benefits. As sending a rover to the surface of a foreign planet costs several billion dollars (see Welch et al. (2013)), making the rover as efficient as possible is a high priority goal. However, for space missions, robustness (i.e., the ability to avoid or reduce any risk to minimum)

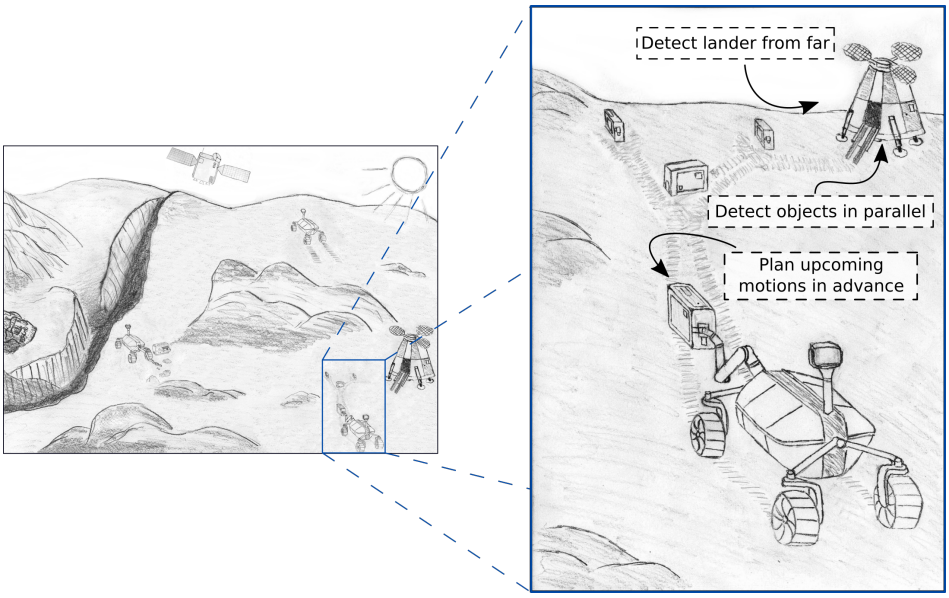


Figure 1.7: *Sketch of the deployment of scientific instruments by the LRU including possible optimization strategies*

is of even higher importance in order not to loose the spacecraft or to jeopardize the scientific achievements. Thus, optimization techniques are required that still ensure total control and trust into the system. The objectives and rules set up by humans have to be ensured in any case, which means that autonomous decisions by the robotic agent are only allowed in a very controlled and limited manner. During and after optimization, the robotic behavior has to be comprehensible and predictable for any point in time. The absence of certain predefined, critical system states must always be provable.

Another reason why the rover has to be efficient is the limited amount of energy per sol[1]. Space rovers often use solar energy (see Landis et al. (2004)) as power supply, which is limited proportional to the duration of the target planet's sol. Navigating inside the shadow of geometric elevations (such as mountains or inside valleys) further reduces the available sun-time as does driving inside cave-like structures. Furthermore, the total battery capacity is also a limiting factor of how long a rover can act per sol. A sufficient buffer for the heating of critical components over night must also be considered in the power management.

[1]The *sol* of a planet is the time, which the planet needs to rotate once around its own rotation axis. On Earth this is equivalent to 24 hours.

1.3 Contributions

The goal of my work is to tackle the presented challenges. Therefore, I present a sophisticated, well tested task control framework for analyzable and resource-aware longterm autonomy, which is composed of three main contributions:

1. a novel, graphical task control language for creating robotic plans that are able to solve complex tasks in a robust manner (see Sec. 1.3.1)
2. an analysis framework for local and remote mission monitoring and for identifying performance bottlenecks in the created robotic plans (see Sec. 1.3.2)
3. a methodology to autonomously optimize the analyzed plans (see Sec. 1.3.3)

In the following, each contribution will be explained in more detail and will be mapped to the presented challenges.

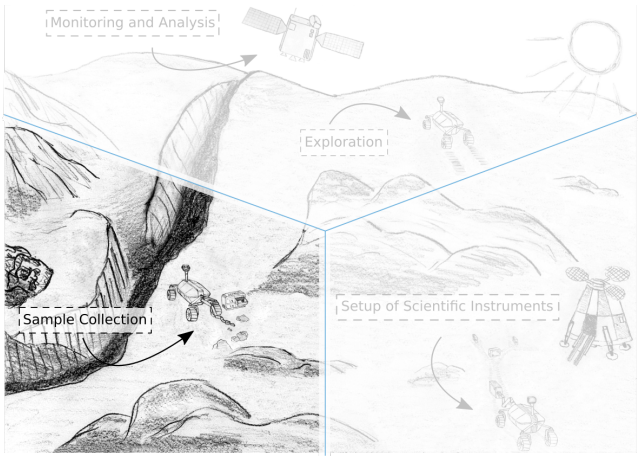


Figure 1.8: *Motivating the first contribution: Creating a task control language to enable robots to solve complex task autonomously, such as the sample return use case shown in Fig. 1.1. The other use cases in the figure are greyed out and are referred to in the next sections.*

1.3.1 The HPFD Graphical Language (HPFD-GL) for Creating Complex Robotic Plans

Description: In my first contribution I present a **graphical task control language** able to define complex robotic behavior (such as shown in the Fig. 1.8). Next to being modular and allowing for easy generalization and reusability of robotic behavior, the language enables the explicit and combined **modeling of control and data flow**. Especially the data flow enables well-informed choices based on system and user parameters and leads to an eminent **diagnosability and transparency of robotic behavior**. The data, which carries parameters and results of critical decisions, can be reliably processed and communicated to other control flow nodes performing more complex decisions. More properties of the language are its clarity, conciseness and its ability to provide deep insights into the contexts of robotic behavior execution. The graphical language supports **Continuous Visual Abstraction**, a concept for representing an arbitrary number of behavior abstraction levels in one 2.5 dimensional view. This enables a developer to use unlimited modeling space in 2D and to model multidimensional symbols, which is a promising approach to reduce the encapsulation problem of common, purely two-dimensional graphical languages.

Realization: The language builds upon the high level programming language Python and extends it by a state-machine-based concept called **HPFD**, representing the basis for the graphical HPFD language. In the scope of this work, I created the

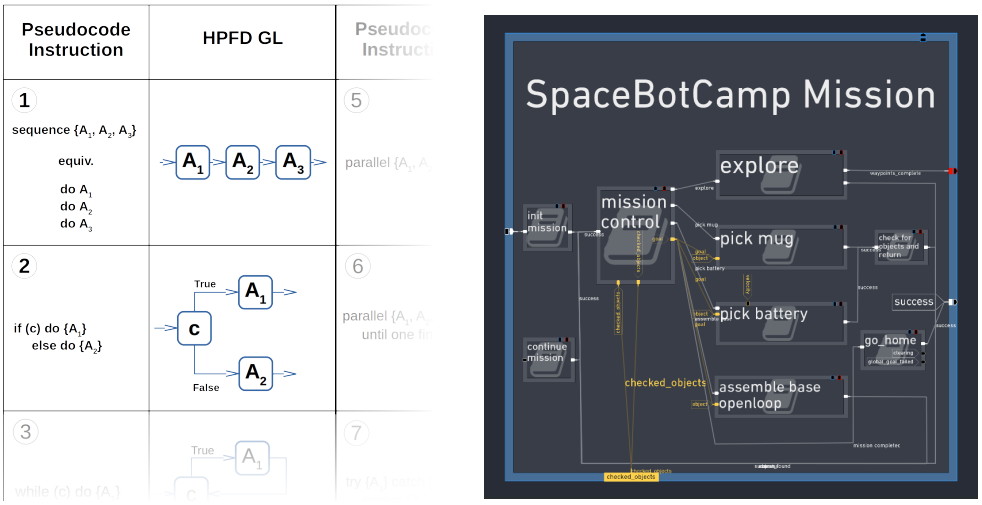


Figure 1.9: *An extract from the HPFD Graphical Language (HPFD-GL) on the left, and the high level state machine used for the SpaceBot Camp on the right.*

RAFCON task control framework[2], which implements the graphical HPFD language. RAFCON allows for visualizing and editing both the logic and data flow in a 2.5-dimensional graphical editor supporting Continuous Visual Abstraction. Thus, it is able to automatically add or hide visualization content based on the content size and zooming level. The left side of Fig. 1.9 shows parts of the graph grammar of the control flow representable by HPFD-GL (presented in Chpt. 4). It offers all building blocks for defining complex behavior, including conditionals, loops, observer structures, various concurrency models, and error handling. The right side of Fig. 1.9 shows a RAFCON rendering of the highest level of the robotic behavior used in the SpaceBot Camp project (a validation experiment highlighted in the following section). RAFCON's powerful execution engine allows for fine-grained execution control as known from modern IDEs.

[2]in collaboration with my colleagues Franz Steinmetz and Rico Belder

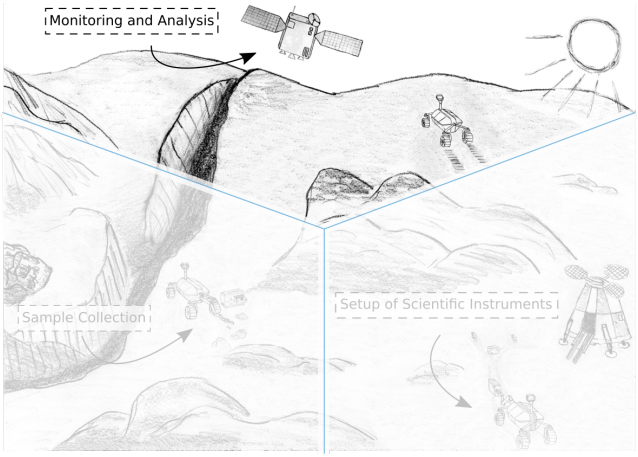


Figure 1.10: *Motivating the second contribution: Mission monitoring and analysis as essential aspects of planetary exploration missions*

1.3.2 Whole Life-cycle Analysis of Complex Robotic Behavior

Description: In my second contribution, I create an **architecture for monitoring and profiling robotic behavior** in order to investigate system failures and to examine the robustness and performance of the behavior. The architecture is especially valuable in scenarios with a high round-trip time for the acquired task logs and sensor data (such as shown in Fig. 1.10). In this context, high round trip times refer to delays of several seconds or minutes (such as the round trip times between the Earth and the Moon or the Mars respectively), which render teleoperation solutions very difficult. This novel analysis framework for robotic behavior offers a **holistic view onto complex robotic tasks for various stakeholders**. It describes analysis possibilities in all three life-cycle phases of a robotic task: the design time phase, the runtime phase and the post-mortem phase. The analysis strongly benefits from the features of the task control language, especially from the information of data flows and the execution history logs. The architecture includes model checking capabilities, **robustness metrics and profiling concepts**, which are able to find robustness and performance bottlenecks. Additionally, the framework allows for the **mapping of semantic task events into both the temporal and spatial domain**.

Realization: I implemented the architecture by using the RAFCON framework. The implementation supports the generation of **detailed, semantic execution histories**, which can be used for the semantic reconstruction of the whole task execution. Fig. 1.11 shows two specific events that happened during the Moon-analogue mission on Mt. Etna, i.e., the LRU rover docking to a seismometer attached to a lander, and

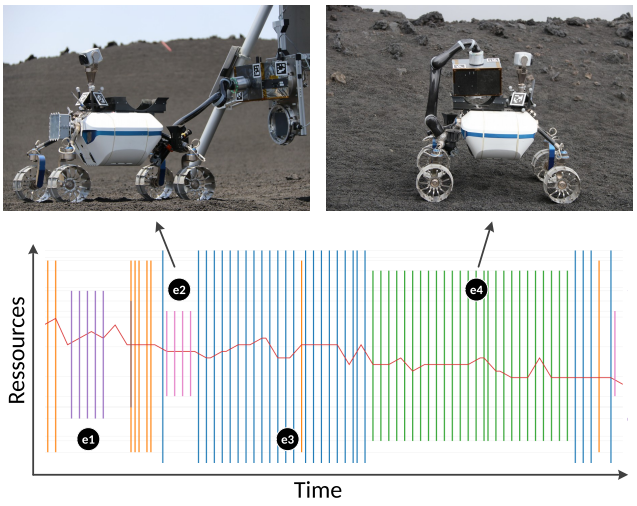


Figure 1.11: *Exemplary mapping of important mission events ex into a resource capability profile recorded during task execution. The colors in the figure map to the different resources used during the mission.*

the rover lifting the seismometer in order to place it onto the ground. By using temporal synchronization those events can be mapped precisely into various kinds of graphs. The presented graph types include Resource Capability Profiles and Gantt charts showing the mission progress and the activation of various system components during task execution. Next to mapping mission events into the time domain, the frameworks allows for mapping them into the spatial domain as well (see Fig. 1.16), and thus eases mission analysis in both domains.

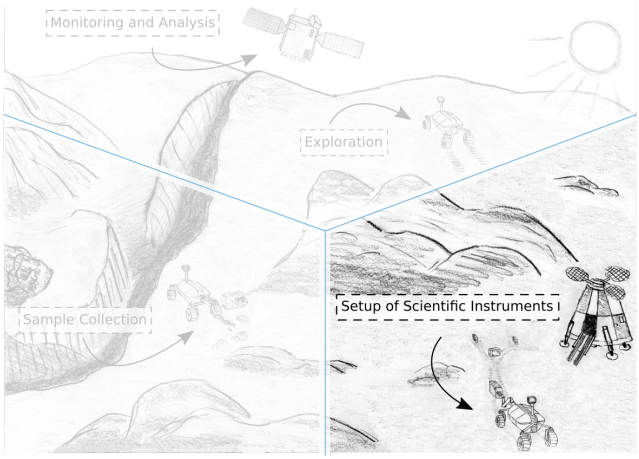


Figure 1.12: *Motivating the third contribution: Autonomously optimizing repetitive tasks, such as the deployment of scientific instruments by a space rover*

1.3.3 Concurrent Dataflow Task Networks (CDTNs): An Approach for Autonomous Optimization of Robotic Plans

Description: In my third contribution, I enable the optimization of the robot's behavior execution by the robot itself, based on the initial, sub-optimal action plans (created by humans) and the analysis results of the logged behavior. Specifically, the **optimization focuses on the runtime performance bottlenecks** and potentials for improvements identified in the analysis phase. The central part of the optimization consists of enhancing the robot's actions with semantic information. This information consists of **three types of semantic dependencies**: First, resource requirements can be modeled for each action, stating which resources are used, blocked, created or released when executing an action. Next, data requirements can be modeled, i.e., the parameters for each action. By this explicit modeling of all data an action needs during its execution, parallelization approaches known from *Dataflow Programming* can be exploited. Finally, semantic world model dependencies are modeled, which map to facts in the robot's belief state or previous achievements of already executed tasks. By leveraging all this semantic information **future actions can be pre-computed and parallelized** in order to save execution time. I show in this work that **significant execution time optimizations of up to** 28% can be achieved for some task types. Especially repetitive or similar tasks benefit most from this approach (such as the task shown in Fig. 1.12). One key aspect of this approach is that these optimizations take place within the framework of clearly modeled dependencies and resources. This results in an optimized workflow without the loss of predictability or confidence, which are essential aspects in every space related mission.

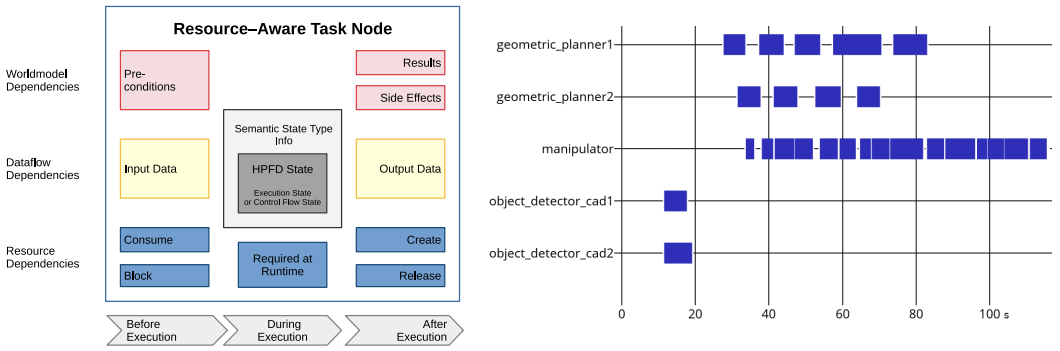


Figure 1.13: *The Resource-Aware Task Node (RATN) action model on the left, and the results of an autonomous task parallelization in form of a Gantt chart on the right.*

Realization: In order to model the semantic information of robotic actions I have created **RATNs**, a novel action model combining operational and descriptive aspects (see left side of Fig. 1.13). It is able to represent all three semantic dependency types. Next to classical *Planning Domain Definition Language* (PDDL)-like precondition and effect modeling, Resource Aware Tasks Nodes are especially suitable to define resource and data dependencies, whereas the latter ones directly benefit from the data flow information of the HPFD task control language.

With the means of RATNs the robotic plans can be converted to ***Concurrent Dataflow Task Network* (CDTN)s**. This is a robotic task representation that by definition does not rely on the rigid execution semantics defined by the transitions of HPFDs but relies on the semantic dependencies of RATNs. I integrated an enhanced execution engine into the RAFCON framework, which is able to automatically optimize plans encoded in CDTNs (while still supporting HPFDs for all types of tasks for which semantic modeling is too intricate or expensive). The implementation renders two types of optimizations possible: On the one hand, independent actions are automatically parallelized. On the other hand, possible future states of the robot's environment are simulated by using a technique called *world model projection*. By using this method, relevant, future actions can already be pre-calculated (e.g., action involving constraint motion planning). Their results can then be used in subsequent actions without delay, which directly leads to the desired task execution speedup. The right side of Fig. 1.13 shows the exemplary use of different resources during plan optimization of a mobile manipulation task. Object detectors are executed in parallel to speed up the reconstruction of the scene. Furthermore, different motion planning instances are executed at the same time and run in parallel with manipulation movements. Especially, as the time for motion planning can be large compared to the execution time of the motion itself, pre-calculating necessary future motions can speed up the task execution time drastically.

1.4 Transferability

Figure 1.14: *Overview of some selected robots programmed by RAFCON.*

Although my motivation and my major evaluation experiment is space-related, my contributions are not only restricted to space use cases. Fig. 1.14 shows on which other robots my contributions in form of the task control framework RAFCON has been used as well. The application domains range from space rovers, over industrial robots, to flying vehicles and finally to human-robot-interaction scenarios.[3]

As in space science, autonomous robots are not employed in these domains for complex tasks yet. Only for very specific, simple tasks robots are used, e.g., for lawn mowing, vacuum cleaning or autonomous transportation in very restricted environments. However, many benefits would arise by employing robots in a broader variety of tasks: the reduction of costs, the increase of efficiency in production, the enabling of new kinds of tasks, which are too hazardous for humans, and the exemption of people from laborious and tedious work. Therefore, I do not only show the applicability of my approach in the space domain but also in some deliberately selected other domains. In my evaluation chapter I present, next to space as my major application domain, a complex use case of RAFCON in the Industry 4.0 context. Showing the applicability of RAFCON in the other two application domains was performed by Vilzmann (2016) and Schuster et al. (2019) for application scenarios using unmanned aerial vehicles (UAVs) and by Steinmetz and Weitschat (2016) for the domain of human-robot interaction.

[3] An overview of all projects in which RAFCON was used is given on RAFCON's website `https://dlr-rm.github.io/RAFCON/projects`. There, even more application scenarios are shown including some in the medical domain.

1.5 Validation of the Proposed Approach

I validated the proposed concepts in several simulated and real world experiments, most notably: the SpaceBot Camp 2015, the ROBEX Moon-analogue demonstration mission and a Gazebo simulator. Their results are presented in the validation chapter, which includes program excerpts of the real, executed robot behavior.

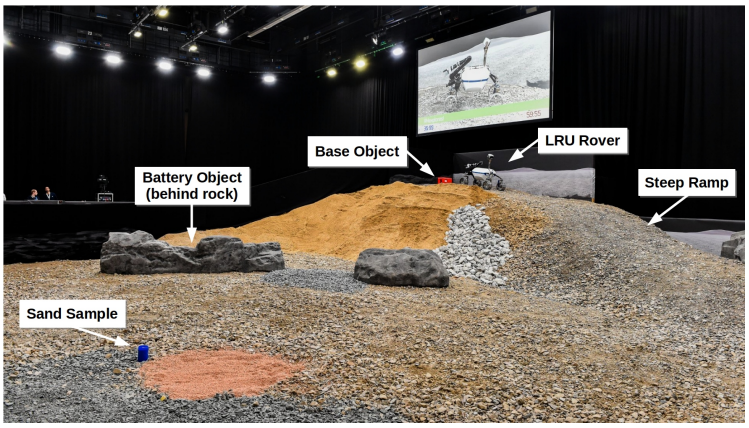


Figure 1.15: *Overview of the SpaceBot Camp challenge setup: The rover had to explore an unknown terrain, pickup two objects (sample and battery object), deliver those objects to the base object and assemble them.*

We successfully employed HPFDs and the graphical task control framework RAFCON for the first time during the SpaceBot Camp 2015 (see Fig. 1.15). The SpaceBot Camp was a national-wide competition of many (space-)robotic research institutes. The goal was to explore an unknown terrain, find two objects, bring them back to base station and assemble them there. Our robot could reach all objectives of the proposed task fully autonomously in only half of the fixed time frame of 60 minutes. RAFCON was responsible for controlling all high level software of the functional layer, i.e., modules for navigation, object detection and localization, manipulation and motion planning. Five developers collaborated on creating the autonomous behavior for our participating robot, i.e., the LRU.

The next major validation experiment is the Moon-analogue demonstration mission ROBEX, which was performed in 2017 on Mt. Etna in Sicily, Italy. In this project, our robot had to perform several missions in a fully autonomous manner: deploying a network of scientific instruments in a previously unknown terrain, using a seismometer in order to analyze seismic events in the target region (see Fig. 1.16), and taking a rock sample from a given point of interest. In order to fulfill the missions autonomously, we employed a sophisticated software architecture on the rover. My

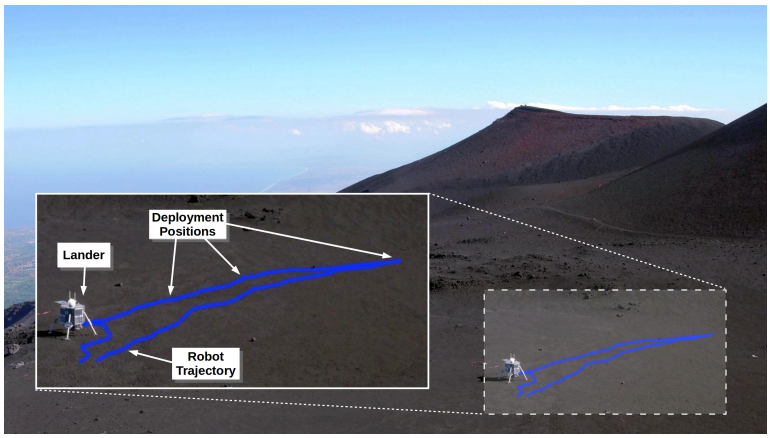


Figure 1.16: *Mission analysis during the ROBEX campaign: The robot's trajectory is mapped into the camera images of a simulated orbiter. Important mission events are highlighted.*

proposed autonomy framework RAFCON was in charge of orchestrating all major software components of the functional layer. It was used to create goal-driven, reactive and parallel behavior and enabled the robot to perform complex decisions on its own. The rover preformed its activities for more than three weeks on the mountain, with around 7 hours of operation time per day. During the test campaign, our robot was able to execute its missions fully autonomously, resulting in self-reliant mission runs with more than 1.5 hours execution time without human interaction (except for changing batteries).

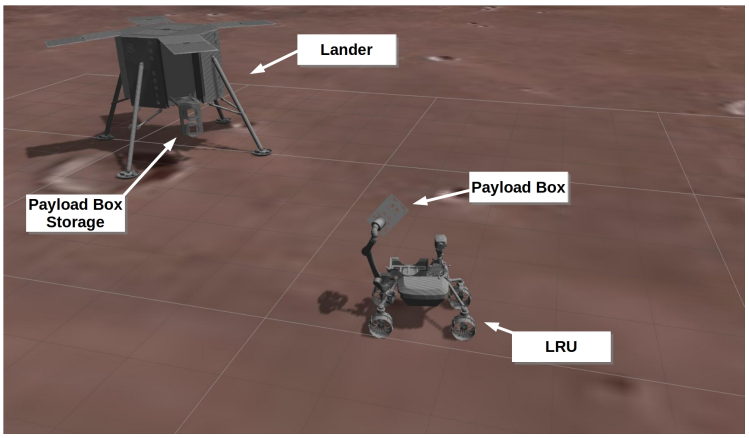


Figure 1.17: *The LRU in a simulator based on the Gazebo framework; it currently picks up a seismometer payload box after having performed seismic measurements.*

Finally, I created a simulator based on the Gazebo framework (see Fig. 1.17), which includes a physics engine to simulate realistic object manipulation challenges. By using the simulator, the high level behavior of our robot could be tested. Furthermore, the simulator was especially valuable for validating my approach for autonomous

execution optimization.

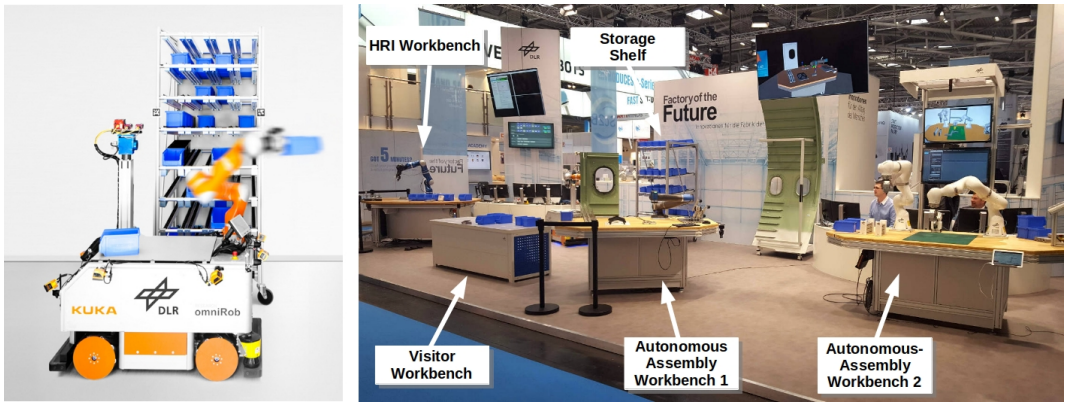


Figure 1.18: *The AIMM robot on the left and our Factory of the Future booth on the right.*

Next to these space related validation experiments, also an industrial use case is presented and analyzed in the evaluation Chpt. 7. In this application scenario, our *Advanced Industrial Mobile Manipulator* (AIMM) robot, see left side of Fig. 1.18, had to deliver different parts to various workbenches in a fully autonomous manner. In the *Storage Shelf* multiple small load carriers (SLCs), filled with different parts, were stored. After fetching the required parts from the shelf, AIMM delivered them either to the *HRI Workbench*, the *Autonomous Assembly Workbenches* or the *Visitor Workbench*. At the latter station, after delivering the parts, the robot picked up other boxes and stored them either in the shelf or brought them to the other work benches. Booth visitors were allowed to disturb AIMM at the visitor's workbench by moving objects away, stacking them or pushing the robot during manipulation. The idea behind this was to show the robustness of the autonomous behavior.

As in the case of LRU, RAFCON was used to control all high-level software modules of our mobile manipulator. Many modules on AIMM are identical or similar to the ones on LRU from a software's perspective. In this case, however, our industrial robot was equipped with more powerful object segmentation and localization routines as AIMM features a *Graphics Processing Units* (GPU) able to run neuronal networks for computer vision purposes. In the context of behavior modeling, the major challenge consisted of making the system as robust as possible to allow for a broad range of disturbances and errors that could be autonomously handled by AIMM.

Finally, as the last part of the evaluation chapter, a small service robotics task is shown, reusing the AIMM robot for a table setting scenario. The task is to set the table with different cutlery and dishes for one, two or three people.

1.6 Dissertation Overview

Fig. 1.19 gives an overview of my thesis. In the left column a chapter overview including the main theoretical contributions of my work is provided. The central column shows all major software packages I created (often collaboratively with other authors). The last column shows the experiments described in my work. The figure shows which theoretical part is evaluated by which experiment. In the following, each chapter is summarized shortly.

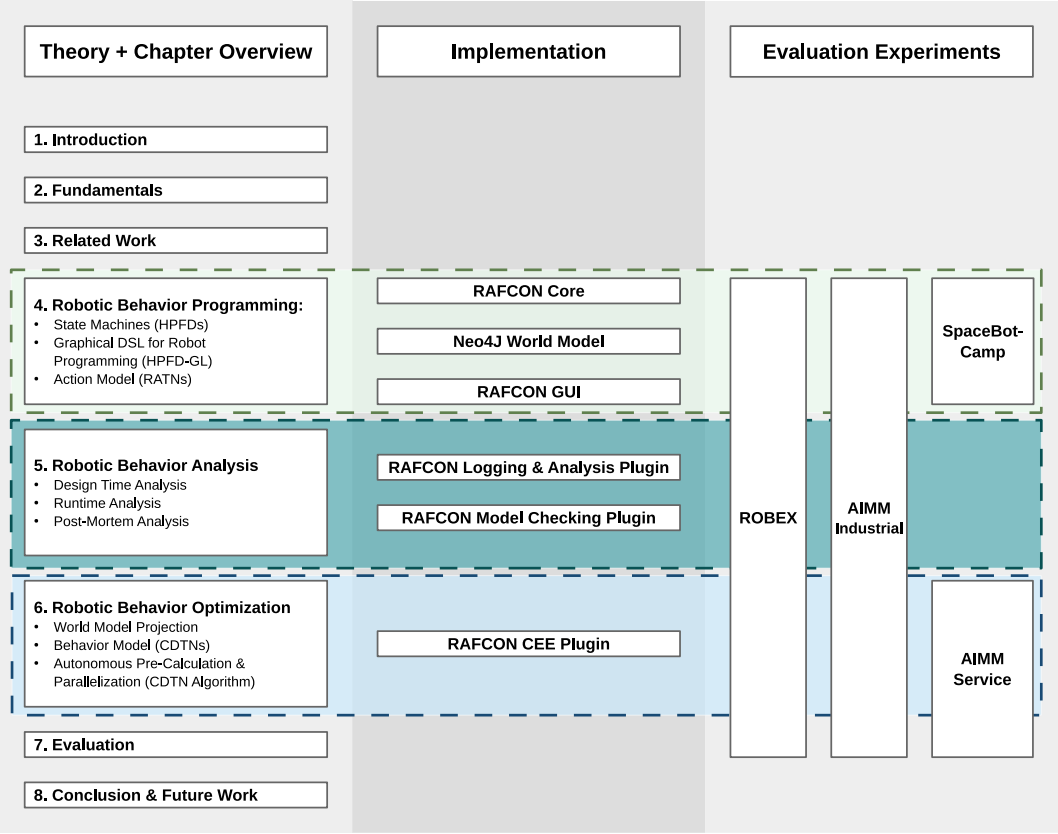


Figure 1.19: *The structural layout of this work. The three main chapters, which align to my three main contributions, are highlighted with a dashed box.*

In **Chapter 1**, I at first give the motivation of my work. Next to providing several challenges of creating, analyzing and optimizing autonomous behavior it summarizes my contributions. Subsequently, information about the evaluation of my theoretical concepts are given. A literature overview of all of my published papers concludes this section.

Chapter 2 presents an overview of classical software architectures for autonomous systems. After that, an action taxonomy and the term *Task Control Framework* is

defined, which is needed to identify and classify the related work. Before describing the LRU robot in more detail, an abstract software architecture for mobile robots, called 2T*, is presented.

Chapter 3 shows a classification of task control frameworks based on different metrics. The metrics range from domain applicability over feature scope to post-runtime analysis capabilities.

In **Chapter 4**, I present HPFDs, a state machine dialect suitable for representing complex robotic behavior. On top of that, I propose a graphical DSL able to represent HPFDs in a concise and scalable manner. Subsequently, I provide insights on how the belief state of robots can be modeled efficiently by using Property-Graphs. Finally, I provide a new action model combining operational and descriptive information, called Resource-Aware Task Nodes (RATNs).

Chapter 5 focuses on my second main contribution, i.e., a holistic approach for robotic behavior profiling. This consists of analysis concepts for all three robotic behavior lifecycle phases, i.e., the design phase, the runtime phase and the post-mortem phase.

Chapter 6 describes an approach for autonomous behavior execution optimization by leveraging the concepts of HPFDs and RATNs. The optimization takes advantage of parallelization and pre-computation possibilities offered by the partially ordered set of actions encoded in semantically annotated HPFDs.

Chapter 7 shows the validation of my approach consisting of several evaluation experiments as listed in the third column of Fig. 1.19. The experiments include the ROBEX mission, a supply chain task in an industrial setting using the AIMM robot and a small service robotics scenario (also employing AIMM). The SpaceBot Camp experiment only covers the theoretical work of Chapter 4, and is thus treated there. In the second part, I discuss my achieved results and also put them into context to related work.

In **Chapter 8**, I finally conclude my thesis by summarizing my main contributions and by providing ideas for further elaboration on the presented topics.

1.7 Prior Publications

In the following, all relevant publications to which I contributed are listed. They include both theoretical contributions and validation experiments. For clarity purposes, I grouped them into different categories. The bold items represent my main publications.

Task Programming:

- S. G. Brunner, F. Steinmetz, R. Belder, and A. Dömel. RAFCON: a Graphical Tool for Task Programming and Mission Control. In *20th RoboCup International Symposium*, 2016a
- **S. G. Brunner, F. Steinmetz, R. Belder, and A. Dömel. RAFCON: A Graphical Tool for Engineering Complex, Robotic Tasks. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016b**
- **P. Lehner, S. G. Brunner, A. Dömel, H. Gmeiner, S. Riedel, B. Vodermayer, and A. Wedler. Mobile Manipulation for Planetary Exploration. In *Aerospace Conference*. IEEE, 2018**

Task Profiling and Optimization:

- **S. G. Brunner, P. Lehner, M. J. Schuster, S. Riedel, R. Belder, A. Wedler, D. Leidner, M. Beetz, and F. Stulp. Design, Execution, and Postmortem Analysis of Prolonged Autonomous Robot Operations. *IEEE Robotics and Automation Letters (RA-L)*, 3(2):1056–1063, 2018**
- **S. G. Brunner, A. Dömel, P. Lehner, M. Beetz, and F. Stulp. Autonomous Parallelization of Resource-Aware Robotic Task Nodes. *IEEE Robotics and Automation Letters (RA-L)*, 2019**

System Architecture and Autonomous Agents:

- S. Jentzsch, S. Riedel, S. Denz, and S. G. Brunner. TUMsBendingUnits from TU Munich: RoboCup 2012 Logistics League Champion. In *RoboCup 2012: Robot Soccer World Cup XVI*, volume 7500 of *Lecture Notes in Computer Science*, pages 48–58. Springer Berlin Heidelberg, 2013
- M. J. Schuster, C. Brand, S. G. Brunner, P. Lehner, J. Reill, S. Riedel, T. Bodenmüller, K. Bussmann, S. Büttner, A. Dömel, W. Friedl, I. Grixa, M. Hellerer,

H. Hirschmüller, M. Kassecker, Z.-C. Marton, C. Nissler, F. Ruess, M. Suppa, and A. Wedler. The LRU Rover for Autonomous Planetary Exploration and its Success in the SpaceBotCamp Challenge. In *IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, 2016

- **M. J. Schuster, S. G. Brunner, K. Bussmann, S. Büttner, A. Dömel, M. Hellerer, H. Lehner, P. Lehner, O. Porges, J. Reill, S. Riedel, M. Vayugundla, B. Vodermayer, T. Bodenmüller, C. Brand, W. Friedl, I. Grixa, H. Hirschmüller, M. Kaßecker, Z.-C. Márton, C. Nissler, F. Ruess, M. Suppa, and A. Wedler. Towards Autonomous Planetary Exploration: The Lightweight Rover Unit (LRU), its Success in the SpaceBotCamp Challenge, and Beyond. *Journal of Intelligent & Robotic Systems (JINT)*, 2017**

- A. Wedler, M. Vayugundla, H. Lehner, P. Lehner, M. J. Schuster, S. G. Brunner, W. Stürzl, A. Dömel, H. Gmeiner, B. Vodermayer, R. Rebele, I. Grixa, K. Bussmann, J. Reill, B. Willberg, A. Maier, P. Meusel, F. Steidle, M. Smisek, M. Hellerer, M. Knapmeyer, F. Sohl, A. Heffels, L. Witte, C. Lange, R. Rosta, N. Toth, S. Völk, A. Kimpe, P. Kyr, and M. Wilde. First Results of the ROBEX Analog Mission Campaign: Robotic Deployment of Seismic Networks for Future Lunar Missions. In *68th International Astronautical Congress (IAC)*, 2017

- A. Wedler, M. Wilde, J. Reill, M. J. Schuster, M. Vayugundla, S. G. Brunner, K. Bussmann, A. Dömel, M. Drauschke, H. Gmeiner, H. Lehner, P. Lehner, M. G. Müller, W. Stürzl, R. Triebel, B. Vodermayer, A. Börner, R. Krenn, A. Dammann, U.-C. Fiebig, E. Staudinger, F. Wenzhöfer, S. Flögel, S. Sommer, T. Asfour, M. Flad, S. Hohmann, M. Brandauer, and A. O. Albu-Schäffer. From Single Autonomous Robots Toward Cooperative Robotic Interactions for Future Planetary Exploration Missions. In *International Aeronautical Congress (IAC)*, volume 42. International Aeronautical Federation (IAF), 2018

- A. Wedler, J. Reill, M. J. Schuster, M. Vayugundla, and S. G. Brunner et al. Insights into the Robotic Exploration Activities in the Research Section of the German Aerospace Center (DLR). In *Symposium on Advanced Space Technologies in Robotics and Automation (ASTRA)*, 2019a

- A. Wedler, M. Wilde, J. Reill, M. J. Schuster, M. Vayugundla, S. G. Brunner, K. Bussmann, A. Dömel, H. Gmeiner, H. Lehner, P. Lehner, M. G. Müller, W. Stürzl, B. Vodermayer, M. Smisek, B. Rebele, and A. O. Albu-Schäffer. Analogue Research from ROBEX Etna Campaign and Prospects for ARCHES Project: Advanced Robotics for Next Lunar Missions. In *European Planetary Science Congress (EPSC)*, 2019b

- M. J. Schuster, M. G. Müller, S. G. Brunner, and H. Lehner et al. Towards Heterogeneous Robotic Teams for Collaborative Scientific Sampling in Lunar and Planetary Environments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Workshop on Informed Scientific Sampling in Large-scale Outdoor Environments*, 2019

- M. J. Schuster, M. G. Müller, S. G. Brunner, H. Lehner, P. Lehner, R. Sakagami, A. Dömel, L. Meyer, B. Vodermayer, R. Giubilato, M. Vayugundla, J. Reill, F. Steidle, I. von Bargen, K. Bussmann, R. Belder, P. Lutz, W. Stürzl, M. Smíšek, M. Maier, S. Stoneman, A. F. Prince, B. Rebele, M. Durner, E. Staudinger, S. Zhang, R. Pöhlmann, E. Bischoff, C. Braun, S. Schröder, E. Dietz, S. Frohmann, A. Börner, H.-W. Hübers, B. Foing, R. Triebel, A. O. Albu-Schäffer, and A. Wedler. The ARCHES Space-Analogue Demonstration Mission: Towards Heterogeneous Teams of Autonomous Robots for Collaborative Scientific Sampling in Planetary Exploration. *submitted to IEEE Robotics and Automation Letters (RA-L)*, 2020a

- M. J. Schuster, B. Rebele, M. G. Müller, S. G. Brunner, A. Dömel, and B. Vodermayer et al. The ARCHES Moon-Analogue Demonstration Mission: Towards Teams of Autonomous Robots for Collaborative Scientific Sampling in Lunar Environments. In *submitted to European Lunar Symposium*, 2020b

- A. Wedler, M. G. Müller, M. J. Schuster, S. G. Brunner, and P. Lehner et al. First Results from the Multi-Robot, Multi-Partner, Multi-Mission, Planetary Exploration Analogue Campaign on Mt. Etna in Summer 2020. In *submitted to 71st International Astronautical Congress*, Dubai, United Arab Emirates, 2020

Supervised Master and Bachelor Theses:

- M. Büttner. Robot Flow Control - Development of a Modular, Graphical User Interface. Master Thesis, Université Toulouse-III-Paul-Sabatier, 2015

- V. Niewada. Development of an Autonomous Mobile Robot. Master Thesis, Strasbourg University, 2015

- C. Sürig. Analyse und Evaluierung semantischer Logikplaner, mit anschließender Integration des Vergleichsergebnisses in RAFCON. Bachelor Thesis, Hochschule München, 2018

- R. Sakagami. Cooperative Operation Framework for Heterogeneous Robotic Teams in Planetary Exploration. Master's thesis, The University of Tokyo, 2020

Chapter two

Fundamentals

Robotic tasks are becoming increasingly complex, and with this also the robotic systems. In the introduction chapter the LRU (one of the robots relevant for my work) was already presented shortly: it has nineteen DOFs, five camera systems, a force-torque controlled manipulator and a docking system, able to grasp various types of tools. Such robot specifics require powerful software architectures and tools to manage their complexity and to orchestrate the robotic system in order to fulfill demanding autonomous tasks.

This section presents the scope of this thesis and distinguishes it from work focusing on artificial, purely simulated domains and tasks (e.g., see Gupta and Nau (1992) or Russell and Norvig (1995)). After presenting some coarse overview of common robot software architectures and action taxonomies, the term *task control framework* is defined in detail. Finally, the architecture of the LRU is shown. It is deliberatively explained in a detailed manner in order to be able to refer to the various software modules throughout the thesis. Defining them here in this chapter in a centralized manner allows me to sketch them in a consistent view.

2.1 Software Architectures for Robotic Systems

There is already a lot of related work about robotic system architectures. Their focus ranges from reactive approaches over deliberative architectures to cognitive systems

and finally to learning-based approaches.

One major class of robotic system architectures are **reactive** control architectures such as the subsumption architecture (see Brooks (1986)) and architectures employing Behavior Trees (see Colledanchise and Ögren (2014)) as their central action selection scheme. The subsumption architecture consists of several layers of different (hierarchy) levels, where each layer is a complete operational control system. Higher level layers can subsume the behavior of a lower level layer and can take control of the system. One major advantage of this architecture is the reactive nature to external events with small delays. The main drawback is that they are not deliberative, i.e., that they cannot create new behavior by planning, which is imperative in the face of new type of tasks. Behavior trees primarily focus on procedural task control. For an in depth presentation and discussion of Behavior Trees (including a comparison with my approach) see Sec. E in the Appendix.

Deliberative architectures operate the robot in a top-down fashion. Each time there is a new task or new events occur, a top level planner analyses the current world state and creates a plan that allows the robot to achieve its goal. In contrast to a reactive architecture, in which control only consists of sense-act couplings, a sense-plan-act cycle is the fundamental concept of deliberative architectures.
Some famous examples of deliberative systems are the three-layered (see Siciliano and Khatib (2007)) and two-layered (see Volpe et al. (2001)) architectures. The three-layered architecture (see left side of Fig. 2.1) consists of the planning layer on top, an executive layer in the middle and the behavioral control (often also "functional") layer at the bottom. The dimension up from one layer to the next (in vertical image direction) depicts the increasing intelligence of the robot: from reflexive, via procedural, to deliberative behavior. In the top layer, planning is performed to reach formally specified task objectives using predefined actions and the information of a central, persistent world state. The executive layer cares about the execution of the high-level plans generated by the top level. The high level plans are divided into sub-tasks of different abstraction levels and translated from discrete symbols into the sub-symbolic, numeric space. Finally, the bottom layer cares about the robot's physical abilities. Hardware controllers, sensor data evaluation and complex algorithms for path planning and computer vision are executed here using the parameterization received from the executive layer.

One disadvantage of the three-layered architecture is that the planning layer does not have direct access to the functional layer. Therefore, it often maintains a dedicated model of the system and the environment, "which may not be directly derived from the functional layer. This repetition of information storage often leads to inconsistencies between the two" (Volpe et al. (2001)). Thus, the two-tiered architecture, as

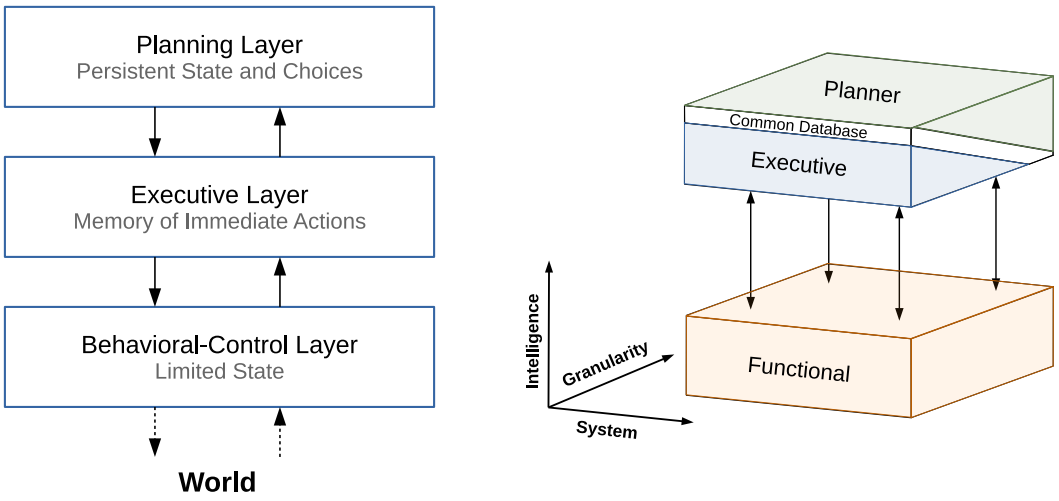


Figure 2.1: *Classical layered architectures: The three-layered architecture as proposed by Gat et al. (1998) (left) and the the two-layered architecture as proposed by Volpe et al. (2001) (right)*

shown in right side of Fig. 2.1, combines the planning and executive layer. They share a common world model, thus avoiding inconsistencies between the representation in the planning layer and the representation in the executive layer. The two-tiered approach, furthermore, explicitly represents the granularity of the system in a dedicated dimension, where granularity refers to the hierarchical structure inherent to each layer. The intelligence dimension was already mentioned above, and the system dimension simply expresses the breadth of the robotic system in terms of hardware and capabilities.

Robotic system architectures also have their origin in **cognitive** architectures such as Act-R as proposed by Anderson (1996) or Soar, created by Laird (2012). A sophisticated overview of cognitive architectures is given by Vernon (2014).
Soar (former SOAR, which was an acronym for: State, Operator, And Result), is a cognitive architecture for the modeling of cognitive processes in cognition science and *Artificial Intelligence* (AI). J. E. Laird invented Soar as a unified theory of cognition (see Newell (1992)), as it defines all mechanisms and structures upon which human cognition and cognition in general is based. This means that by using Soar, robots could be built that are capable of human behavior. Furthermore, Soar is a control architecture for autonomous robots. It has been used in various robotic scenarios (see Puigbo et al. (2013) and Laird et al. (2012)). In Soar, task control is performed by mapping the current state of the robot and its environment together with the task objectives (i.e., the so called goal states) into a problem space. In this problem space, a search is performed to reach a goal state starting with the current state. In each state several operators can be applied. If one operator is chosen and applied the

current state can transform to a new state. Even complex actions such as planning a motion or a new action can be composed of simple operators inside the problem space. One advantage of Laird's methodology is that the problem space must not be represented completely but only the subset of states relevant during problem solving. More features of Soar are different kinds of memories such as short-term working memories and long-term, persistent memories including episodic, descriptive and procedural memories. Finally, learning is used to "fill" the different memories with content based on the information of the agent's experience.
However, there exists a lot of criticism about Soar as well. Cooper and Shallice (1995) investigated the early versions of Soar and came to the conclusion that "whilst Soar represents an impressive body of research, its methodological foundations are insecure, it is ill specified as a computational/psychological theory, and under empirical testing it does not stand up to close scrutiny as a unified theory. The Soar research program as it currently stands thus fails to meet the necessary methodological demands imposed by unified theorising". Based on the point of view of Allard (2014), certain technologies apply better to certain tasks. He continues by stating that reactive architectures might be better in many situations as they do not suffer from the higher delay of cognitive decision making. Another critique with Soar is that there are few tools available to assist in programming, debugging, and maintaining these cognitive architectures (see Allard (2014)). Furthermore, although Soar is quite old and dates back to 1983, there is only a small community developing Soar itself[1]. Finally, Puigbo et al. (2013), who implemented Soar on a mobile service robotic, state that "SOAR cannot detect if the goal requested to the robot is achievable or not. If the goal is not achievable, SOAR will keep trying to reach it, and send skill activations to the robot forever".

Finally, there exists the class of **learning**-based architectures. In the case of being able to produce a large amount of realistic data for an agent (e.g., by having an exact and complete simulation available), reinforcement learning (see Sutton and Barto (1998)) is a promising approach. It is based on the idea to iteratively face the agent with new tasks and observe how it reacts. Based on the reaction a cost function is evaluated stating if the agents behavior was good or bad. If the cost is low then the agent's reaction is rejected. Otherwise it is integrated into the agents behavior. The applicability of reinforcement learning was successfully shown by OpenAI's OpenAI Five for the Dota computer game (see OpenAI (2019)) as well as for Google Deepmind's AlphaStar AI for the even more complex Starcraft game (see Vinyals et al. (2019)). As having an exact simulation is very hard for complex, real robots, such simulations do not exist yet.

[1] see https://github.com/SoarGroup/Soar

Another problem of learning-based architectures is that for complex scenarios (such as robotic manipulation) it is very hard to find suitable, intermediate goals for which the robot gets rewards for. If the reward is only provided after achieving the final goal (e.g., stacking several objects) that the robot, because of the task's complexity, will never reach without further guidance, than the learning process gets stuck at the very beginning (see Sutton and Barto (1998)). Thus, employing reinforcement learning for all abstraction layers of autonomous behavior has not been performed for real scenarios yet.

2.2 Action Taxonomies

This section provides a literature overview of the definition of the term "action". Furthermore, it provides an action taxonomy, which is consistently used throughout this work.

There are several definitions of the terms *action*, *skill* and *task* in literature. The term "skill" was already used in the work by Archibald and Petriu (1993). In this definition, a skill "is an ability of a robot to repeatably accomplish any useful action that can be described unambiguously". Furthermore, skills are composed of "methods", which are modules responsible for driving an actuator or interpreting sensor data. Finally, an "operation" is a collection of robot skills that is able to fulfill a specific task. Skills feature a formal description including preconditions and post-conditions.

The definition of Andersen et al. (2014) is very similar: They define an "action" as an abstract superclass of which "skills" and "primitives" are specializations. Primitives (which map to the "methods" of Archibald and Petriu (1993)) are the basic functionalities of a robotic system and interface hardware such as actuators and sensors. Skills are composed of primitives and can also be composed of skills themselves. A "task", finally, is a process description and consists of a set of fully parameterized skills. Primitives, skills and tasks all have a formal description. Another related action taxonomy was created by Pedersen et al. (2016), which distinguishes between "device primitives", "skills" and "tasks".

Thomas et al. (2013) also uses a similar action abstraction approach, however, the naming is again slightly different. The lowest level of actions are called "elemental actions (EA)", which access the robot's hardware. A "skill" models a "capability" of the robotic system and consists of a "net of EAs". A "task" is composed of a "net of skills". Tasks can then be combined to model the final "process", e.g., an assembly

process.

Finally, Zech et al. (2018) created a detailed survey about the definition of the term "action" as used in many different fields, including (reinforcement) learning, motion planning, assembly and task planning. The paper shows that the term "action" is not standardized and used differently in different contexts.

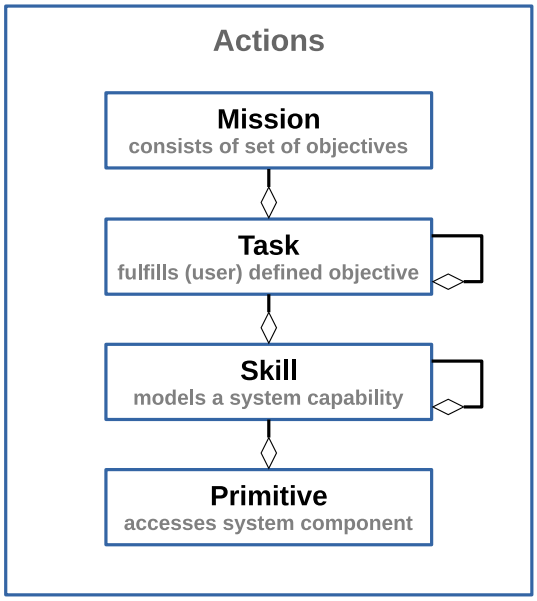


Figure 2.2: *The action taxonomy as used throughout this work; there are four different types of actions: missions, tasks, skills and primitives; missions consist of tasks, tasks of other tasks (i.e., sub-tasks) or skills, and skills of sub-skills or primitives*

In order to allow for a consistent usage of these terms, I provide my own definition action taxonomy in Fig. 2.2, which is similar to the one of Andersen et al. (2014). There are four types of actions: primitives, skills, tasks and missions. Primitives are used to access the robot's hardware, such as actuators and sensors. On top of that, they provide access to arbitrary software modules of the functional layer (see Sec. 2.4), e.g., object detection, path planning and navigation services. Skills consist of primitives and combine them to more complex behavior. Skills model specific capabilities the robot is able to execute. Analogue to a high level capability (*drive-to-poi-collision-free*) being able to consist of lower level capabilities (*drive-to-poi*), a skill can be composed of skills by themselves (i.e., sub-skills). Tasks consist of skills and lower-level tasks (i.e., sub-tasks) and represent behavior able to fulfill defined objectives, e.g., *bring-object-o-to-place-p1* or *perform-measurements-at-place-p2*. These objectives can either be defined by a user or generated by high level mission planning component. Finally, the term mission describes a set of objectives and, thus, consists of one or several tasks.

2.3 A Specification of the Term "Task Control Framework"

The term *task control framework* is a very broad term, which is not defined precisely in the robotics domain. Different researchers call their modules responsible for task control differently, e.g., "behavior control" (Schillinger et al. (2016)), "software toolbox for the design, the implementation, and the deployment of cognition-enabled autonomous robots" (Beetz et al. (2010)) or "task-level robot control" (Bohren and Cousins (2010)). In the course of this work, *"task control framework"* refers to the module(s) in a robotic system that are responsible to select the robot's next action in order for a robot to achieve a specific goal. In other words, it enables a robot to work autonomously.

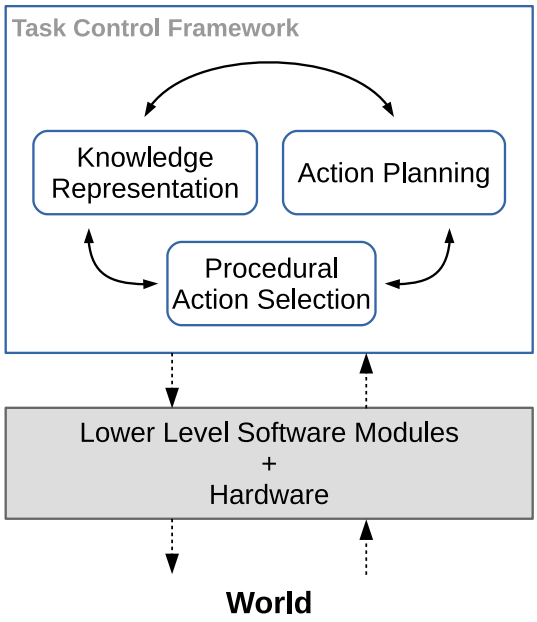


Figure 2.3: *Common responsibilities of a task control framework*

There exist many different task control frameworks (see Chpt. 3). They differ by the set of properties they equip the robot with, such as the level of intelligence, generality, efficiency, scalability and analysability, just to name a few important ones. Fig. 2.3 shows the most basic building blocks of a task control framework in an abstract manner. It represents the largest common denominator of all frameworks analyzed in the related work chapter. They are

- Procedural Action Selection: It is responsible to execute a robotic action in a hard-coded step-by-step fashion. It executes sequential, conditional or concurrent behavior as defined by a programmer beforehand. An exemplary order of

triggered actions would be: First do *step1*, then *step2*. If *step2* fails, do *step3*, otherwise perform *step4* and *step5* in parallel.

- Knowledge Representation: Every analyzed approach commanding a robot in an autonomous manner uses some kind of knowledge representation. It is a central or distributed way of storing all information a robot has about itself and its environment, such as the current set of goals, the current state of other objects and agents, and the current internal state (e.g., power consumption, computation load).

- Action Planning: In scenarios with a high level of task variability or complexity, an action planning component is used to select the next action. Opposed to the *Procedural Task Control* the action sequence is not predefined by a programmer but generic approaches (such as heuristic search or constraint solving) are used to choose the next action in order to bring the robot one step closer towards its goals (for more information see Sec. D of the appendix). Also, user interfaces to enable humans to supervise or support the action planning are part of this module.

2.4 2T* - A General Robot Architecture

Building a robotic system means to chose an appropriate system architecture from existing approaches or to define a new one. As the common architectures include various drawbacks I created a new architecture, which is called *Enhanced Two-Tiered Architecture* (2T*). The name refers to its similarity with the two-tiered architecture. Although 2T* resembles the two-layered architecture in many aspects, there are still some modifications and features that justify the definition of a dedicated architecture.

Robotic architectures are often formulated in a very abstract manner, which leaves too much freedom for the developer. Thus, system experts try to “expand the capabilities and dominance of the layer within which they are working” (Volpe et al. (2001)). As a consequence, robotic systems emerge, in which (in the case of the three-layered architecture) either “the function layer is dominant, the executive dominant or the planner is dominant” (Volpe et al. (2001)). Thus, the 2T* definition is rather fine-grained and tries to guide the developer to integrate the required functionality at the appropriate location.

The 2T* graph, as shown in Fig. 2.4, combines and structures abstract building blocks

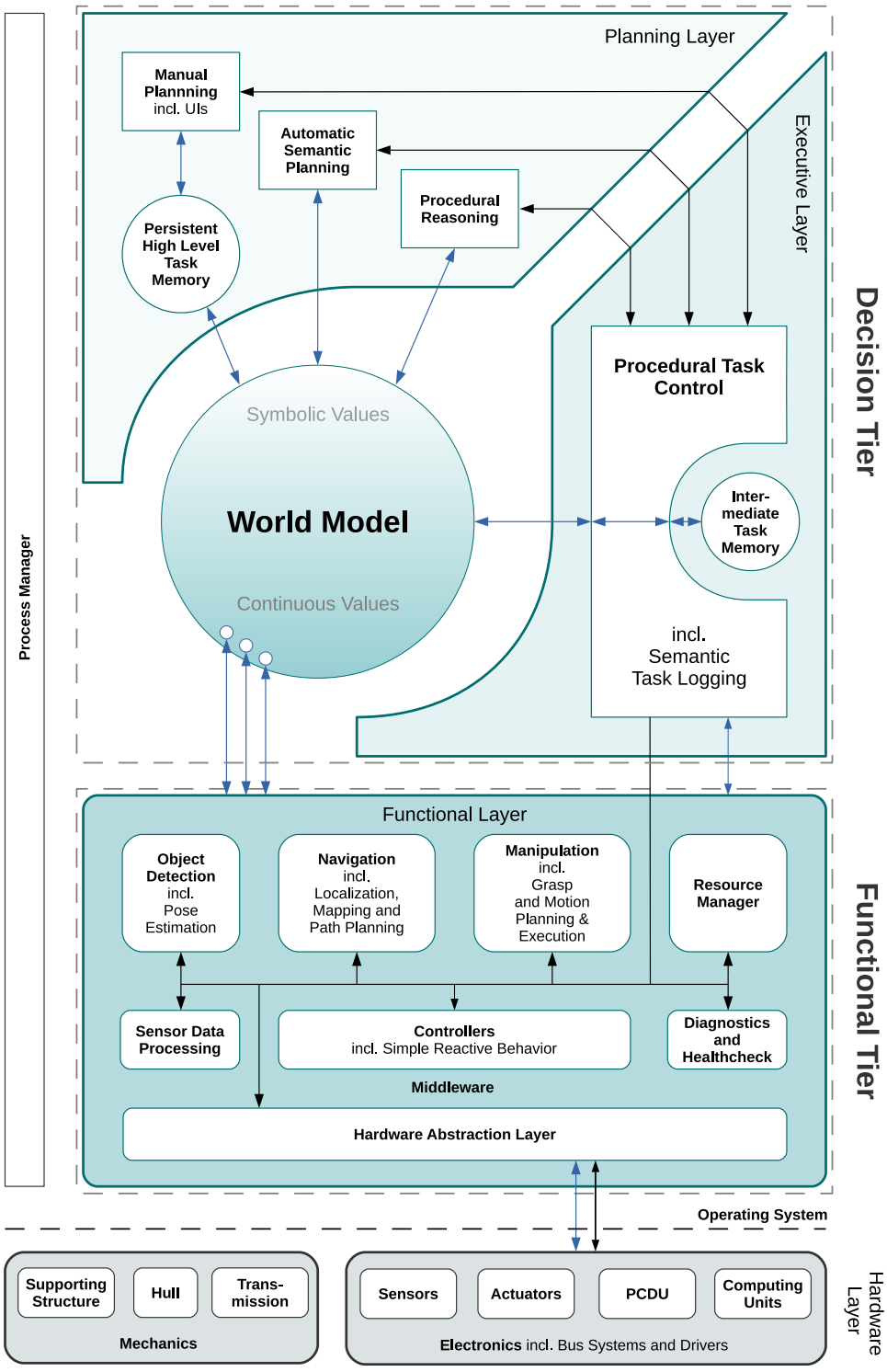


Figure 2.4: *The 2T* architecture; white boxes represent modules, black arrows control and blue arrows data flow; round boxes represent memory modules*

of complex, mobile agents. It consists of three main software layers in total: The "Planning Layer", the "Executive Layer" and the "Functionl Layer". All software layers are visualized in teal color. The lighter the color, the higher the cognitive abstraction level. Although not belonging to the software part of a system, an additional hardware layer is shown for better overview and consistency. It is visualized in gray, which shows that its components are not software-related (except of some driver firmware).

In the context of this architecture, *layers* must not be confused with *tiers*. As in the two-tiered architecture (see Sec. 2.1) the first tier (i.e., the decision tier) consists of the planning and executive layer. The second tier consists of the functional layer, which accesses the hardware layer of the robot to drive its actuators or receive sensor readings. Before highlighting the differences between the two architectures the different building blocks of 2T* are presented briefly. For a concrete example implementation of the architecture, I refer to Sec. 2.5. As for the two-tiered architecture the breadth (horizontal axis) of 2T* maps to system's complexity and capabilities, and the height (vertical axis) to the system's intelligence.

The missing modeling of the data flow is a major issue in most of the architectures. In 2T*, I model the data flow as follows: The blue arrows show the data flow between the different layers of the architecture. Inside each layer, dataflow can exist between every module and is thus not modeled by arrows. The black arrows show the interaction between the components of different layers in a logical sense, which includes function invocation and parameterization.

The Hardware Layer

The hardware layer constitutes the hardware of a robotic system. First and foremost, it consists of mechanical parts such as the supporting structures and the hull, which together define the robot's kinematics. Furthermore, a robot has actuators, such as motors and powertrains, and sensors, such as cameras, *Light Detection and Ranging* (LIDAR) systems or force sensors. Another important aspect are the electronics, such as electronic boards for accessing the drivers and sensors, and computation units e.g *Central Processing Units* (CPU), GPU and *Field Programmable Gate Arrays* (FPGA). Moreover, a *Power Control and Distribution Unit* (PCDU) is needed for providing power to all system components. Finally, various bus systems connect all electric devices and allow for communication between the computation units, the actuators and sensors. The electronic devices are integrated in and accessible via the operation system, visualized as a dashed line.

The Functional Tier

The functional tier consists of the functional layer only and provides different functionality to the upper layers of the robotic system. All communication inside the layer is done via one or several middlewares, which also connect the executive layer to the modules of the functional layer. The most basic module is the ***Hardware Abstraction Layer* (HAL)**[2], which enables the functional layer to communicate with all electronic devices, especially the actuators and sensors.

The **sensor data processing** node uses the data provided by the HAL and performs filtering, image registration, (down-)sampling or stereo matching tasks. It is then used by other modules such as **object detection** and **navigation**. The former module is used for detecting objects, estimating their relative pose to the robot or classify them. The latter one is responsible for estimating the robot pose, building a map of the robot's surroundings and planning collision free paths.

The paths are sent to and executed by the **controllers**, which are responsible for moving the robot's base, arms, grippers or actuated perception devices (such as a pan-tilt unit). An imperative aspect for in-contact motions and manipulation of delicate objects is the ability to command and track the forces applied to each joint. By using predictions based on the own kinematic and dynamic structure, external forces can be measured and reacted to.

Various **manipulation** modules equip the robot with the capability of deliberately changing its environment. Therefore, grasp planning, obstacle avoidance, reachability analysis, waypoint interpolation and motion planning are required.

The **resource manager** is a component keeping track of all resources relevant for the robot and the task. It tracks, e.g., the robot's power, the available tools and storage spaces, and task relevant objects of the environment. More information about the resource manager is given in Sec. 6.2.1. As managing processes (such as creating additional object detection and planning modules for task execution speedup as shown in Sec. 6.3.2) is also a central part of resource management, it is tightly coupled to the external **process manager**. The process manager cares about starting, stopping and monitoring the system processes of the modules of all 2T* layers.

The last module of the functional layer is the **diagnostics and healthcheck** component. It monitors all system critical states. This includes checking the available power not falling below a certain threshold, listing current error states of all processes and checking environmental conditions such as temperature or communication availability.

[2]The HAL is not, as the name might suggest, an architectural layer such as the planning or executive layer. It is a common name for the hardware abstraction unit.

The Decision Tier

The decision tier consists of the world model, the executive layer and the planning layer.

The World Model A robot's world model holds the robot's current knowledge about its environment, its own system state and the task objectives, both, the already fulfilled and the pending ones. Knowledge structuring, classification and management are critical in order to be able to represent and manipulate complex information. The information can consist of two types: symbolic and sub-symbolic values. Symbolic values are abstract and can refer to arbitrary entities (such as objects of the environment, the robot itself or other agents) and facts (such as environment states or object properties). Sub-symbolic values are continuous or discrete numeric values such as object poses, voltage levels, or mass and force information.

The information of the belief state has to be represented in appropriate data structures. Often, triple stores (see Margitus et al. (2015)) or graph databases (see Blumenthal et al. (2013)) are used to efficiently store and retrieve the information. Graph structures allow for the investigation of new contexts and ultimately the inference of new information (for more information see Sec. 4.4).

Concept-wise, the world model belongs to the first tier, i.e., the decision tier consisting of the planning and executive layer. Additionally, it also offers a reduced interface to the functional layer. The advantage of this approach, compared to dedicated world models in each layer, is consistency. If the information about the robot's environment is split into several layers, consistency is hampered as the information has to be synchronized between the layers. Furthermore, information redundancy emerges if the same knowledge is processed in various layers concurrently.

Principally, a centralized knowledge storage can lead to performance break-downs if several modules are allowed to make extensive changes. However, this is not the case in 2T*. The functional layer is only allowed to update specific pre-defined continuous values in the world model such as robot positions, manipulator *Tool Center Point* (TCP) positions or important telemetry data. Also, it may read a predefined set of knowledge, e.g., the geometric representation of the robot's environment (i.e., the CAD models), which is required for motion planning. The planning layer is only allowed to modify high level semantic information such as task objectives (e.g., three blue objects have to be fetched instead of two). The only layer allowed to conduct substantial changes to the belief state is the procedural task control. This layer calls scene parsing and object localization routines and uses their results to create and

update all information about the robot's environment.

The Executive Layer In the executive layer, tasks are executed by a **procedural task control** unit. Depending on the complexity of the task, it can consist between tens and several hundreds of actions (see Brunner et al. (2018)), and must thus allow for extensive modularization. Actions are executable functions, that perform calculations, control a actuators or read sensor values.

In order for a robotic system to scale, an action model is needed. A consistent action interface and description is essential if the number of available actions or their parameterization possibilities grows. Actions can be defined for arbitrary abstraction levels. For example, there are actions just reading some measurements from a sensor and other ones computing and executing complex trajectories for a robotic manipulator. Next to the abstraction level, an action model defines several other relevant properties. Regarding this, I refer to the descriptive and operational elements of the action model described in Sec. 4.5.

To keep track of all fulfilled intermediate task objectives an **intermediate task memory** is required. Based on this task memory, the outcome of the executed actions and their parameters, the task execution decides upon the next action to be executed. The task execution does not only care about the logic flow between actions (i.e., which execution is executed next) but also about the data flow (i.e., how the next action is parameterized). Data scopes, which restrict the data access are important for a modular and generic task architecture (data handling is treated in Sec. 4.1.1 in more detail).

The **semantic task logging** module is responsible for logging all executed actions in a so called execution history. All parameters and output data of actions, including their outcomes and errors are logged. During the execution of various tasks the robot can accumulate experience data. This can be used to get the average execution duration for frequently executed actions and finally be utilized to calculate performance and robustness bottlenecks. Next to the execution history the experience data contains recorded raw data streams of, e.g., forces, poses or images. This data is indexed using timestamps and can be accessed by using the timestamps of the actions in the execution history. More information about how experience data can be exploited is shown in the task analysis Chpt. 5 and the validation Chpt. 7.

The Planning Layer The planning layer is the highest layer of the 2T* architecture. In here, the deliberation about task objectives takes place including the making of high level decisions concerning execution start times and deadlines. Based on this information a plan implementing all task objectives is created by using one or a mix

of the following methods.

First and foremost **manual planning** can be used to create plans. Therefore, user interfaces are required in order for an operator or mission control team to be able to create a plan interactively. Various tools can ease this interactive plan creation such as wizards, automatic checkers or templates.

Another possibility for creating plans is to use **procedural reasoning**, which in this non-formal context (similar to Ingrand et al. (1996)) uses predefined routines to search through a problem domain and decide upon the next action to take. Thus, it is no general purpose planning system but rather a methodology to assemble predefined action sequences and to select specific action parameterization which fulfill the task constraints. One example of this are parallel observer structures that monitor certain system or environment states in order to modify a robotic behavior accordingly (for concrete examples see Sec. 4.3.2). Another example are decision trees, which check various conditions in a hierarchical manner to decide upon the next action to take (see Sec. 4.3.2 for a concrete example). Furthermore, custom loops or recursive functions can be used to search for valid actions and action parameters that fulfill specific task constraints (for examples, in which possible grasp and approach poses are evaluated based on reachability and collision constraints see Sec. 4.3.2)

Finally, an **automated semantic planning** module can generate a plan by taking the robot capabilities, the target domain and the robot's environment into account. These planners are based on heuristic search and try to find valid action sequences in the problem domain. Opposed to procedural reasoning, in which specific routines focus on selected task constraints, automated planning find new, complete plans only based on the current system state and the goal constraints. More information about semantic planning and how it is integrated implementation-wise is given in Appendix D.

All (sub-) task objectives are stored in the **persistent high level task memory**. Per default, it is not located on the robot but kept externally in order to be accessed by a high level mission planner responsible for several robots. Thus, it is not bound to specific agents and does not suffer from system breakdowns or losses. Furthermore, it is a storage to keep all decisions of a human operator before scheduling tasks for the agent(s).

Important 2T* Specifics

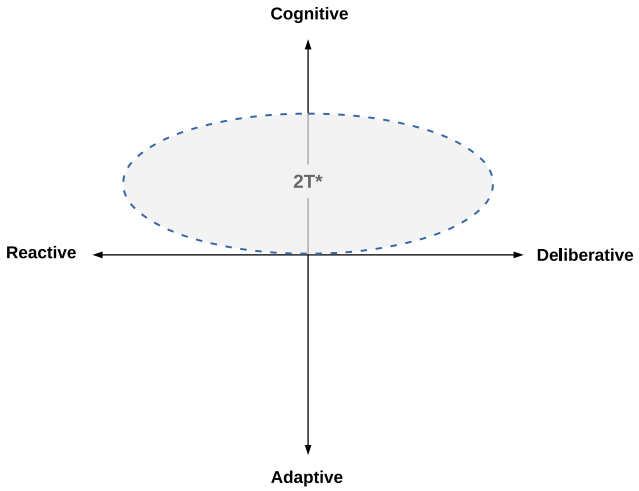


Figure 2.5: *A classification of the 2T* architecture. The dimensions map to architecture properties already defined in Sec. 2.1.*

The 2T* architecture has several properties of the **common architecture patterns** presented in Sec. 2.1. It is influenced by reactive, deliberative and cognitive trends (see Fig. 2.5). On the one hand, it is not purely deliberative as such behavior is often not capable of reacting to critical events in time. On the other hand, it is not purely reactive as well, as such systems are not suitable for reaching complex goals in environments in which goal driven behavior including (pre-)planning is required. Finally, cognitive systems tend to produce non-deterministic behavior which is difficult to reconcile with the robustness requirements needed for critical tasks especially in space scenarios. The architecture is not (yet) capable to adapt to the environment or task by using learning-based approaches. In future work, I will focus on adding adaptive capabilities to 2T*.

As stated above, the 2T* layers are similar to the layers of the **two-tiered architecture**. Nevertheless, there are several important differences and enhancements: The hardware layer is omitted in the two-tiered architecture, which is an essential part for embodied cognition (see Shapiro (2014)). In principle, embodied cognition means that true cognition can only be achieved by taking feedback of physical actions into account. The dynamics of the robotic structure (structural stiffness, masses and inertia) and the agent's environment (e.g., force events) play a vital role. The higher cognitive layers must not neglect this knowledge. Thus, 2T* shows the hardware layer including its interaction with the executive layer.

The two-tiered architecture does not put any constraints onto the **middleware**,

which, however, is a vital part of 2T*. The middleware is not above the controller in a hierarchical sense, but serves as input and output of the controller. The executive layer has direct access to low level sensor readings available via the middleware. Thus, the executive layer can access both, the data going into the controller and the data going out. This includes forces, currents, positions, velocities and accelerations. Hence, it is possible to read out single joint values and force measurements and derive task-related decisions directly from this data. For example, based on the contact force between a target object and the robot's gripper, a robot can infer if it grasped the object or not and decide upon the next action to execute.

The **world model** is not only accessible by the planning and executive layer, but from the functional layer as well. This is especially beneficial for updating states which change continuously, e.g., the robot's pose or the location of its TCP. In case a decision depends on such information, it does not have to be synchronized beforehand explicitly as it is always up to date.

The **resource manager** and the **process manager** are central parts of the architecture. The resource manager manages and tracks the use of the computing units, which are also part of the hardware layer. The process manager manages the location control. In combination they take care about scalability and load balancing.

The **planning layer** can just be a human operator (or team) defining a simple sequence or a complex reactive behavior, e.g., by using a graphical user interface. Thus, the planning layer does not have to resort to an automated semantic planner in all cases. This is advantageous, e.g., if there is a lot of interaction between the operator and the robot, the available number and heterogeneity of actions including their parameterization exceeds the capabilities of a semantic planner, or not all of the domain inherent objects and relations can be modeled because of missing knowledge about the robot's environment (which holds true in several use cases including planetary exploration).

Cognitive architectures for robot control highlight the importance of knowledge management and define **dedicated memory modules** (e.g., SOAR by Laird et al. (1987) or ACT-R by Anderson (1996)). The same holds true for 2T*, in which memory modules are visualized as circular shapes, whereas processing/deliberation nodes are rectangular. The various memory modules serve different purposes concerning persistence, access restrictions, scalability and information rate.

To conclude the architecture definition, I define the responsibilities of the layers and the various modules. This is done via the principle of *Separation of Concerns*. Separation of Concerns is an elementary concept of elaborated software architectures. This important principle of modern system engineering tries to reduce complexity

through abstraction and a clear allocation of responsibilities. “Separation of concerns can also be characterized as introducing a loose coupling between concerns. The effect of the loose coupling is that changes to one concern have a limited or no effect on other concerns. Single concerns can be flexibly changed or exchanged without affecting other concerns.” (Hürsch and Lopes (1995)) There are many important concerns relevant for controlling complex robotic systems. Radestock and Eisenbach (1996) identified the concerns *Computation, Communication, Configuration and Coordination* as being one of the most important. Hürsch and Lopes (1995) identified more concerns, such as *Class Organization, Synchronization, Location Control and Failure Recovery* and, furthermore, proposed separation techniques for and composition of concerns. In the following I will list all major ones and point to the place in the architecture, where the specific concern is treated.

- **Computation**: In general, all instructions performed by a computer can be referred to as computation or calculation. In the context of this architecture I differentiate between two classes of computations: Computation for (sensor) data processing and computation for action selection and parameterization. Sensor data processing is done in the functional layer (e.g., in the context of environment modeling and path planning, object detection, and map generation and navigation). The computation for action selection and parameterization is done in the decision tier. This includes behavior execution, (semantic) task planning, and knowledge generation and processing.

- **Communication**: All modules have to receive data, based on which they perform their calculations. Computed results are then forwarded to other nodes for further computation. The communication is done by one or several *middlewares*, which connect all modules inside and between the respective layers.

- **Synchronization**: Normally, data must be able to be received in the order it was sent. Furthermore, some nodes need their data to be timely aligned (e.g., the two image streams for stereo matching). Thus, it must be possible to synchronize data relative to their timestamps and send order. A *middleware* has to offer possibilities to do so. When talking about synchronization in terms of concurrent action execution and coordination, the *procedural task control* has to offer respective features.

- **Real-Time Constraints**: An important concern identified by Hürsch and Lopes (1995) is the ability to serve real-time constraints. Especially for manipulator or UAV control this is imperative. Dedicated hardware and software modules

rely on the real-time property (controllers, filters etc.). The operating system and at least one middleware have to support real-time communication.

- **Configuration**: Configuration as defined by Radestock and Eisenbach (1996) handles the challenge of defining which nodes talk to which other nodes and which data streams or communication channels are used. Obviously, also dynamic system re-configurations during runtime can be a crucial aspect. Both are the responsibility of the *process manager* and the *procedural task control*.

- **Location Control**: In order to decide, which node runs on which computer in a distributed system, location control is used. One major goal of this distribution is optimizing the distributed system concerning load and communication costs. The process manager is responsible for this task.

- **Parametrization**: One way to achieve high reusability is that *all components* offer a broad set of parameterization possibilities. This especially holds true for all robot capabilities, called by the *procedural task control*. Offering many parameterization possibilities lead to a reduced total set of primitive actions, which have to be defined in order to access all robot capabilities.

- **Coordination**: Coordination is the key feature of the *executive layer* and the *planning layer*. While the executive layer contains many domain-specific routines and cares about the specific sub-task- or object-dependent coordination, the planning layer is responsible for general, task-related, cross-domain coordination.

- **Persistence**: In order to be able to recover from system reboots (e.g., because of severe system failures), concepts responsible for storing robot knowledge in a persistent format is required. Task-related data, the robot's belief state and execution histories have to be stored in order to be able to debug the system. The *world model*, the *persistent high level task memory* and the *semantic task logging* are responsible for that.

- **Composition**: Composition is the major approach to tackle complex problems by using the principle of "divide and conquer". *Each component* has to employ this important principle in order for their concepts being understandable, modifiable and reusable.

- **Analyzability or Debuggability**: *Every node* has to offer easy possibilities to analyze and debug its behavior. Common means are informative logging outputs, useful comments, graphical visualizations and various verbosity or debug levels configurable upon startup or even during runtime.

- **Failure Recovery**: Failure recovery is another complex concern identified by Hürsch and Lopes (1995). If possible, *each component* has to perform local failure recovery on its own. In a complex system, failures have naturally complex reasons that can only be identified by components with a lot of context information, i.e., the *procedural task control*.

2.5 The Lightweight Rover Unit (LRU)

This section highlights the complexity of real systems, which is not present for agents in artificial problem spaces, such as the *blocks world* (see Gupta and Nau (1992)) or the *Wumpus world* (see Russell and Norvig (1995)). This includes a high domain, task and robot complexity, non-deterministic actions, uncertainty in both the symbolic and sub-symbolic domain, sensor noise and non-linear behavior of manipulator actuation.

The LRU is one of the more complex, autonomous mobile robots in literature (see Schuster et al. (2016), Wedler et al. (2015) and Schuster et al. (2017)). We used it in various real world scenarios including the SpaceBot Camp competition (see Schuster et al. (2016)), the ROBEX Moon-analogue demonstration mission (see Lehner et al. (2018)) and the ARCHES project (see Schuster et al. (2019)).

In the following, I will describe how we implemented the 2T* on the LRU. Fig. 2.6 shows the architecture of the LRU robot with the focus on the software modules. The colors of the figure are identical to the colors of Fig. 2.4. This allows for an easy mapping between the abstract architecture definition and the implemented software modules.

The figure's main intent is to show compositional information, i.e., which software module belongs to which 2T* layer. The control and data flow of the functional layer was already shown in 2T* definition and is omitted for clarity purposes. For an in-depth explanation of the data flow inside the functional layer I refer to our previous publication (see Schuster et al. (2017)). The names of the software modules are omitted as well, except for middlewares and those modules that will be presented in detail in this work. Instead of the name, the functional capability of each node is shown.

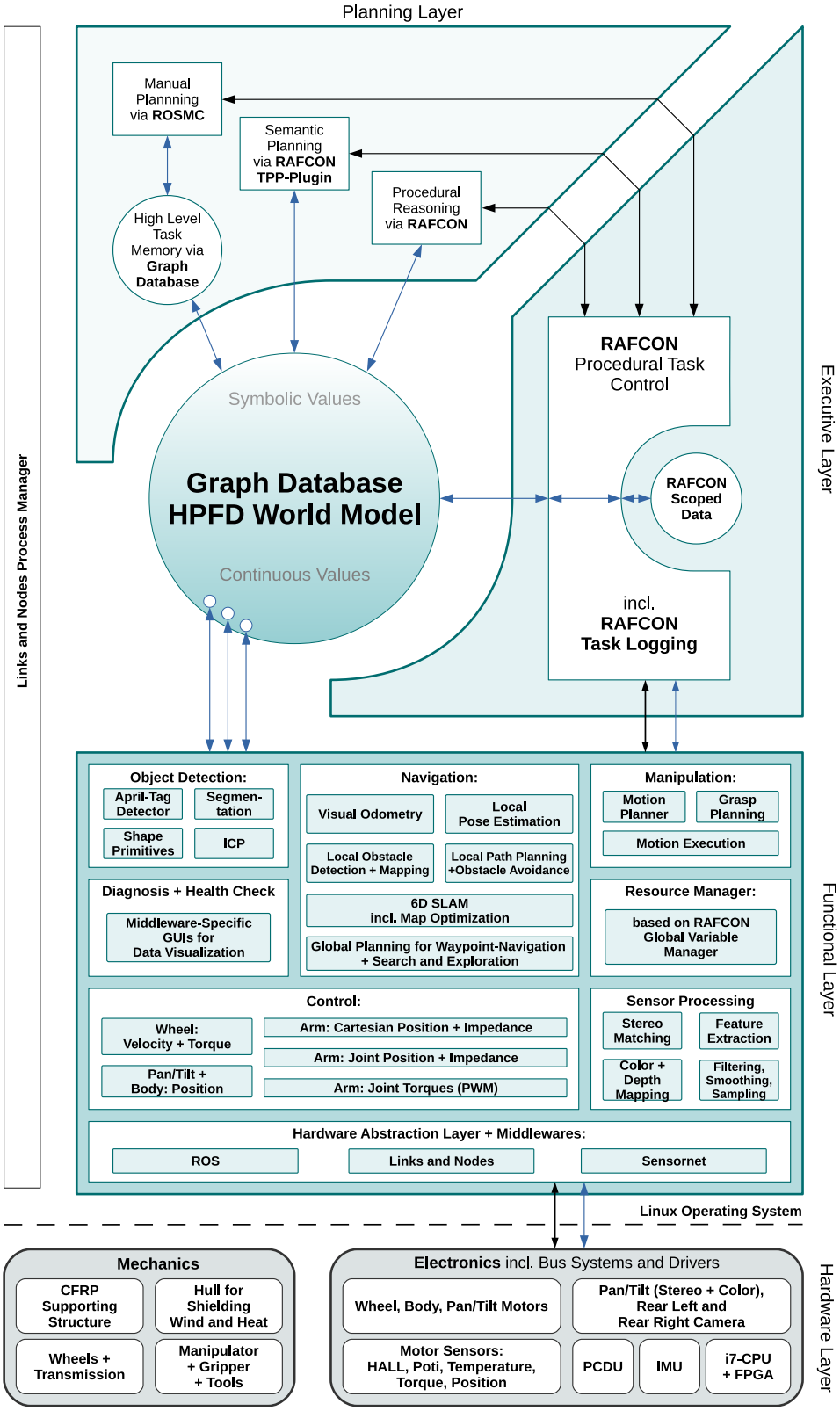


Figure 2.6: *The 2T* architecture (see Fig. 2.4) as implemented on the LRU robot. A more detailed visualization of the functional layer including data and logic flow is given by Schuster et al. (2017).*

The Hardware Layer

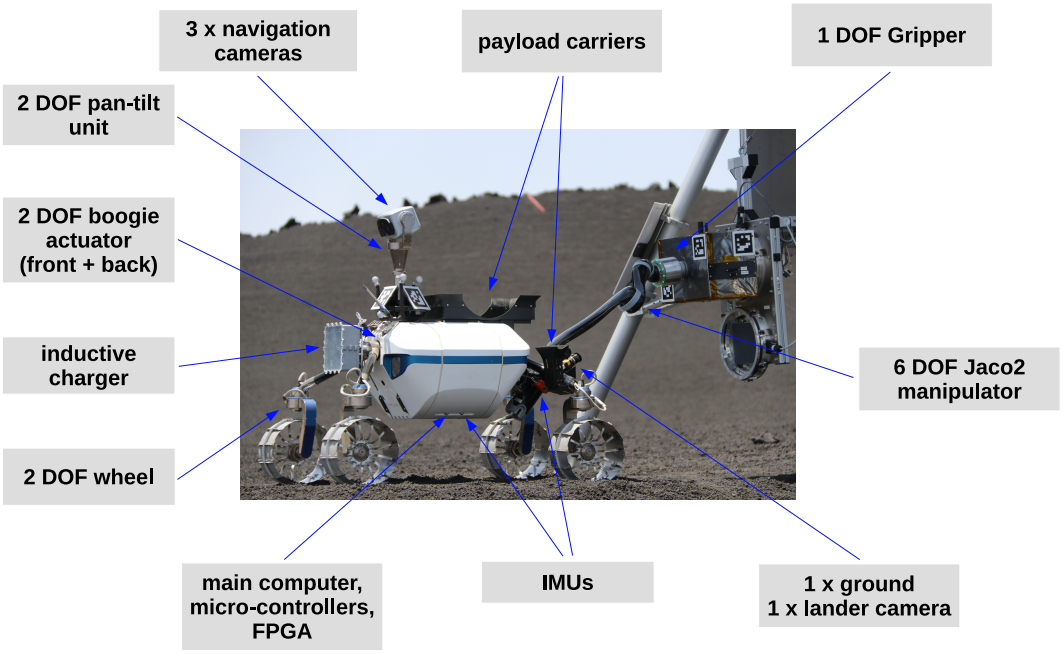


Figure 2.7: *Selected hardware characteristics of the LRU robot.*

Fig. 2.7 shows a picture of the LRU rover including selected properties about actuator and sensor types and positions.

The design goals when building the LRU were to create a small and agile space rover prototype. The length of the LRU is 1.09m and the width is 0.73m. In total, the LRU weights about 43kg. It is able to drive with up to 1.1m/s through rough terrain, using its individually powered and steered wheels. They can be actuated independently from each other allowing for turning in place or driving orthogonal to the robot's main travel direction. The rover's bogie is mounted on two actuated joints that are capable of controlling the bogie's rotational position. This can be used to adjust the center of mass in order to optimize stability and friction in rough terrain and steep slopes. The robot's wheels have been designed on the one hand to passively absorb shocks that occur during movement (by the geometry and material of the spokes) and on the other hand to provide a stiff robot structure for precise grasping.

The LRU is equipped with a Jaco2 manipulator, which provides high dexterous abilities while satisfying LRU's weight constrains (see Schuster et al. (2017)). It only weights 5.4kg and can carry a payload of up to 2.5kg. Custom controllers have been implemented for the manipulator that enable robust in-contact motions (in the face of uncertainties concerning the robot position and its environment) and to follow position-controlled trajectories precisely. A docking interface is mounted at

Table 2.1: *Overview of the DOFs of the LRU robot.*

robot element	**# DOFs** single part	**# installed**	**# DOFs total** total
wheel	2	4	8
bogie actuator	1	2	2
Jaco2	6	1	6
docking interface	1	1	1
pan-tilt unit	2	1	2
			19

the robot's flange. It is capable of docking to a standardized counterpart, which can be mounted on various objects such as payload boxes and scientific instruments, e.g., shovels or geophones (see Chpt. 7.1). Table 2.1 summarizes all DOFs (i.e., all robot actuators) and maps them to the respective robot component.

Table 2.2: *Overview of the sensors of the LRU robot. Low level sensors such as encoders or potentiometers are omitted.*

sensor	**location**	**purpose**	**#**
stereo camera	pan-tilt unit	navigation	1
color camera	pan-tilt unit	navigation	1
ground camera	rear bogie	object detection	1
lander camera	rear bogie	object detection	1
center IMU	body	rover attitude estimation	1
manipulator IMU	manipulator mounting	arm attitude estimation	1
force-torque sensors	Jaco2	force sensing	6
			11

As required for real space missions our robot is only equipped with sensors having low power footprints and offering a high robustness level. In contrast to LIDAR systems, camera-based systems (as used for LRU) are superior in terms of weight, power consumption and robustness concerning heat, vibration and radiation. Thus, our computer vision software components (such as object localization, navigation and mapping) fully rely on mono and stereo camera data. For navigation, a sufficient camera field of view is required to detect obstacles and create a reachability map for local path planning. Thus, the LRU is equipped with a pan-tilt unit able to tilt the front cameras by 90 deg and to pan them by up to 180 deg (90 deg in each direction). For manipulation, two static, high resolution cameras are mounted on the rear part

of the rover. By using those cameras, the robot can detect objects of interest on the ground or higher up at the lander. By using the information of the object detection the LRU can create a model of the robot's surroundings in order to plan collision-free trajectories for object manipulation.

We use several *Inertial Measurement Unit* (IMU)s to estimate the robot's state. One IMU is used in an *Extended Kalman Filter* (EKF) setup for improving the ego motion estimation. The second one is used to determine the direction of gravity, which is needed for the manipulator's impedance controller to compensate the gravitation force during in-contact motions. The LRU needs two IMUs as the kinematics between the rover's main body (where the center IMU is is fixed) and the rear bogie (where the manipulation IMU is attached) is not static. The robot's main sensors are summarized in Table 2.2.

All computations are performed onboard the robot, which enables the rover to perform its task fully autonomously. The main computer is a main-board with an Intel Core i7-3740QM CPU (2.70 GHz) and several interfaces such as *PCI Express* (PCIe), Ethernet, *external Serial AT Attachment* (eSATA) and *Universal Serial Bus* (USB). All main computation nodes (e.g., for navigation, object localization, manipulation planning and mission control) run on this computer. Next to a BeagleBone Black computer board with an *Advanced RISC Machine* (ARM) Cortex-A8 CPU (including a custom cape) for accessing the Jaco2 joints we use a Spartan-6 FPGA board for stereo matching. Finally, we employ an additional computer board with an Intel Atom E3845 CPU (1.91 GHz) running a real-time operating system (RTLinux). It hosts the lower-level realtime controller for the wheels, the bogie and the manipulator.

Although we use modern computation hardware built for terrestrial use (which allows for fast research and development cycles), several parts of the software also run on the Spartan FPGA. For these software parts, the transition to space-proof hardware could be done without much effort as the VHDL code could be easily transferred to radiation hardened FPGAs.

Next to the computation units, also the PCDU was built according to space requirements, such as safety for the mission and safety for the equipment. The unit delivers power to all system components by using several buses (24V and 12V). The PCDU is capable of choosing the best power source in a seamless, uninterrupted manner, i.e., if the voltage of the first battery drops it switches to the second battery without suspending other system components (same holds true if the rover is connected or disconnected from an external power supply).

More information about the hardware layer aspects of LRU can be found in the work by Schuster et al. (2017).

The Software Layers

Another intent of Fig. 2.6 is to show the complexity of LRU's software stack. Three different middlewares are used by the LRU in order to account for various requirements. All three are used for hardware abstraction and for inter-module communication in the functional layer. On top of that, each middleware has dedicated features: *Links and Nodes* (see Florian Schmidt and Robert Burger (2014)) was built for high-frequency realtime-critical communication especially suitable for low-level controller, *Sensornet* (see Schuster et al. (2017)) is used for high-bandwidth camera image transmission and *Robot Operating System (ROS)* offers powerful logging and visualization (e.g., RViz, see Quigley et al. (2009)) capabilities.

The sophisticated control module offers a broad set of controllers ranging from cartesian and joint position controllers to cartesian and joint impedance controllers using torque control via *pulse-with modulation* (PWM). Next to the controller suite for the manipulator there are also controllers and interfaces for the wheels and the pan-tilt unit.

In terms of sensor processing, multiple processes are running for the filtering, the smoothing and the rectification of images. Furthermore, there are modules for feature extraction (which is needed as input for the visual odometry), for stereo matching using the FPGA, and for image stream mapping to align the depth camera of the pan-tilt unit with its color camera. For diagnosis and health check, middleware specific tools are used, e.g., the ROS diagnostics package[3].

The navigation component is another complex module offering a broad set of functionality. It does not depend on any external sensors such as a *Global Navigation Satellite System* (GNSS) but fully relies on visual and wheel odometry fused in a local pose estimator (EKF) employing an IMU. The navigation solution offers both a local obstacle detection, mapping and avoidance and a global navigation solution. For the latter one, we have chosen a *Simultaneous Localization and Mapping* (SLAM) approach, offering global path planning, waypoint navigation and map optimization features (see Schuster (2019)).

The manipulation relies on a motion planner which is based on the *Rapidly-Exploring Random Tree* (RRT) algorithm. It allows to add various constraints to a motion task including geometric, joint and dynamic constraints (see Lehner et al. (2018)). The RAFCON global variable manager is used for resource management and is further described in Sec. 6.2.1. More information about the modules of the functional layer can be found in Schuster et al. (2017).

[3] http://wiki.ros.org/diagnostics

In the executive layer the procedural task control is performed by RAFCON (see Brunner et al. (2016b)). The implementation of the graphical task control language is presented in Chpt. 4. RAFCON's features are also used for the intermediate task memory and part of the experience data logging.

For the *world model*, we employ a graph database based solution, which will be further explained in Sec. 4.4.2.

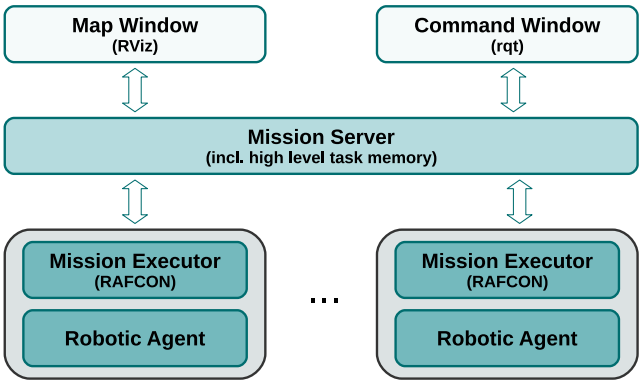


Figure 2.8: *Architecture of the mission control framework ROSMC.*

For manual semantic planning the high level mission control framework ROSMC (see Sakagami (2020)) is used, which is based on ROS's RViz (see Quigley et al. (2009)), the ROS integration for the Qt GUI framework[4] (rqt) and RAFCON (see Fig. 2.8). It enables non-roboticists to control a heterogeneous team of robots with different capabilities. Next to the command window, listing all executed and future actions per robot, the map window allows for easy tracking the robots current state and position on a 3D map. Fig. 2.9 shows a screenshot of the map window, allowing for directly creating new robot actions by interacting with the 3D meshes of the positions of interest (labeled as *Task Markers in the figure*) or the robotic agents. Furthermore, the map window allows for visualizing different map layers such as the 3D map recorded by the robotic agents navigation system and a map indicating the network signal strength. ROSMC maintains the *high level task memory* for tracking all mission-relevant high level data (such as robot status, POIs, ROIs and the status of scientific instruments). Furthermore, it keeps track of all future and already achieved mission goals (and sub-goals).

For automated semantic planning, the RTT planning plugin of RAFCON is employed. As the plugin does not belong to the core of my contribution they are presented in more detail in Appendix D.

The *Links and Nodes* process manager manages all processes including monitoring

[4] `https://www.qt.io/`

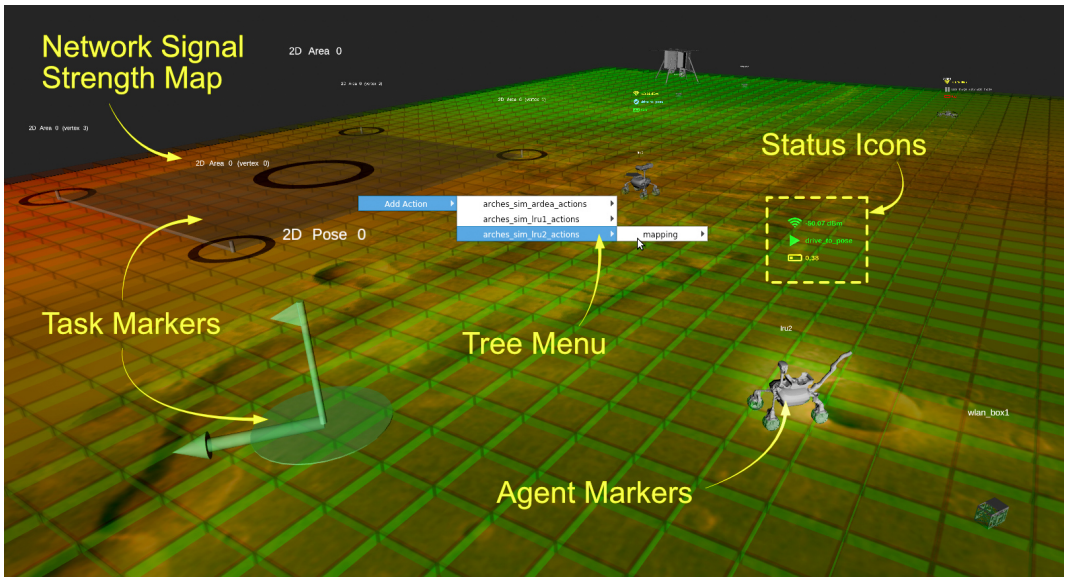


Figure 2.9: *The map window of the ROSMC mission control framework. Different markers offer different interaction possibilities for the user such as placing a region of interest or creating new actions. In this screenshot, two layers blend over each other: the map created by the robot navigation system and a map indicating the network signal strength.*

their startup and runtime-dependencies, checking the process outputs for warnings and errors, and automating startup and shutdown of whole process groups. More information about the manager can be found in the work by Florian Schmidt and Robert Burger (2014).

In summary, more than 100 software processes are executed in parallel on top of the operating system processes. More than 30 developers created the software modules. These aspects motivate a sophisticated versioning and *Continuous Integration* (CI) infrastructure used for guaranteeing consistency of the system (for more information again see Schuster et al. (2017)).

Chapter three

Overview of Task Control Frameworks for Autonomous Robots

In this chapter, I will classify all major task control frameworks of the related work concerning their supported features. The features are selected based on their ability to satisfy task requirements of common application domains. At the end of the chapter, the advantages and disadvantages of graphical task control frameworks will be discussed.

Many papers about various types of task control frameworks have been published so far. I have investigated an extensive amount of related work and compared their concepts. There have also been published several survey papers about this topic, most notably the work by Ingrand and Ghallab (2017). On the one hand, I refer to their work at various places throughout this chapter and will analyze their most important, analyzed frameworks. On the other hand, I skip much purely theoretical work and work only tested in simplified simulations, e.g., blocks world (see Gupta and Nau (1992)) and Wumpus world (see Russell and Norvig (1995)).

First and foremost, I focus on frameworks that have been implemented and tested in real experiments on real robots, or that are very promising and currently under development. On top of that, purely educational frameworks such as Scratch by Resnick et al. (2009) or NXT-G by Kelly (2010) are disregarded as they do not target challenging, real use cases as their coverable complexity is limited. Finally,

I neglect all frameworks, whose concepts are not fully published or whose code is not accessible, e.g., RAPs by Firby (1995), RMPL by Williams et al. (2003), RPL by McDermott (1991), PRS by Ingrand et al. (1996) or COLBERT by Konolige (1997). An exception to this are very popular, commercial frameworks such as Matlab Simulink or LabVIEW.

Table 3.1 gives an overview of all compared task control frameworks including references to both the approach description and one of the most notable experiments, in which the respective task control implementation was used in. The table does not rank the framework in any sense. The order was chosen purely to optimize the presentation and clarity, which becomes more clear in later sections, where I compare these frameworks using various features and keep the exact same order as in this table.

In order to get a reasonable understanding of each framework I compare them using their supported features. The importance of a feature can be estimated by analyzing its ability to solve a common task requirement. The next section analyzes major robot application domains in order to identify such task requirements.

3.1 Deriving Task Requirements from Application Domains

Robots are and will be able to be used in a multitude of domains. In this section I will focus on four major domains which are:

- collaborative industrial production
- autonomous, industrial mobile production
- domestic services
- planetary exploration in space

Collaborative industrial production (see Steinmetz et al. (2018)) refers to future tasks, in which humans and robots cooperatively assemble and create products. From the robot's perspective, this includes tool handling, tool application (like screwing) and lifting of heavy objects in order to help a human to assemble other parts. **Autonomous industrial mobile production** areas are normally controlled environments, where no interaction with humans and other unknown agents is necessary (except for collision avoidance), such as agriculture (see King (2017)) or

Table 3.1: *List of all task control frameworks analyzed this chapter.*

Task Control Framework	Framework Definition Reference	Experiment Reference
Xabsl	Loetzsch et al. (2006)	Loetzsch et al. (2006)
TDL	Simmons and Apfelbaum (1998)	Simmons et al. (2003)
CRAM + openEASE	Beetz et al. (2010)	Beetz et al. (2011)
OpenPRS + IxTeT	Ingrand et al. (1996)	Ingrand et al. (2007)
Teleo-Reactive Programs (TRPs) + T-REX (incl. EUROPA)	Nilsson (1993) + Frank and Jónsson (2003)	Nilsson (1984) + McGann et al. (2008)
SOAR	Laird (2012)	Puigbo et al. (2013)
ROSPlan	Cashmore et al. (2015)	Cashmore et al. (2015)
SmartTCL + SmartMDSD	Steck and Schlegel (2010)	Dennis et al. (2016)
Smach	Bohren and Cousins (2010)	Bohren et al. (2011)
Fawkes + Golog	Niemueller et al. (2009) + Levesque et al. (1997)	Srinivasa et al. (2012) + Hofmann et al. (2016)
YARP-Behavior-Trees	Colledanchise and Ögren (2014)	Colledanchise and Natale (2018)
Robot Task Commander	Hart et al. (2014)	Hart et al. (2014)
ROS Commander	Hai Nguyen et al. (2013)	Hai Nguyen et al. (2013)
FlexBE	Schillinger et al. (2016)	Schillinger et al. (2016)
RoboFlow	Alexandrova et al. (2015)	Alexandrova et al. (2015)
Interaction Composer	Glas et al. (2016)	Glas et al. (2016)
ALICA	Opfer et al. (2019)	Opfer et al. (2017)
Matlab Simulink Stateflow	Mathworks (2019)	Haddadin et al. (2011)
Modelica State Machines	Elmqvist et al. (2012)	Pohlmann et al. (2012)
LabVIEW State Machines	Bitter et al. (2007)	Indrawati et al. (2015)
CS:APEX	Buck and Zell (2019)	Buck and Zell (2019)
Urbi	Baillie (2005)	Kruijff-Korbayová et al. (2011)
Coregraphe	Pot et al. (2009)	Pot et al. (2009)
ArmarX	Wächter et al. (2016)	Wächter et al. (2016)
RoboChart	Miyazawa et al. (2019)	Miyazawa et al. (2017)
RAFCON	Brunner et al. (2016a)	Brunner et al. (2018)

logistics and simple packaging (see Dömel et al. (2017)). **Domestic services** refer to tasks such as cleaning and cooking (see Beetz et al. (2011)). Finally, the domain of **planetary exploration in space** include such scenarios as already presented in the introduction of this thesis, i.e., exploration of unknown terrain, setup of scientific instruments, and sample return missions.

There are many more application domains that are not going to be tackled for various reasons but which can be added to this comparison analogously. **Classical robots in manufacturing**, for example, are already used in industry a lot, but are not interest-

ing with respect to autonomy challenges. **Exploration** in mines or other sub-terrain structures are similar to the space domain as the challenges are complementary, with the exception that a stronger prior concerning the map properties exist. Moreover, **Search** & **Rescue** is very similar to space as the exploration and mapping part can be comparably complex. The additional problems that arise in Search & Rescue can complexity-wise be mapped to related challenges in space: e.g., the problem of blocked passages can be mapped to re-planning in case of impassable terrain and the rescuing of arbitrary objects can be mapped to sampling. Complex properties of the entombed object can be mapped to the selection of complex sample properties. **Autonomous driving** is also not covered as the task complexity is reduced as no manipulation is involved. **Military missions** such as reconnaissance or combat support are not discussed here for ethical reasons.

Different tasks in the mentioned domains demand various requirements to be fulfilled. Among the most important requirements are **efficiency**, **robustness**, **scalability** and **interaction abilities**. The latter one determines the degree of autonomy needed for a task, e.g., if a lot of interaction between the robot and another agent is required we speak of high interaction abilities.

Table 3.2: *Task requirements per robot application domain. The number of + shows the importance of the requirement inside the domain. The mapping from the column colors to the major requirements are: light gray - efficiency; gray - robustness; light teal - scalability; teal - interaction ability*

Application Domain	Parallel Action Execution	Action Pre-Computation	Low Latencies	Error Handling	Preemption Handling	High Redundancy	High Task Complexity	High Task Variability	High Analyzability	Other Agents' Behavior Anticipation	High Level Command Understanding	Explainability of Internal State
Collaborative Industrial Production	+++	++	+++	+	+++	+	++	++	++	+++	+++	+++
Autonomous Industrial Mobile Production	+++	+++	++	+	+	++	++	+++	++	++	+	+
Domestic Services	++	+	++	+	+++	+	+++	+++	+	+++	+++	+++
Space	+++	++	++	+++	++	+++	++	+	+++	+	+	+++

For all analyzed domains, Table 3.2 shows major task requirements needed to be fulfilled by the task control software in order to be used in that domain. All major requirements form their own class and consist of sub-requirements (i.e., members of their class), which are further described in the next section. These sub-requirements help to identify the real needs of tasks in the target domains. Relevant information of the survey paper by Ingrand and Ghallab (2017) was used to perform the domain classification.

For the collaborative industrial production domain, the major requirements are represented by the efficiency class and the interaction ability class. The robustness class is not as important as the human can easily handle a certain degree of fault tolerance as long as safety constraints are obeyed by the robot. Similarly, the scalability class is also not that relevant as in a collaborative setting, the human can perform the more complex actions while the robot helps to hold and lift heavy objects or delivers necessary parts for the next steps.

The requirements for the autonomous, industrial production domain consist of all efficiency requirements. Analogous to the previous use case, the faster the robot can work, the more economical is its purchase. As the robot does everything autonomously, pre-calculation of future actions is possible as the interaction with other agents can often be heavily reduced (compared to the collaborative industrial domain). The latter point is also the reason, why interaction class requirements are not as relevant in this domain.

For domestic services it is the opposite to the previous domain: robots have to interact with other agents a lot but do not have to be as efficient as in industry. If cleaning the floor needs some extra time, this does not mean economic loss immediately.

Finally, in space, the focus lies on task robustness and scalability, while efficiency plays a smaller (but still relevant) role and interaction requirements are clearly the least relevant part.

3.2 Task Requirement Refinement

In the following section I list and elaborate about all sub-requirements for each of the four main requirement classes. Especially possibilities for (partially) realizing them in task control frameworks are given.

Table 3.3: *Task efficiency requirements*

Sub-Requirement	Framework Feature
Parallel action execution	Concurrencies
Action pre-computation	Pre-computation of actions by forward simulation
Executing actions with low latencies	Maximum latency bounds for action selection and execution by supporting realtime capabilities

Efficiency Class: Table 3.3 shows possible task requirements in the efficiency class. In many scenarios parallel action execution is crucial. Thus, a task control framework has to support concurrency patterns. Furthermore, in many situations it might be possible to calculate intermediate results (such as planning obstacle free trajectories or calculating object poses) before their actual use, which leads to improved execution speed and thus a higher efficiency. Finally, realtime support might be needed by some tasks, e.g., for visual servoing.

Table 3.4: *Task robustness requirements*

Sub-Requirement	Framework Feature
Error detection and handling	Error detection and handling; errors modeled as first order objects; includes error taxonomy
Preemption handling	Preemption handling (preemption signal as first order object)
High level of redundancy	Error handling on multiple abstraction levels by error handling forwarding
Behavior verification	Model checking

Robustness Class: Requirements related to robustness are given in Table 3.4. In order to enable prolonged autonomous operations a high robustness level is imperative. Error detection and handling can be fulfilled by a framework if errors are modeled as first order objects. Error detection and handling is partially supported if errors are expressed implicitly by using default events with appropriate parameters. Furthermore, an error taxonomy is required (e.g., by using and extending the error taxonomy of the higher level programming language).
Often, tasks require an agent to preempt its current action and start another one. This is especially relevant, when reacting to external events, dealing with other agents or receiving new commands from a human operator. The framework supports this requirement if preemption is modeled explicitly. It is supported only partially if preemptions are achieved by (ab-)using default or already existing event types.
High redundancy can be achieved by being able to pass an error to various abstraction

levels in case lower abstraction levels cannot handle it. This is called *Error Handling Forwarding*. A framework can support this in a dedicated way or indirectly by using task-dependent, custom parameters or data flows for error forwarding. While this is easy for textual languages (try - catch - except - re-raise patterns) this features is more difficult to support in a graphical manner.
Finally, many critical use cases (as in the space domain) require some system properties to always hold (e.g., enough energy for temperature regulation) or never hold (process starvation). Model checking can be used to solve this challenge.

Table 3.5: *Task scalability requirements*

Sub-Requirement	Framework Feature
High task complexity	Modularity
High task variability	Generality of robotic skills
	Semantic planner integration
High analyzability	Explicit modeling of data flow
	Monitoring capabilities during execution
	Semantic task analysis and profiling

Scalability Class: Scalability is a very broad term and can refer to scalability in terms of complexity, analyzability or implementability. The more complex a task gets (number of actions, parameterization possibilities), the higher the need for modular designs and the ability to write generic skills. In order to tackle highly variable tasks planning components can be integrated or a high degree of skill generality has to be achieved. In terms of analyzability, offline and online debugging and monitoring concepts, semantic task analysis and profiling tools are crucial. Debugging includes the analysis of control flow and data flow, and the online inspection of executed actions. Profiling concepts focus on analyzing action histories in order to find robustness or performance bottlenecks. It mostly requires a descriptive action model which allows for efficient querying and filtering of action types and parameter semantics. For both, monitoring and profiling, graphical tools can enhance the clarity and speed of information retrieval drastically.

Table 3.6: *Task requirements for interaction abilities*

Sub-Requirement	Framework Feature
Behavior anticipation of other agents	Observer structures
Understanding of high level commands	Semantic action abstraction
Explainability of the robot's behavior including its current objectives	Visual or auditive presentation of the robot's internal state and the current actions' parameterization

Interaction Class: As shown in the last section, behavior anticipation of other agents is highly important. This requires the estimation of the internal state (e.g., via Markov Models), which requires the observation of the other agent. This means that a framework at least has to offer observer structures, i.e., parallel execution branches able to monitor certain system states able to preempt the current behavior execution if needed.
Moreover, the understanding of high level commands is crucial, especially if humans with no robotics expertise (non-experts) are involved in the task. For example, the system must be able to process commands such as: "Fetch the blue container from the workbench and bring it to me". The command specification can be done using natural language or using an UI for action selection. The robot must be able to understand the high level command and map it to its own skills.
Finally, if the human's decisions depend on the robot's acting, the robot must be able to publish its internal state (e.g., show a human the progress state of an action). Especially, graphical or auditive feedback (e.g., showing the robot's world model, the execution history or the action parameters) are suitable for showing the robot's internal state to both robotic experts and non-experts.

3.3 Framework Classification by Tasks Requirements

In the following, all task control frameworks listed in the beginning of this chapter are analyzed concerning the features identified in the previous section.

Table 3.7 gives an overview of which task requirements are fulfilled by the respective task control frameworks listed in Table 3.1. Frameworks can support a feature completely (denoted with "x"), partially (denoted with "(x)") or not at all (denoted with an empty space). I do not claim that the requirements or features lists are complete. However, these requirements have been considered in many of the task control frameworks as central aspects. Furthermore, this table can be easily extended if required and all the following examinations and classifications can be performed analogously. As for Table 3.1, Table 3.7 does not rank the frameworks in any manner. Every framework has various advantages and disadvantages and I compare the frameworks only by a subset of all possible features. A detailed discussion of all frameworks is out of the scope of this work. However, I give a more detailed comparison of my approach with the most relevant frameworks in each theoretical chapter and a final summary in Chpt. 7.2.

Table 3.7: *Features supported by different task programming frameworks.*

Task Control Framework	Concurrencies	Result Pre-Computation	Realtime Support	Semantic Error Handling	Explicit Preemption Handling	Error Handling Forwarding	Model Checking	Modularity	Generality	Semantic Planner Integration	Data Flow Modeling	Execution Monitoring	Semantic Analysis and Profiling	Observer Structures	Semantic Action Abstractions	Belief State Visualization	*Visual Task Control Language*
Xabsl				x	(x)	x		x	x			(x)		(x)			
TDL	x		(x)	x	x	x		x	x			(x)		x		(x)	
CRAM + openEASE	x	(x)		x	x	x	(x)	x	x			(x)	x	x	x	x	
T-REX	x		(x)	x	x	x	(x)	x	x	x				x	x		
SOAR	x			x	x	x	(x)	x	x	x				x	x		
ROSPlan				(x)	(x)			x	x	x					x		
SmartTCL	x			x	x	(x)	(x)	x	x	x		x	x	x	x		
OpenPRS	x		x	x	x	x		x	x	x		(x)		x	x		(x)
Smach	x			x	x	x		x	x			(x)		x		(x)	(x)
Fawkes + Golog	x			x	x	x	(x)	x	x	x		(x)		x	x	(x)	(x)
YARP Behavior Trees	x		(x)	x	x	x		x	x			(x)		x			x
Task Commander	x			x	x	(x)		x	x		x	(x)		x		(x)	x
ROS Commander	x			(x)	(x)			x	x			(x)		x		(x)	x
FlexBE	x			x	x	x		x	x		x	x		x		(x)	x
RoboFlow				(x)	(x)			x	x		x			(x)			x
Interaction Composer			(x)	(x)	x			x	x					(x)			x
ALICA	x		(x)	x	x	(x)		x	x	x		x		x	x	(x)	x
Simulink	x		x	x	x	x	x	x	x		x	x		x		(x)	x
Modelica	x		x	x	x	x	x	x	x		x	x		x		(x)	x
LabVIEW	x		x	x	x	x	x	x	x		x	x		x		(x)	x
CS:APEX	x		(x)	x	(x)			x	x		x	x		x		x	x
Urbi	x		x	x	x	x		x	x			x		x		(x)	x
Choregraphe	x			(x)	(x)			x	x			x		x		x	x
ArmarX			(x)	(x)	x			x	x		(x)	x		(x)		(x)	x
RoboChart	x			x	x	x	x	x	x					x			x
RAFCON	x	x		x	x	x	(x)	x	x	x	x	x	x	x	x	x	x

The sheer amount and variety of existing task control frameworks raises the question, why **a new framework had to be created** in the scope of this work. The answer is that none of the available frameworks offered all prerequisites for my work, which include: source code availability, ability for data flow modeling (which is relevant for task analysis and mandatory for autonomous execution optimization) and dynamic behavior modifications during runtime. I discarded most frameworks because of their missing data flow modeling feature. From the remaining frameworks, Simulink, LabVIEW and Task Commander had to be filtered out because of their non-existing or limited access to the source code. As they additionally do not support any features for modeling semantic information they could not serve as base framework for my objectives. This resulted in four remaining framework: RobotFlow, Modelica, CS::APEX and ArmarX. RobotFlow, CS::APEX and ArmarX had not been published at the time RAFCON was invented. Finally, Modelica was not the right tool as it focuses on the modeling of complex systems (containing mechanical, electrical, hydraulic or thermal sub-components), does not offer concepts for creating abstract skill models with semantics and omits dynamic behavior changes during program runtime.

The number of frameworks in Table 3.7 supporting a graphical editor or visualizer is bigger than the number of purely textual frameworks (a "(x)" in the last column means that only a graphical visualizer for the task control language exists but no graphical editor). This could lead to the conclusion that there are more graphical tools for task control than textual ones. That conclusion, however, is not true. There are several reasons for this. On the first hand, as I already mentioned above, I neglected frameworks for which no source code is available. This holds true for many older, textual frameworks dating back several decades. On the other hand, I neglect mostly theoretical frameworks having only be applied on virtual problems (as mentioned above as well). Finally, many frameworks with a graphical component use textual languages or planners (e.g., Golog, EUROPA, IxTeT etc.) to enhance their capabilities. Thus, these textual approaches have not been listed redundantly in a dedicated row. Nevertheless, the trend to more graphical representations cannot be denied. The next section presents main advantages of the graphical approach.

3.4 Motivating the Graphical Approach

In the famous survey paper "Metacognitive Theories of Visual Programming: What do we think we are doing?" Blackwell (1996) refers to much related work, which shows

several advantages of *Visual Programming Languages* (VPL)s compared to textual languages.

VPLs can improve productivity as they are more readable than textual languages, and easy to learn, to write and to understand (see Blackwell (1996)). The learning rate is increased because concrete words (i.e., figures) are easier to remember than abstract ones (i.e., text). Moreover, related work from situated logic shows that there are diagram formats that "increase the specificity in a problem representation " (Blackwell (1996)). There are more examples for these claims given by other authors: Graphical programming for FPGAs can be used by mechatronic system designers and researchers to quickly develop robust and high-performance applications (see Falcon and Trimborn (2006)). Furthermore, Sheppard et al. (1982) highlight the power of hierarchical ideograms, which have a significant positive effect on the performance of programmers on software-related tasks. On top of that, the famous system design platform LabVIEW saves design time when constructing FPGA based hardware tests (see Verret and Thompson (2010)). Finally, VPLs can reduce the complexity of the game development process. "Reducing the complexity of developing games means, among other things, a reduction in the effort required for writing video games. This reduction in effort translates into a shorter time spent programming the game" (Hernandez and Ortega (2010)).

Another advantage of VPLs is that they help to express **problem structure** and thus to get an overview of the problem and the program quickly. If relationships between entities are modeled via lines instead of textual symbols, a developer can follow the routes of control or data flow more easily. Relationships can be modeled more explicit "as locality and topological connection between elements reduce the need to label corresponding items. Reduced labeling accounts for observations that visual representation of an abstract expression syntax can reveal hidden semantic relationships" (Blackwell (1996)).

The next major advantage of VPLs is their ability to use more **dimensions**. These additional dimensions can be used to "convey semantics" (Burnett (1999)). Examples of such additional dimensions are the use of multi-dimensional objects, the use of spatial relationships, or the use of the time dimension to specify 'before-after' semantic relationships" (Burnett (1999)). Especially, spatial relations of icons in *Visual Programming* (VP) help finding particular functions (Blackwell (1996)). VP often enables the modeling in two dimensions by providing a drawing area consisting of a x- and y-axis. This can be superior to text, which can represent information just in one dimension (if indents are not taken into account). Thus, the additional dimension of 2D can be used to visualize complex constructs more easily. However, for presenting highly multi-dimensional problems in an concise, comprehensive way,

one additional dimension of visualization is often not sufficient. “Once a system is sufficiently complex that encapsulation becomes necessary, hidden dependencies are just as problematic in text as in visual languages” (Blackwell (1996)). Yet, taking *Continuous Visual Abstraction* into account (as defined in Sec. 4.2.2 as one of my contributions), unlimited space becomes available for modeling multidimensional symbols, which is a promising approach to reduce this encapsulation problem.

Another superiority of VPLs opposed to textual languages is that they can be more **expressive**. Symbols and pictures contain more information than text does (Blackwell (1996)). Visual blocks are more self-explanatory as they can make use of intuitive icons. The classical saying “A picture is worth a thousand words” does of course not always fit, especially in some strongly formalized contexts such as algebraic expressions. However, there “are strong objections to the idea that the visual sensory modality is some kind of high speed data network” (Blackwell (1996)). The fact that humans prefer pictures to words (Shu (1986)), videos to books (Tanimoto and Glinert (1986)) and graphs to tables (Tanimoto and Glinert (1984)) is another evidence for the superiority of the visual sensory modality. “The superiority of VP languages is most broadly expressed by appealing to the impressive capacity of the human brain for image processing” (Blackwell (1996)). This is supported by Myers (1986), who claims that the human mind is optimized for vision, and Ford and Tallis (1993) making the statement that shapes are easier to process than words. There are other opinions that highlight the expressiveness of textual programming languages, with Lisp being at the forefront (see Jr. and Gabriel (1993)). Nevertheless, VP seems to be a reasonable choice also for Lisp programmers as a famous example of Bresson et al. (2009) shows. They offer a comprehensive development environment with a graphical *Integrated Development Environment* (IDE) based on Lisp called PatchWork Graphical Language (PWGL).

On top of the previously described advantages, VPLs can “represent an existing **mental model** better than text can” (Blackwell (1996)). A good mental model helps to bridge the “semantic gap between the programmer’s conceptual model of what his program is supposed do, and the computational model of the program itself” (Blackwell (1996)). Huang and Cakmak (2015) stress the importance of the mental model of the programmer during programming. They claim that each programmer has to build a mental model of the code. If the model and the code align, they speak of high mental model accuracy, if they do not align, they call it mental model inaccuracy. More evidence of VP supporting the mental model was shown by Navarro-Prieto and Cañas (2001): “In general, the results showed evidence that imagery influences programmers’ mental representations”. Furthermore, they made clear that “images optimize access to semantic information”. In particular, they performed a substantial

amount of research related to spreadsheet programming. They made the discovery that the "mental structure of the spreadsheet programmers had more information about both control and dataflow" than the programmers relying on textual languages.

Finally, Blackwell (1996) makes clear that VPLs help users to spent less energy to get the **syntax** right (as this is performed by the VP framework) such that they can fully concentrate onto the semantics of the program. For example, VPs can free the programmer from writing boiler plate code or from making sure that parentheses match (see Diaz-Herrera and Flude (1980)). Of course, also VPLs need syntax. The "real advance in VP is achieved by assisting the programmer to translate a set of design semantics from one syntax to another" (Blackwell (1996)).

Blackwell (1996) admits that VP is not the 'superlativism' of programming as Green et al. (1991) tries to make clear five years before. Green showed that there are indeed examples in which visual languages are inferior to text languages. It is always a matter of what kind of problem is faced, or in which domain the challenges reside: "Different representations are good for different things" (Blackwell (1996)).

VP always comes with the issue of creating an **IDE** enabling the user to create the Visual Language. This has the advantage that VP allows for programming, debugging and performance analysis in a single consistent framework (Browne et al. (1995)). On top of that, "Visual environments provided a more positive user experience, alongside a reduced perceived workload and higher perceived success" (Booth and Stumpf (2013)). A good example for a VP IDE is FLAIR, a user friendly graphical interface for FLUKA, a multipurpose transport Monte Carlo code, for calculations of particle transport and interactions with matter. According to Vlachoudis (2009) it quickly became very popular, is very robust and stable under the heavy test and greatly increased the learning curve of the new-users. Next to useful features such as auto-complete, IDEs can constrain the programmer during design time in a way, that (common) mistakes are not possible or immediately highlighted, e.g., typos, wrong data types or the use of special operators in a certain context (e.g., in C++ the use of "auto" while iterating over an Eigen data type). Of course, pre-commit checks or compiler warnings can also disclose the existence of errors before runtime but in a subsequent step and not directly during editing.

Apart from these general advantages of visual languages, interesting insights **related to robotics** have already been gained. One example is the graphical programming approach called Graphical State Space Programming (GSSP), which appears to greatly improve the efficiency of code development for typical robot plans, as Li et al. (2011) have examined. Also, flowcharts, which are often (directly or via a dialect) used for control flow design were studied. Curtis et al. (1989) discovered that a

flowchart representation was superior to textual pseudo code when the task involved tracing flow of control. Furthermore, he claims that visuals can be easier to reason about for programs that are related to flow control. Finally, Brooke and Duncan (1980) show how flowcharts help to trace execution flow and localize the area where the bug is located.

Although the advantages of VP for control flow and data flow related task become increasingly obvious, there are many task frameworks for robot control, which do not rely on VP (see Table 3.1). Interestingly, the majority of those 'textual' task programming frameworks use tree-like or flowchart structures for explaining their concepts in their papers. CRAM, for example, uses graphical *fluent networks*, *code trees* or *task trees* (see Mösenlechner (2016), Kazhoyan and Beetz (2018) and Kazhoyan et al. (2018)), Xabsl applies *option graphs* (see Loetzsch et al. (2006)), TDL uses also *task trees* (see Simmons (1988)), and for planning in Golog Hofmann et al. (2016) leverages *assertion graphs*. On top of that, Teleo-Reactive Programs uses *T-R trees* (see Nilsson (1993)) and SOAR uses *subgoal trees* (see Laird et al. (1987)). Thus, it is reasonable to conclude that tree-like structures or flowcharts are a meaningful way to design, structure and program complex robotic behavior.

One question the reader might ask while reading this chapter is: Why do not all task control frameworks rely on graphical programming? There are two striking reasons for this: At the one hand, many people still like textual programming better for various reasons (also based on habits) (see Blackwell (1996) and Green et al. (1991)). On the other hand, the effort for creating a textual DSL versus creating a VPL differs substantially. Defining a DSL in e.g., Lisp is a common practice and straightforward, as a substantial amount of documentation therefore exists (e.g., see Fowler (2010)). In opposite to textual languages, VPLs must ship with an IDE as there are no IDEs for arbitrary VPLs yet. The effort for building and maintaining IDEs for (graphical) programming languages is very high. This is shown by several numbers such as the size of companies selling IDEs (such as MathWorks, Microsoft, JetBrains, Adobe or National Instruments) and especially the number of IDE contributors (such as for Simulink[1], JetBrains' IDEs[2], Eclipse [3] or Visual Studio Code[4]).

[1] https://www.mathworks.com/matlabcentral/answers/contributors/
[2] https://www.jetbrains.com/company/index.html
[3] https://www.eclipse.org/org/foundation/reports/annual_report.php
[4] https://github.com/Microsoft/vscode

Chapter four

Engineering Complex Robotic Behavior: A Graphical Approach

The goal of a task control language is to enable a robotics engineer to effectively program various plans for robots, allowing them to solve complex tasks. As shown in the previous chapters, sequential, parallel and conditional behavior needs to be easily representable with such a language. Easy debugging, error and preemption handling, modularization and generalization capabilities are further task control features such a language has to offer. In this chapter I present a task control language that builds upon the HPFD formalization, which I originally introduced in my previously published paper Brunner et al. (2016b).

4.1 Hierarchical, Parallel, Finite State Machines with Data Flows (HPFDs)

Before presenting the formal definition of HPFDs, I describe their core concepts in a general manner. Thereafter, I highlight the world model architecture for HPFDs, needed to cover complex robotic use cases.

4.1.1 Core Concepts

The core of the task control language are hierarchical, concurrent state machines. Their main elements are shown in the UML diagram of Fig. 4.1. Subsequently, every item of the diagram is described.

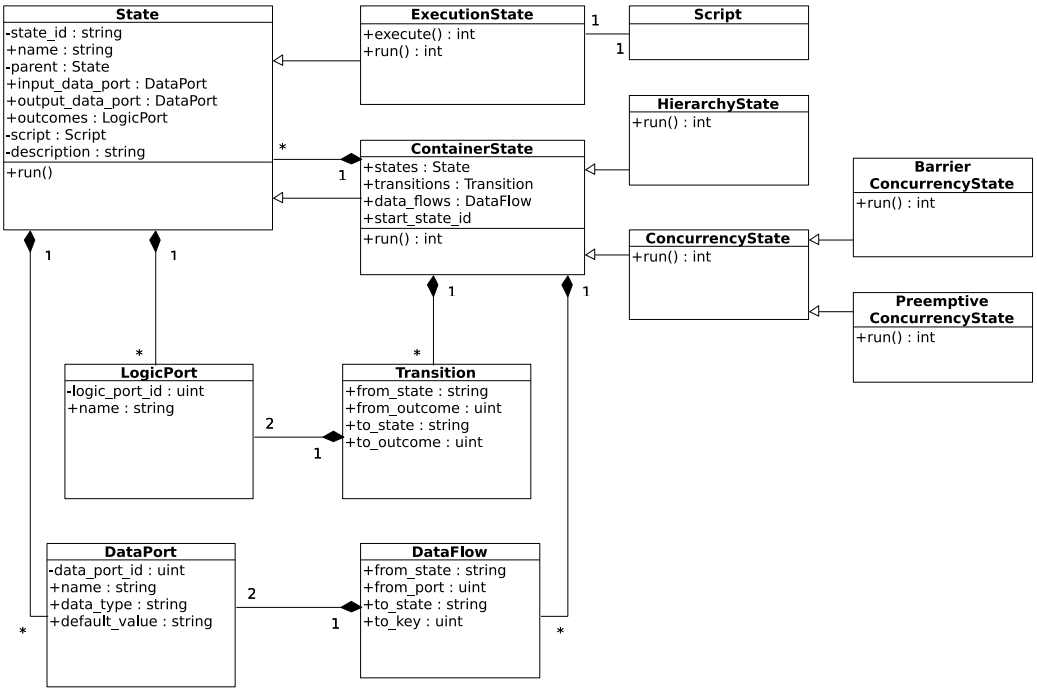


Figure 4.1: *The basic components of HPFD state machines using an UML class diagram*

States A HPFD is a hierarchical structure of states, with a single state at the highest hierarchy level called the *root state*. Basically, there a three different types of states: *HierarchyStates*, *ConcurrencyStates* and *ExecutionStates*.

HPFDs can be directly mapped to trees (see Fig. 4.2 for an example), in which all leaves are ExecutionStates and all non-leaf-nodes are either HierarchyStates or ConcurrencyStates. ExecutionStates do not have any child states but contain the code of a high level programming language (such as Python), which is executed if the state is activated. By using the high level programming language, arbitrary functions such as IO-calls, API-calls or mathematical calculations can be performed.

HierarchyStates belong to the type of *ContainerStates* and can contain child states of any type, which are called *child states* or *children*. The ContainerState is, accordingly, called the parent state of its child states. The main function of HierarchyStates is the modularization of HPFDs into different entities. Each HierarchyState has a

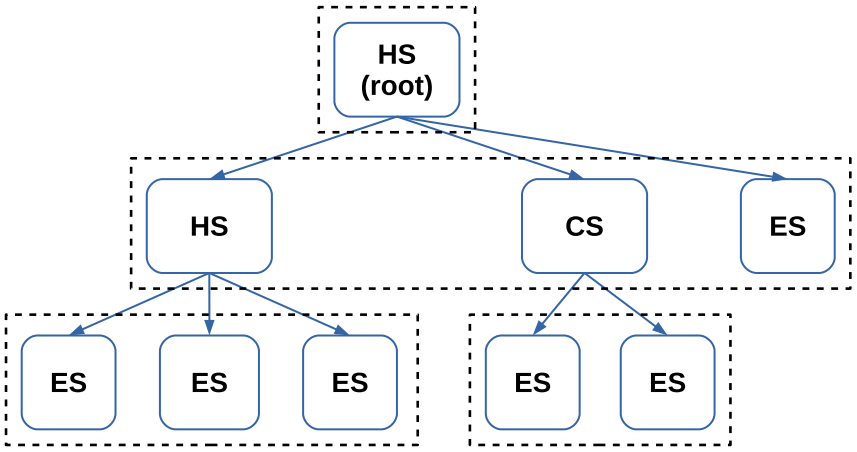


Figure 4.2: *The HPFDs in RAFCON can be represented as a tree. The root state, all direct children of a HierarchyState (HS), as well as each direct child of a ConcurrencyState (CS) can be considered as a separate, flat state machine (dashed rectangles). The leaves of the tree are ExecutionStates (ES).*

single state, which is defined as its entry state. The entry state is the first state to be executed, when the HierarchyState is activated. A hierarchy waits for its currently active child state to be finished and subsequently triggers the next state depending on the outcome of the executed child.

Next to HierarchyStates, ConcurrencyStates belong to the type of ContainerStates as well. All their children are executed concurrently if the ConcurrencyState is activated. *PreemptiveConcurrencyState*s and *BarrierConcurrencyState*s are two manifestations of the abstract ConcurrencyState type. BarrierConcurrencyStates synchronize the execution of their children: all child states have to be finished before the execution can continue. In the case of PreemptiveConcurrencyStates, all their children are executed but as soon as one state finishes the other states are preempted. ConcurrencyStates are needed to execute multiple tasks in parallel. This is often required in robotic applications, if, e.g., some observers track important environment states with various sensors while at the same time the robot navigates from one location to another.

Finally, there is a special type of states called *LibraryState*. It enables the programmer to reuse other state machines by linking them into the LibraryState. This enables to encapsulate functionality in a modular manner and allows for easy collaboration on bigger state machines in larger developer teams.

As the reader may have noticed, a state in the context of HPFDs does not refer to a state in the environment of a robot but to a state of the execution of a task.

Outcomes and transitions Every state possesses several outcomes which can be connected to transitions. When finishing the execution of a state, it returns an outcome-

ID, which exactly maps to one of the defined outcomes. A transition connects this outcome to either another sibling state (i.e., a state which shares the same parent state) or to an outcome of the parent ContainerState. If the transition connects to a sibling state, it is simply executed. If the transition connects to the a parental outcome, then the transition starting at this outcome is followed to the next state or outcome etc.

If a state machine is started from the beginning, the root state is executed first. In the case of an ExecutionState, simply its high level programming language code is executed. If the root state is a HierarchyState, then a so-called start transition selects the first state to execute. In the case of a ConcurrencyState all child states are executed concurrently, as mentioned above. Thereby, outcomes and transitions define the execution order of the state machine, which is called *logical flow* (in opposite to the *data flow*, described in the next section).

Each state has at least three outcomes: a *success*, a *preempted* and an *aborted* outcome. The success is the default outcome of a state and is used if (in the case of an ExecutionState) the internal computations were successful or (in the case of a ContainerState) all child states of one execution branch executed successfully. The developer can add arbitrarily more outcomes to a state, depending on its complexity. The preempted outcome is essential if the state resides inside a PreemptiveConcurrencyState. The first child that finishes execution preempts all sibling states via its parent. If a ContainerState is preempted, all its children are preempted recursively. A preempted state always returns the preempted outcome. In the case of internal computation errors (e.g., division by zero) an ExecutionState returns automatically the aborted outcome.

A programmer can add transitions to the preempted and aborted outcomes if preemption handling or error handling procedures are desired (e.g., to maintain system stability or to shutdown certain components before leaving a state or the state machine). If the preempted or aborted outcome of a state is not connected to a transition, the parental ContainerState is left at the respective outcome, i.e., again either via preempted or via aborted. Thereby, the preemption outcome is propagated up the hierarchy until the PreemptiveConcurrencyState, which is responsible for the preemption, is reached. The error outcome is propagated until either an appropriate signal handler (i.e., a state connected to the respective outcome) is reached at any higher level hierarchy or the state machine is terminated.

Data ports and data flows HPFDs support variables with different visibility, i.e., global, scoped and private. A global central variable store allows access from all hierarchy levels in a thread safe manner. Furthermore, scoped variables are only visible on the

same hierarchy level and are represented by data ports and data flows. States can possess input and output data ports, and an output port can be connected by data flows to one or several input ports of other states. During runtime, concrete values are sent via data flows between ports of different states. Thus, data flows map the value of a so called *source port* to the so called *target port*. As data flows are only allowed between sibling states or between a child state and its parent, single data flows cannot transfer data across several hierarchy levels. Finally, also private data is supported by HPFDs which is only read- and writable by its state.

Data flows are the recommended way to organize the data of a complex task, as they are explicitly visualized next to the transitions between states. The usage of global variables can always be mapped to data flows. However, in some (implementation specific) cases, using global variables is more concise (e.g., for holding global references to logger or communication objects). The differentiation between data flow and logic flow is one of the major strengths of HPFDs and leads to an improved overview. Also, the number and location of data access points can be retrieved in linear time (see Sec. 4.3).

In our experience, trying to maintain the knowledge about the environment (i.e., the world state) in the state machine dramatically increases the complexity of task programming and the system overview is lost soon, even within smaller tasks. Thus, HPFDs have to be extended with an external knowledge base, i.e., a world model, which will be described in Sec. 4.4.

4.1.2 HPFD Formalization

In this section I will provide the formal definition of HPFDs using set theory and predicate logic. This formally distinguishes HPFDs from the other task control languages mentioned in Sec. 3.

The execution semantics of the HPFD definition are similar to those of a *Finite State Transducers* (FST) as defined by Moore (see Xavier (2005) for an overview of state machine dialects): the actions of HPFDs take place inside the state and not while transitioning to another state. One major difference is that in HPFDs, only ExecutionStates perform actions. The results of those actions are mapped to discrete outcomes.

In contrast to the execution semantics, the data handling of HPFDs is comparable with the Mealy FST definition (see Xavier (2005)). However, in the Mealy model, the input data for a state is linked to the transitions. In the HPFD formalization it is

mapped to dedicated data flows, i.e., connections between states to exchange data. Particularly, data is passed from output ports of states to input ports of other states (1×n relation).

According to my prior publication (see Brunner et al. (2016b)) I provide the definition of HPFDs, starting with their prerequisites.

Definition 1 (State) *A state s is an entity having an execution mode and type:* $\text{sem}(s) \in \{\text{active}, \text{inactive}\}$, $\text{type}(s) \in \{\text{execution}, \text{hierarchy}, \text{concurrency}\}$. *The different states are called ExecutionStates, HierarchyStates and ConcurrencyStates, respectively.*

Definition 2 (RAFCON state machine tree (RST)) *A RAFCON state machine tree (RST) is a rooted tree $G = (S, E)$ with $r \in S$ being the root vertex. S is the set of nodes and E is the set of edges which define the tree-order $x < y$ ($\forall x \in S \setminus r : r < x$), following the definition by (Diestel, 2012, p. 13–15). All vertices $s \in S$ are states, therefore r is also titled the root state. All states $s \in S$ with* $\text{type}(s) = \text{execution}$ *must be leaves of the tree. In Fig. 4.2, an example of a RAFCON state machine tree is shown.*

Based on the tree-structure of RSTs, further relationships inside such trees are defined as follows:

Definition 3 (Parent) *In a RST $G = (S, E)$, state $p \in S$ is called a parent of state $s \in S$, if $p < s$ and p and s being neighbors (Diestel, 2012, p. 3) (i.e., share a common edge $e \in E$). This is written as* $\text{parent}(p, s)$. *The short hand notation for retrieving the parent for a state s is $p = parent(s)$.*

Definition 4 (Children) *In a RST $G = (S, E)$, the children of a state $p \in S$, written as* $\text{children}(p)$, *is defined as the set of all $s \in S$:* $\text{parent}(p, s)$.

Definition 5 (Siblings) *In a RST $G = (S, E)$, two states $s, s' \in S$ are said to be siblings (write* $\text{siblings}(s, s')$*), if* $\exists p \in S : \text{parent}(p, s) \wedge \text{parent}(p, s')$. *Note that s and s' can be the same state.*

The logic flow of HPFDs consists of the following two definitions.

Definition 6 (Logical port) *A logical port is a couple $l = (s, t)$ with state s and type $t \in \{\text{income}, \text{outcome}\}$,* $\text{outcome} \in \{\text{aborted}, \text{preempted}, \text{common}\}$. *A logical port with any type of outcome is literally called an outcome.*

Definition 7 (Transition) *A transition is a couple* $t = (l_1, l_2)$ *with* $l_1 = (s_1, \text{outcome})$*,* $l_2 = (s_2, \text{income})$ *both being logical ports and* $\text{siblings}(s_1, s_2)$ *holds.* l_1 *is called the source port,* l_2 *the target port.*

The data handling of HPFDs is described by the following four definitions.

Definition 8 (Data port) *A data port is a triple* $d = (s, t, v)$ *with state* s*, type* $t \in \{\text{input}, \text{output}\}$ *and default value* $v \in A$*, whereat* A *is an arbitrary alphabet.*

Definition 9 (Data flow) *A data flow is a couple* $F = (d_1, d_2)$ *with* $d_1 = (s_1, t_1, v_1)$ *and* $d_2 = (s_2, t_2, v_2)$ *both being data ports, which must fulfill*

$$(\text{siblings}(s_1, s_2) \wedge t_1 = \text{output} \wedge t_2 = \text{input}) \quad \vee$$

$$(\text{parent}(s_1, s_2) \wedge t_1 = t_2 = \text{input}) \quad \vee$$

$$(\text{parent}(s_2, s_1) \wedge t_1 = t_2 = \text{output})$$

Thus, a data flow exists either between two sibling states or between a child and its parent state.

Definition 10 (Data port value) *Each data port* $d = (s, t, v)$ *has a value* a *of a certain alphabet* Σ*:* $\text{value}(d) = a$*. By default,* $\text{value}(d) = v$*. If* $t = \text{output}$ *and* $\text{type}(s) = \text{execution}$*, then the output function* μ_e *determines this value (see Definition 12). If there is a data flow* $f = (d', d)$*, then the value of the source port* d' *defines the value of the target port* d*:* $(\text{value}(d') = a') \Rightarrow (\text{value}(d) = a')$*.*

Definition 11 (Value vector) *A value vector* $V_{s',t'}$ *with state* s' *and data port type* $t' \in \{\text{input}, \text{output}\}$ *is a vector consisting of all values assigned to all ports of that type at that state, thus* $[\text{value}(d_1), \text{value}(d_2), \ldots]$ *for all* $d_i = (s = s', t = t', v_i)$*.*

Based on the definition of states and their inter-relations, the formal notation of HPFDs is provided next:

Definition 12 (HPFD) *A hierarchical, parallel, finite state machine with data flows (HPFD) is a 11-tuple* $(\Sigma, W, G, L, T, D, F, M, S_{exit}, T_{exit}, P)$*, where*

- Σ *is the data alphabet. This is typically* $\mathbb{R}$*, but can also contain, e.g., strings or lists.*
- $W(\tau) : \tau \mapsto \Sigma^n, n \in \mathbb{N}$ *is a vector function representing the robot's world state at time instance* τ*. This vector function serves as access to global variables (see the core section) or sensor readings as well. In robotic contexts,* Σ *is neither finite nor deterministic and thus neither are the elements of* $W(\tau)$*.*

- $G = (S, E)$ *is a RST, with the set of states* S *being finite and non-empty.*
- L *is the set of logical ports, whereat* $\forall s \in S : (\exists! l \in L : l = (s, \text{income})) \wedge (\exists! l' \in L : l' = (s, \text{aborted})) \wedge (\exists! l'' : l'' = (s, \text{preempted}))$. *Thus, all states must have exactly one logical port of each type income, aborted and preempted. Furthermore,* $\forall l = (s, t) \in L : s \in S$.
- T *is the set of transitions, with* $\forall t = (l_1, l_2) \in T : l_1 \in L \wedge l_2 \in L$.
- D *is the set of data ports, with* $\forall d = (s, t, v) \in D : s \in S \wedge v \in \Sigma$.
- F *is the set of data flows, with* $\forall f = (d_1, d_2) \in F : d_1, d_2 \in D$.
- M *is the set of all output functions* $\mu_e : V_{e,\text{input}} \times W(\tau) \mapsto L_{e,\text{outcome}} \times V_{e,\text{output}}$ *for the ExecutionStates* $e \in S$. *Here,* $V_{e,\text{input}}$ *and* $V_{e,\text{output}}$ *are the input respectively output value vectors of* e *and* $L_{e,\text{outcome}}$ *is the set of all logical ports* $l = (s = e, t = \text{outcome}) \in L$. *The output functions have access to the current world state* $W(\tau)$. *Together with the input port values, they determine the outcome port and the values assigned to the output ports. The evaluation of an output function takes a certain amount of time* $\Delta\tau > 0$. *When the output function* μ_e *returns a value then* e *is said to be* executed.
- $S_{\text{entry}} \subset S$ *is the set of all start states. A start state is the state evaluated first when executing a HierarchyState. Thus, for each HierarchyState* $h \in S$, *there is exactly one state* $s \in S_{\text{entry}}$ *with* $\text{parent}(h, s)$.
- T_{exit} *is a set of couples* $t_{\text{exit}} = (l_1, l_2)$ *with* $l_1 = (s_1, t_1 = \text{outcome}) \in L$ *being the source outcome,* $l_2 = (s_2, t_2 = \text{outcome}) \in L$ *being the target outcome and* $\text{parent}(s_2, s_1)$ *holds. If the* execution *reaches an outcome being the source port of a couple* $t_{\text{exit}} \in T_{\text{exit}}$, *then the state* s_2 *of the target outcome is left at this target outcome* l_2.
- P *is the set of all outcome functions* $\rho_c : \mathbb{P}(L_{\text{children}(c),\text{outcome}}) \mapsto L_{c,\text{outcome}}$ *for all concurrency states* $c \in S$. *Here,* $\mathbb{P}(L_{\text{children}(c),\text{outcome}})$ *is the power set of all outcomes of all* $\text{children}(c)$ *and* $L_{c,\text{outcome}}$ *is the set of all logical ports* $l = (s = c, t = \text{outcome}) \in L$. *If one child state of* c *exits (*c *being a PreemptiveConcurrencyState) or all children of* c *have exited (*c *being a BarrierConcurrencyState), then* ρ_c *determines the outcome of* c *on which* c *is exited.*

HPFDs have the following properties:

- *If a HierarchyState is* executed, *then the execution directly starts with the entry state defined by* S_{entry} *and finishes when an exit outcome defined by* T_{exit} *is reached.*

- *Initially, all states are inactive:* $\forall s \in S$, $\text{sem}(s) = \text{inactive}$. *When a state* s *is entered, meaning that the execution reaches an income* $l = (s, \text{income}) \in L$, *then* $\text{sem}(s) = \text{active}$. *Accordingly, when a state is left, meaning that the execution reaches an outcome* $l = (s, \text{outcome}) \in L$, *then* $\text{sem}(s) = \text{inactive}$.
- *If a ConcurrencyState is entered, all of its children are entered in parallel. Thus, the execution splits up. The outcome function* ρ_c *defines on which outcome to leave the ConcurrencyState.*
- *Every port* $l = (s, t = \text{outcome}) \in L$ *can only be the source of either one* $t \in T$ *or one* $t_{\text{exit}} \in T_{\text{exit}}$.

With these definitions deadlock detection and reachability analysis are possible (see Sec. 5.3.1). Furthermore, it enables comparisons with other state machine approaches.

At a closer look, HPFDs are a super-set of FSTs. This means that any behavior modeled using a FST can be translated to a HPFD. FSTs are sextuplets $(\Sigma, \Gamma, S, s_0, \delta, \omega)$, whereat Σ is the input alphabet, Γ is the output alphabet, S the set of states, s_0 the initial state, δ the state-transition function and ω the output function. The elements of FSTs have the following correspondences to HPFD elements:

- Σ: the set of state outcomes activated by M
- Γ: Σ (or a subset of it)
- S: the set of states of the RST G
- s_0: the root state of the HPFD RST G
- δ: the set of transitions T
- ω: the output data generating part of M, i.e., $V_{e,output}$

The key point here is that the input alphabet Σ, which equals the external events triggering the automaton, is mapped to the activation of outcomes by the set of HPFD output functions M. All μ_e of M can use the world model for calculating the outcome to activate. The world model represents (next to the robot's system state) also the robot's sensor data and is thus source of non-determinism (equivalent to the non-determinism of external events of a FST).

4.2 The HPFD Graphical Language (HPFD-GL)

We invented HPFDs for graphical programming, which offers many advantages compared to textual programming (as shown in Sec. 3.4).

4.2.1 The HPFD-GL Graph Grammar

Based on HPFDs I have defined a graphical programming language called HPFD-GL. Its main purpose is to be a DSL for defining robot behavior. Fig. 4.3 shows the graph grammar of HPFD-GL (for more information about graph grammars see Zhang (2010)). It provides a mapping from all essential textual instructions needed to program complex control flow to their graphical equivalent in HPFD-GL. Thus, to the best of my knowledge, all instructions known from other task control languages such as RMPL (see Effinger et al. (2010)) or TDL (see Simmons and Apfelbaum (1998)) could be expressed in HPFD-GL.

All supported instructions shown in Fig. 4.3 are explained in the following. Although HPFDs consist of states, which execute actions, I just refer to those as *actions*.

1. *Sequences* execute actions one after another.
2. *Conditions* decide between several options; switch statements are supported in an equivalent fashion.
3. *While-Loops* execute an action as long as a certain conditions holds; both while patterns, do-while and while-do, are supported.
4. *For-Loops* run an action multiple times (limited by a maximum loop counter).
5. *Concurrencies* run actions in parallel until all actions finish execution.
6. *Preemptive Parallelisms* execute several actions in parallel. If one of the actions finishes execution all other actions are preempted.
7. *Try-Catch* statements run a certain action until it either succeeds or fails with an exception. In the latter case they catch the exception and run a configurable error handling procedure.
8. *Event Handling* and *Monitors* are implemented using PreemptiveConcurrencyStates An observer for a specific event is registered in a parallel branch inside

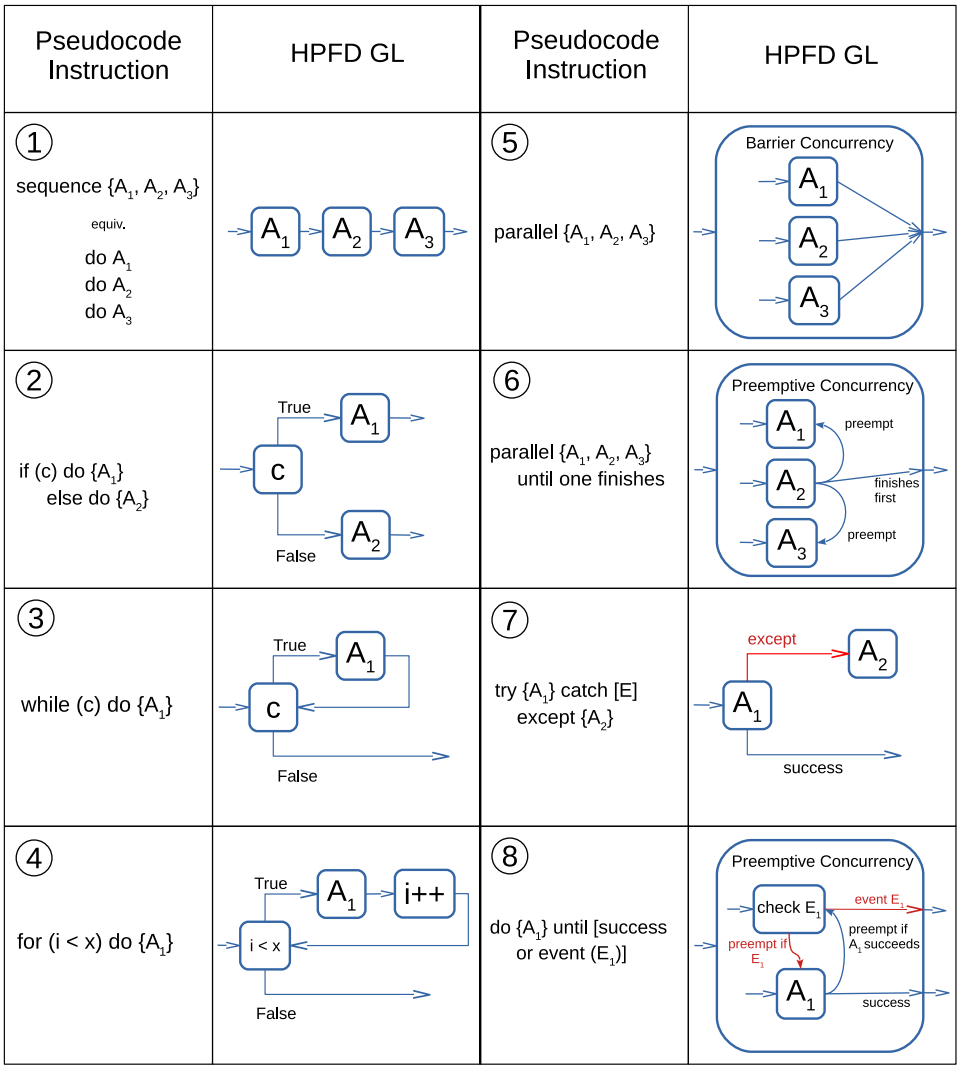


Figure 4.3: *Mapping of pseudo-code instructions to HPFD-GL. Boxes represent HPFD states, arrows transitions. Text boxes are used to add state type annotations or outcome information.*

the PreemptiveConcurrencyState and preempts the nominal action in the case of the event.

In the explanations an *action* refers either to a single action or a group of actions encapsulated in a container state. As it is the case for ConcurrencyStates (see 5 in Fig. 4.3), a container state groups other states and is visualized as a box around its child actions.

All entities in HPFD-GL have a variable position and a size. Ports (incomes, outcomes, input and output ports) only have a position on the border of the state. Child states are constrained to to be smaller than their parents. Additionally, their position must

not be outside their parent state, nor must their borders exceed the limits of the borders of the parent state.

Another feature of HPFD-GL not mentioned so far are libraries. Libraries encapsulate one or more actions and can be linked to another action inside, e.g., a sequence, concurrency or hierarchy. If the content of a library changes, then those modifications are reflected in each location, where the library is used.

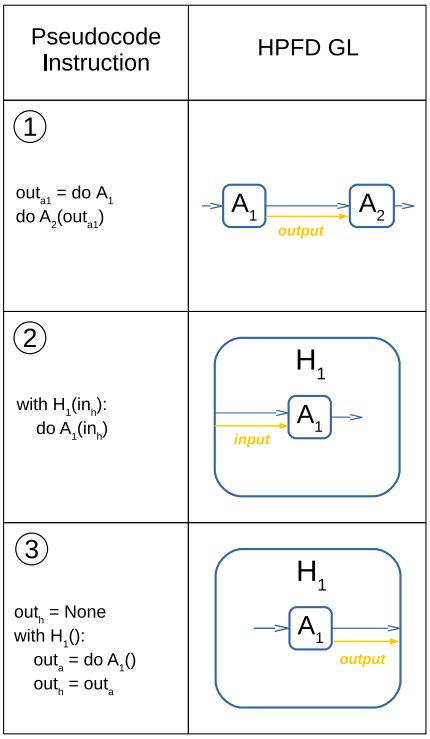


Figure 4.4: *Mapping of pseudo-code instructions to HPFD-GL. Boxes represent HPFD states, blue arrows transitions and yellow arrows data flows. Text boxes are used to add outcome information.*

Fig. 4.4 shows the graph grammar of HPFD-GL related to data flow. The single elements have the following meaning:

1. *Output-Input Forwarding*: state A_1 forwards its output data out_{a1} to the state A_2.
2. *Input-Input Forwarding*: HierarchyState H_1 forwards its input data in_h to the state A_1.
3. *Output-Output Forwarding*: HierarchyState H_1 forwards the output out_a of its child state A_1 up into the scope of its parent.

The support for error and preemption handling is not only deeply rooted into the core

design of HPFDs but also into its graphical language. Fig. 4.5 shows the concepts of error handling as implemented using HPFD-GL (preemption handling is modeled analogously). Every state has an *aborted* (red) and *preempted* outcome (blue). To create error and preemption handling routines other states can be connected to these outcomes: this is called explicit error/preemption handling. If no transition is connected to the outcome, then, in the case the state fails (abortion) or the state is preempted (preemption), the execution is forwarded to the parent's outcome of the same type: i.e., the aborted outcome of a state is forwarded to the state parent's aborted outcome and the preempted outcome to the parent's preempted outcome. At the parent's outcome an error/preemption handling routine is then able to handle all abortion/preemption events originating in the parent. This second case is called implicit error handling.

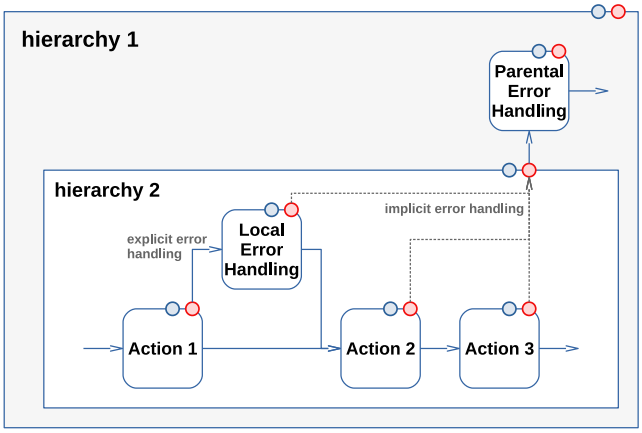


Figure 4.5: *The figure shows the two types of error handling procedures: implicit and explicit error handling. The aborted outcome, to which the error handling routine connects is visualized in red. The blue outcome shows the preempted outcome. Analogously to error handling, preemption handling can be defined in an implicit and explicit manner as well.*

4.2.2 Continuous Visual Abstraction

Visual programming is a kind of programming, in which multiple dimensions (space, time, multi-dimensional objects) are used to "convey semantics" (Burnett (1999)). Especially, spatial relations help to gain an overview and to localize certain functions or problems easier (Blackwell (1996)).

Two dimensional drawing areas are superior to one-dimensional text in many aspects (Blackwell (1996)). However, for presenting complex multi-dimensional problems in a concise way, one additional dimension of visualization might not be sufficient. "Once a system is sufficiently complex that encapsulation becomes necessary, hidden dependencies are just as problematic in text as in visual languages" (Blackwell (1996)).

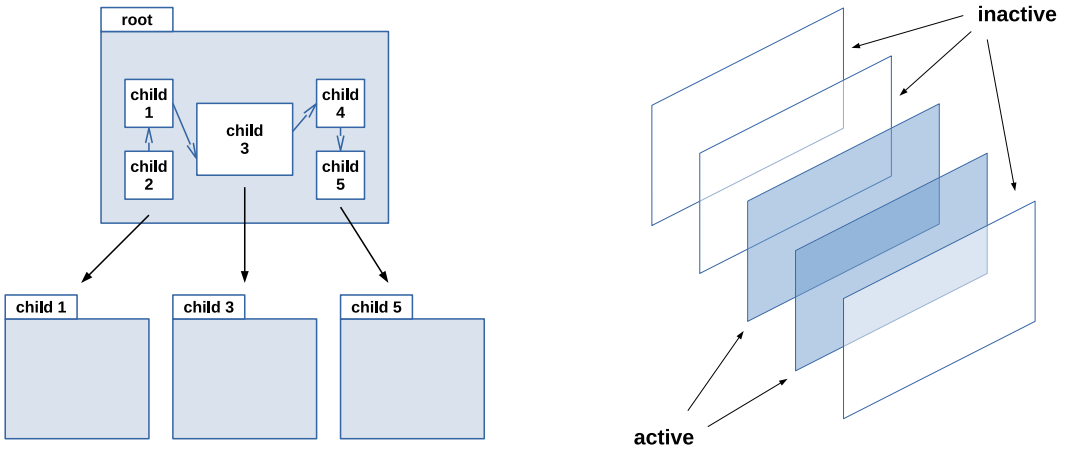


Figure 4.6: *Visual abstraction approaches*

One way to reduce the encapsulation problem is to use different views for each encapsulated module or node, as shown in the left side of Fig. 4.6. This strategy is pursued, e.g., by MATLAB (2018) Simulink. The main drawback of this approach is that for complex systems the user ends up having dozens of open windows, one per node that is currently either being under inspection or under modification by the user.

Another approach is to use several layers for visualizing complex content, such as a digital map. This strategy, which is visualized on the right side of Fig. 4.6, is pursued by Watanabe et al. (2006) for Google Maps[1]. However, the layers to display are calculated by a function only dependent on the zooming level of the user, not on the visualized content. This is disadvantageous if various data with various

[1] https://www.google.de/maps

importance levels has to be visualized and a global setting for "enabling" or "disabling" certain levels does not account for the heterogeneity of the data. Especially, if some information entities are too small, adding additional content does not make sense. On the other hand, if some entities are coarse or big enough, additional content can already be included to enhance the current view.

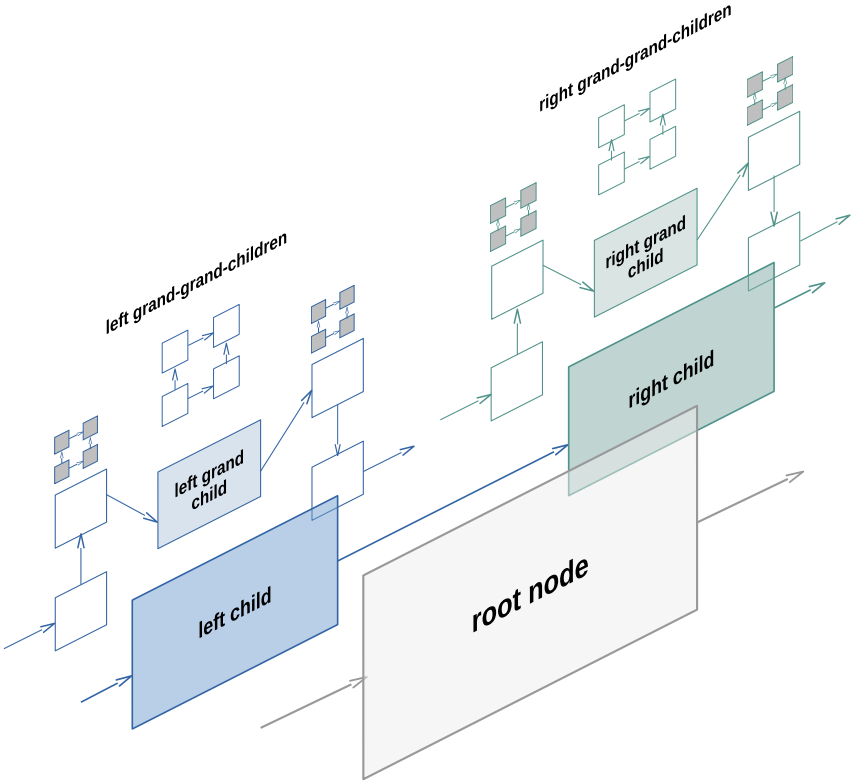


Figure 4.7: *The model of Continuous Visual Abstraction*

I propose a new methodology, called *Continuous Visual Abstraction*, in order to reduce the mentioned encapsulation problem. This approach uses zooming as well, but renders content as a function of both the zooming level and content size. Fig. 4.7 shows how the content (in my case instances of HPFD-GL) is modeled. Each hierarchy level is mapped to a dedicated layer in the continuous visualization model. The number of a layer l relative to the top layer is called the depth d_l of the layer l. The depth d_s of a state s is the same as its hierarchy level and thus d_l. The current zoom level c_z of the user is always in the range of $[0, max\_zoom]$, with max_zoom defined as

$$max\_zoom = x \times d_{max} \tag{4.1}$$

x is the zoom granularity, i.e., how many zooming operations have to be performed, until the max_zoom level is reached. It must be possible to configure this value depending on the scenario and user preferences. The current zoom depth c_d is defined as

$$c_d = \frac{c_z}{x} \tag{4.2}$$

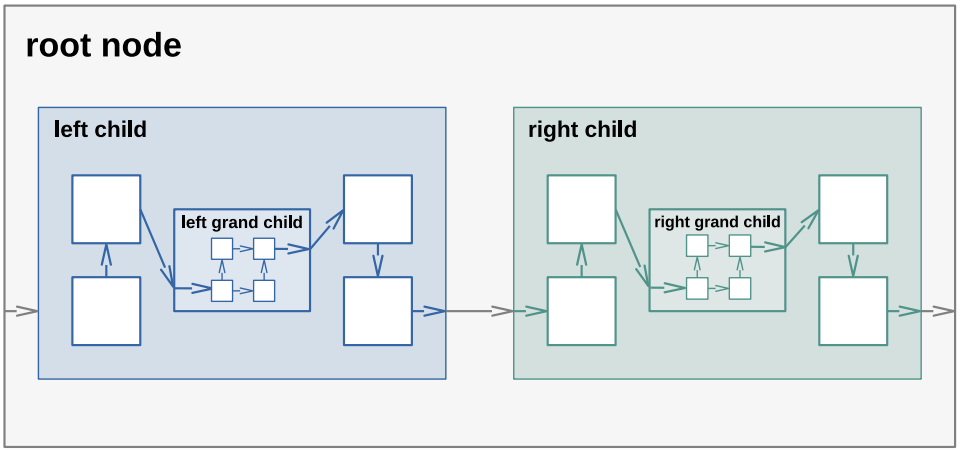


Figure 4.8: *A visualization example of Continuous Visual Abstraction. The example is based on the model shown in Fig. 4.7*

The state score s_s defines if a state content is visualized:

$$s_s = (d_s - c_d) \cdot y + size_s \cdot z \tag{4.3}$$

In the equation above, $size_s$ is the size of the state s, y is a linear factor to what extent the state's depth is taken into account and z is a linear factor for the state's size. If the state score is bigger than a predefined threshold factor t, then the state is visualized, if it is smaller, then the state is not rendered. If s_s is smaller than $size_s \times z$ (i.e., c_d is bigger than d_s) then the current zoom depth exceeds the state's depth (i.e., the user already zoom through the state) and thus the state is not rendered as well.

Based on the example model of Fig. 4.7, Fig. 4.8 shows the visualization for the user. Only the scores of the colored states exceeds the predefined threshold and thus their content is shown. All white states do not have a high enough score for their content (shown in gray in Fig. 4.7) to be rendered.

This approach to Continuous Visual Abstraction implies that by zooming in, more

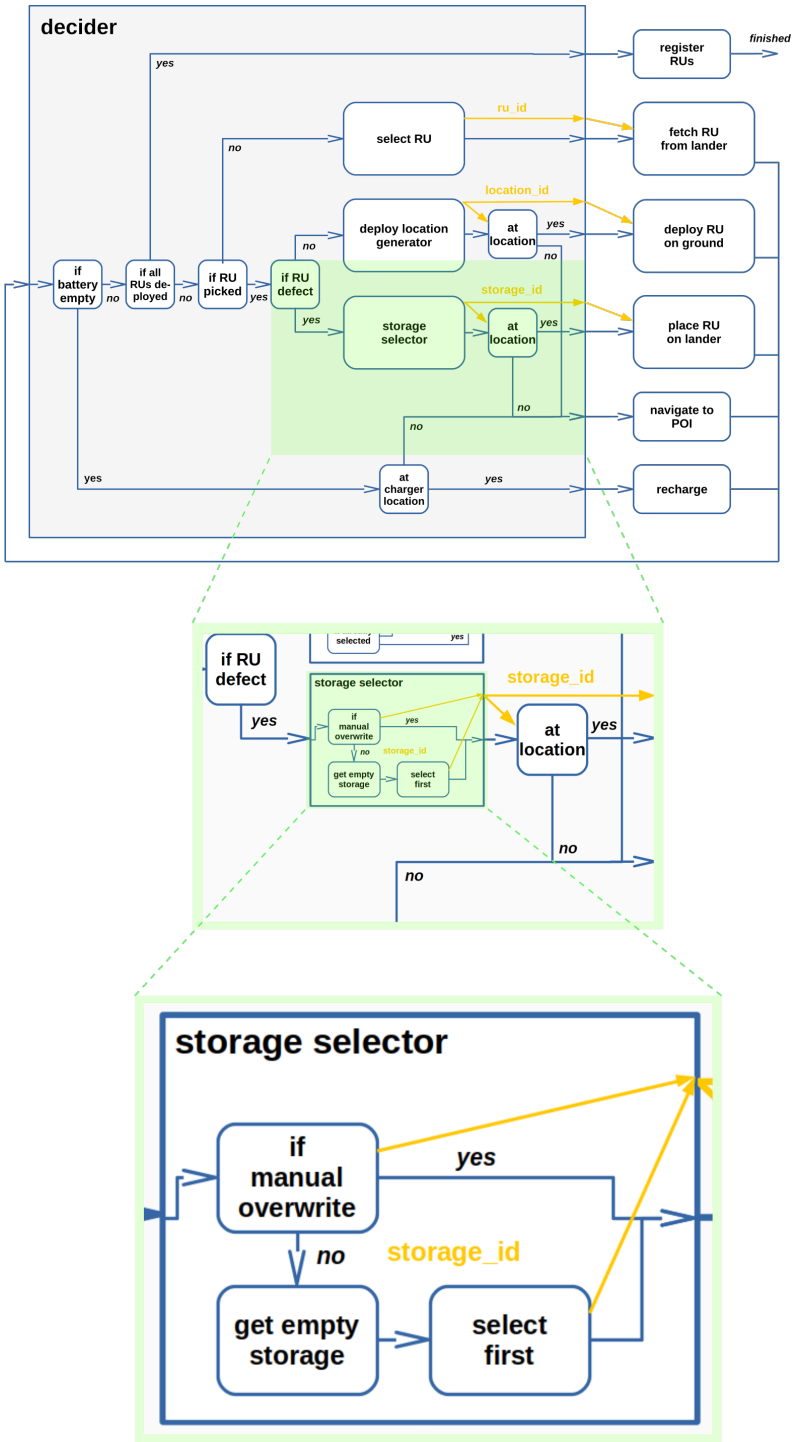


Figure 4.9: *HPFD-GL decision tree example of Fig. 4.19 with various zooming layers.*

information at the focused area can be revealed and information outside the field of view is hidden (concretion). Opposed to that, if the user zooms out, the specificity of a complex entity is hidden in order to gain overview of higher order concepts

(abstraction). Fig. 4.9 shows the continual visual abstraction applied to a HPFD-GL decision tree example explained in the next section (see Fig. 4.19). Here the user zooms three times towards the “storage_selector” state, until only this state is visible.

By using this approach, unlimited space becomes available in 2D for modeling of HPFDs, offering easy ways for encapsulation, concretion and generalization.

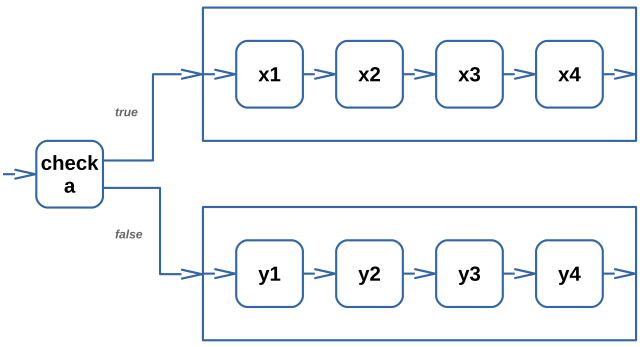
4.3 HPFD-GL Examples

In the following, I give several examples in order to show the applicability of HPFD-GL. I divide the examples into the two classes “control flow examples” and “data flow examples”. For both classes, I show abstract examples and robot specific ones.

4.3.1 Abstract Control Flow Examples

Algorithm 1 Abstract sequence example, left in pseudocode and right in HPFD-GL.

```
 1: def main():
 2:     if a is True then
 3:         do sequential:
 4:             do x1(params)
 5:             do x2(params)
 6:             do x3(params)
 7:             do x4(params)
 8:         end
 9:     else
10:         do sequential:
11:             do y1(params)
12:             do y2(params)
13:             do y3(params)
14:             do y4(params)
15:         end
16:     end if
17: end
```

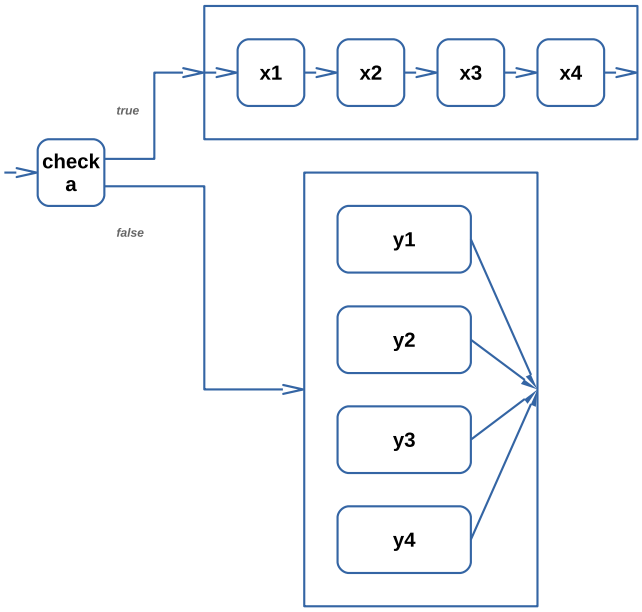


The first two examples show basic control flow structures. Fig. 1 depicts a simple conditional sequence. On the left side of the figure the textual representation in pseudo code is given, on the right side the visual representation in HPFD-GL is shown. Both sequences consist of four actions. In Fig. 2 either a sequence or a concurrency is executed depending on a condition check. Again, the left side of the figure shows the

textual representation in pseudo code, the right side shows its visual representation in HPFD-GL. The additional "params" keyword is added to each action in order to highlight that each action is normally executed with various parameters, which are neglected in these figures in order to be able to concentrate on the control flow.

Algorithm 2 Abstract concurrency example, left in pseudo code and right in HPFD-GL.

```
 1: def main():
 2:     if a is True then
 3:         do sequential:
 4:             do x1(params)
 5:             do x2(params)
 6:             do x3(params)
 7:             do x4(params)
 8:         end
 9:     else if a is False then
10:         do in parallel:
11:             do y1(params)
12:             do y2(params)
13:             do y3(params)
14:             do y4(params)
15:         end
16:     end if
17: end
```



The figures show that both, the textual and visual, representations are concise and simple. However, the analyzability of the visual representation is clearly superior. At the first glance the textual presentation of both programs does not differ at all. On closer examination the second program features a "parallel" block in the second conditional branch, leading to a fundamentally different behavior than the first program. In the case of the visual HPFD the difference of the programs become clear at the first glance, as the parallel states are ordered vertically instead of horizontally and their outcome transitions are merged. The two examples introduce the reader to the fact (see Curtis et al. (1989)) that visual representation is superior to textual one in terms of representing control flow, especially if the task is related to tracing the flow of control. As Navarro-Prieto and Cañas (2001) and Blackwell (1996) state, this also holds true for data flow.

The interested reader might argue that the code in lines 3 - 7 and 10 - 14 could be wrapped into dedicated functions. At the top of these functions the *sequential* or *parallel* modifier would then be more obvious. Although this might improve the overview in this simple case, it is not possible for more interwoven problems. In

fact, it leads to a new problem of textual programming of complex behavior: the problem of the source code locality not being aligned with the semantic locality of the program (see Larkin and Simon (1987)).

Algorithm 3 Abstract decision tree example in pseudo code.

```
 1: def do-check-g():
 2:     if g is True then
 3:         do x2(params)
 4:     else
 5:         do x3(params)
 6:     end if
 7: end
 8:
 9: def main():
10:     if a is True then
11:         if b is True then
12:             do w1(params)
13:             do w2(params)
14:             if f is True then
15:                 do w3(params)
16:             else
17:                 do do-check-g()
18:             end if
19:         else
20:             do x1(params)
21:             do do-check-g()
22:         end if
23:     else
24:         if c is True then
25:             if d is True then
26:                 if e is True then
27:                     do do-check-g()
28:                 else
29:                     do y2(params)
30:                 end if
31:             else
32:                 do y1(params)
33:                 do y3(params)
34:             end if
35:         else
36:             do z1(params)
37:             do z2(params)
38:         end if
39:     end if
40: end
```

This leads us to the next example, given in the form of a decision tree. Fig. 3 shows the textual representation of a decision tree. While the indenting helps to identify different logical blocks, it is hard to get an overview of the overall program. Especially finding all places where the common “do-check-g” function is called is cumbersome and requires to scan the whole program. Ultimately, restructuring the program by modularizing it and splitting it up into smaller functions does not help at all. The

additionally introduced function signatures would just add to the total number of lines of code, which had to be searched through in order to find the usages of the "do-check-g" function. Of course, using text based search to find the occurrences is still possible, however locality is lost (see Larkin and Simon (1987)) and gathering the whole context of the "do-check-g" function invocation stays unnecessarily complex.

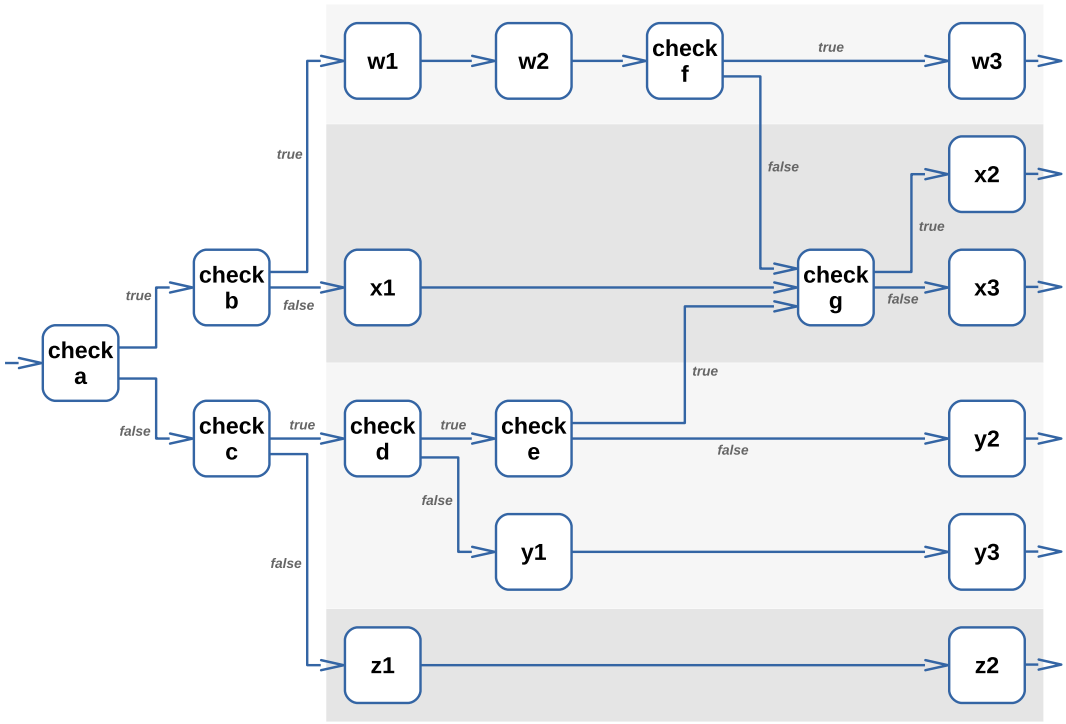


Figure 4.10: *Abstract HPFD-GL decision tree example.*

Fig. 4.10 shows the same decision tree using HPFD-GL. The overall structure of the program consisting of four main branches can be identified easily. The predecessors and successors of the "check g" state (which maps to the "do-check-g" function in the textual representation) are determined quickly. This example supports the claim of Blackwell (1996) that locality and topological connection of visual languages help the developer to easily follow the routes of control (or data) flow.

4.3.2 Robotic Control Flow Examples

Next to the abstract control flow examples of the previous section, this section shows examples based on the validation experiments of Sec. 7. Fig. 4.11 shows a robotic behavior, partly implemented in the AIMM experiment (Sec. 7.1.2). After the initialization of the system and the robot's belief state, the robot's main task

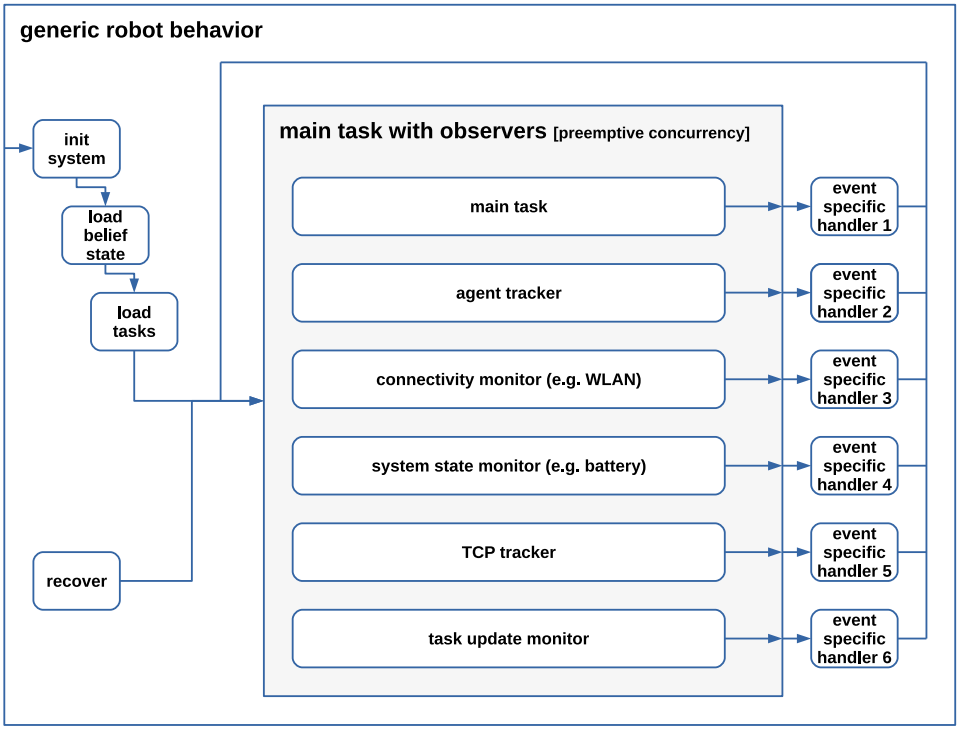


Figure 4.11: *HPFD-GL observer structure example.*

is executed. In parallel to the task execution, several observers monitor different internal or external environment states. The observers communicate with the other parallel layers by either populating the robot's belief state or using signals or events (via global variables). In the case of an emergency they can directly preempt the other layers. This may be the case if another agent, e.g., a human, interferes with the robot and forces it to pause or re-plan. Other external events would be connection loss or a task update request by a higher level scheduling component. Internal states are also of high relevance to the robot. First and foremost, the energy supply must always be ensured. On top of that, also the monitoring of processes and the checking of is-alive status messages from different components have to be performed. Next to such kind of health data, the data logging has to be actively triggered. The TCP during manipulation can be of vital interest for manual monitoring or automated failure detection. Thus, instead of idling, the pan-tilt unit of a robot can record the TCP during manipulation (the benefits of TCP tracking are highlighted in Sec. 7.1.2).

The final control flow example shows a decision tree based on the ROBEX planetary exploration mission scenario presented in Sec. 7.1.1 in more detail. The scenario is sketched in Fig. 4.12. The goal for LRU is to deploy four *Remote Unit* (RU)s to a target area next to the lander platform. The exact deploy locations can be chosen by the robot. RUs are boxes able to carry different payload, in this case seismometers.

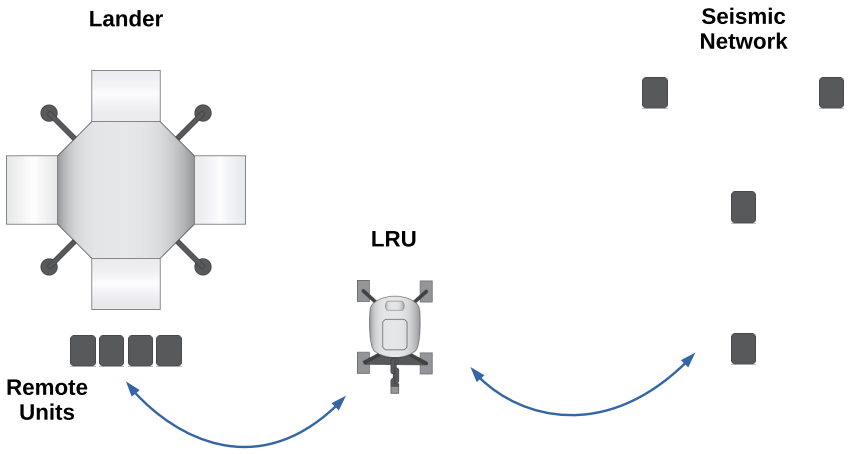


Figure 4.12: *The seismic network mission of the ROBEX experiment.*

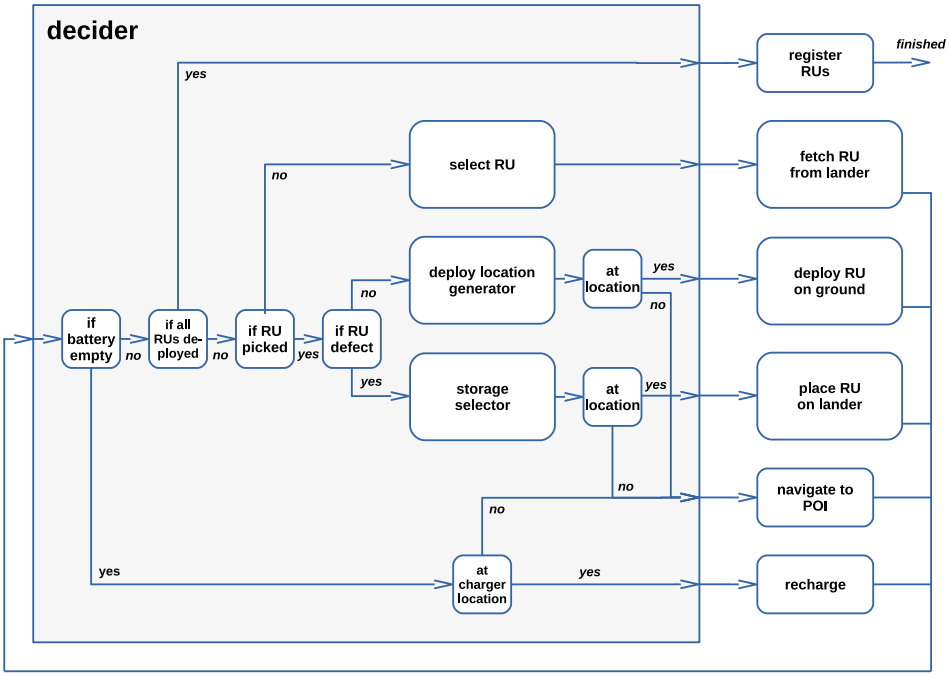


Figure 4.13: *High level view onto a HPFD-GL decision tree able to solve the mission sketched in Fig. 4.12.*

The lander has several storages, in which RUs reside. They have an active lock, which is released once the rover grasped the RU. The lock can get stuck, forcing the rover to release the RU and select another one. On placing a RU on the ground, the rover can test its proper deployment by triggering a seismic test event, by carefully punching on the ground with its manipulator. In the case the seismometer does not sense the test impulse, the RU is picked up again and either re-placed at a nearby location or placed back into the lander for remote maintenance.

Fig. 4.13 shows the appropriate decision tree for the rover in order to fulfill the mission: The first condition checks if the battery of the rover is still charged, so that the rover can execute its next action. If not, the rover drives to the lander for recharging. The second condition checks if all RUs have successfully been deployed. In that case the rover registers the RUs and estimates their relative positions by mapping the deployment area thoroughly. The next condition checks if the robot has already picked a RU. If not, the rover selects a RU at the lander and picks it up. If the rover has already picked up a RU, and the RU is not defect, the LRU deploys the RU at a target location. If the RU is defect (because the LRU picked it up from the ground during its last action as the seismometer was not responding) the rover brings it back to the lander for remote maintenance. The decider shows the control flow in a clear manner. The decision tree is built in a horizontal manner and is propagated from left to right. The decider is, however, not complete as the data flow is missing. I will introduce the data flow examples in the following section and I will conclude the next section by referring back to this example and complementing it.

4.3.3 Abstract Data Flow Examples

Fig. 4.14 shows different use cases for the modeling of data flow. It can be used to track the data flow throughout various hierarchy levels. This especially is supported by the Continuous Visual Abstraction (see Sec. 4.2.2). Furthermore, data flows can be used to track the usages of one single data item over multiple hierarchies and states. On top of that, data flows are suitable to track the origin of all state parameters. Without visualizing data flows explicitly, the overview over a task can be lost quickly, especially if data is passed several hierarchy levels up and down.

This abstract data flow visualization can be used to highlight the benefits of HPFD-GL in terms of analyzability (regarding this also see Curtis et al. (1989)). In the following examples I use the O notation both to express the complexity of an operation for the computer (as commonly used to specify upper bounds for algorithms) and for the user. For the latter one I use O_U to express the amount of operations the user has to perform, i.e., the amount of lines the user has to analyze in order to find a variable or function occurrence in the code. For the latter one, I will assume supporting features as commonly known from IDEs.

I analyze two use cases of data modification. The first is that we suppose that one parameter of a state is wrong. To analyze the bug, the origin of the data flow has to be identified. Using a HPFD-GL the data flow of a state (e.g., state “a9”) can be

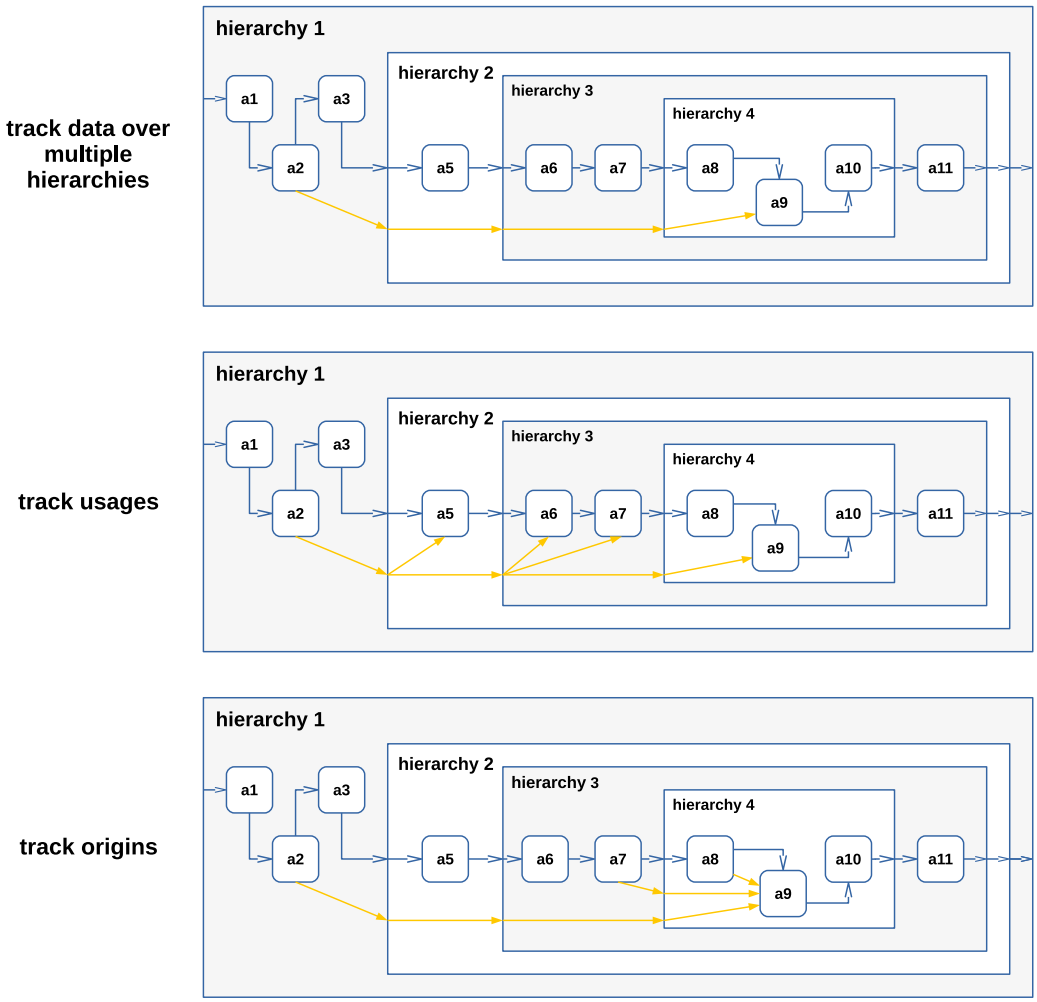


Figure 4.14: *Abstract HPFD-GL examples with control and data flow.*

followed until the data flow reaches the originating state (i.e., “a2”,). The user just has to follow one data flow via multiple data flow hierarchies h_i, which is supported by the Continuous Visual Abstraction, i.e., zoom (see Sec. 4.2.2). This means a complexity for the user of $O_U(1)$. The system’s complexity would be $O(n_h)$ (with n_h being the number of hierarchies of the current program), as the system has to travel up all hierarchies until it reaches the data source. In the case of a textual representation the user would have to trace back the call hierarchy starting from the hierarchy of the bug occurrence (i.e., “hierarchy4”). As already shown in Sec. 4.3.1, hierarchies in HPFD-GL map to functions in textual languages. Thus, the complexity for the user is $O_U(n_h)$. In our example the user has to trace back the control flow four times, i.e., starting at “a9”, going to “hierarchy3”, to “hierarchy2”, “hierarchy1” and finally to “a2”. This assumes the ability of the user’s text editor to search for variable (re-)assignments. The search support saves the user from having to analyze each line

of code between the function signature of the current hierarchy and the line in which the next function of the traceback is called (these lines of code are now called l_{h_i}). In case the search is not available the complexity for the user raises to $O_U(\sum_{i=0}^{n_h} l_{h_i})$. In any case, the system's complexity would be $O(\sum_{i=0}^{n_h} l_{h_i})$, as the system has to travel up all hierarchies and search for variable access in all lines l_{h_i} of each hierarchy h_i.

The second example is the refactoring of a variable (e.g., changing the data type or its content) and thus its usages, which are spread over various states (e.g., "a2" in the second image of Fig. 4.14). To find all usages of a data in the HPFD-GL case the user just has to follow the spreading of the data flow (four times in the example case). Thus, the user's complexity is $O_U(u)$, with u being the number of usages. The system's complexity would be $O(u \times n_h)$, as the system has to travel up all hierarchies h_i for each of the usages u. In the case of a textual representation the user has to trace all lines of code between the source and the sink for all usages. This leads to $O_U(u \times n_h)$ with search support and to $O_U(u \times \sum_{i=0}^{n_h} l_{h_i})$ without search support. In any case, the system's complexity would be $O(u \times \sum_{i=0}^{n_h} l_{h_i})$, as the system has to travel up all hierarchies h_i for each of the usages u and has to analyze all lines of code l_{h_i}.

Consequently, being able to model the data flow explicitly leads to much less user effort when data origins and usages have to be tracked.

4.3.4 Robotic Data Flow Examples

Next to the abstract examples of the previous section, I want to highlight real robotic use cases as well. The use case of Fig. 4.15 is related to the AIMM validation experiment (see Sec. 7.1.2). The figure shows parts of the plan to collect all objects of a certain type from a target location (i.e., scene) and places it on another object (e.g., another desk or the robot's transport area). The data distribution of the *object_id* data item to the "pick item" state and the *target_object* data item to the "place item" state is highlighted.

Fig. 4.16 shows the "pick item" action (of Fig. 4.15) in more detail. It shows the control flow of picking an item including various failure handling. After retrieving the possible grasps for an item a grasp is selected, which has a collision-free approach pose and for which valid inverse kinematics can be found. Subsequently, a collision-free trajectory is planned for the manipulator, which is executed afterwards. After querying various grasp parameters the grasp is executed and the world model is updated.

Next to the control flow, Fig. 4.16 also shows the data flow of its actions. This includes

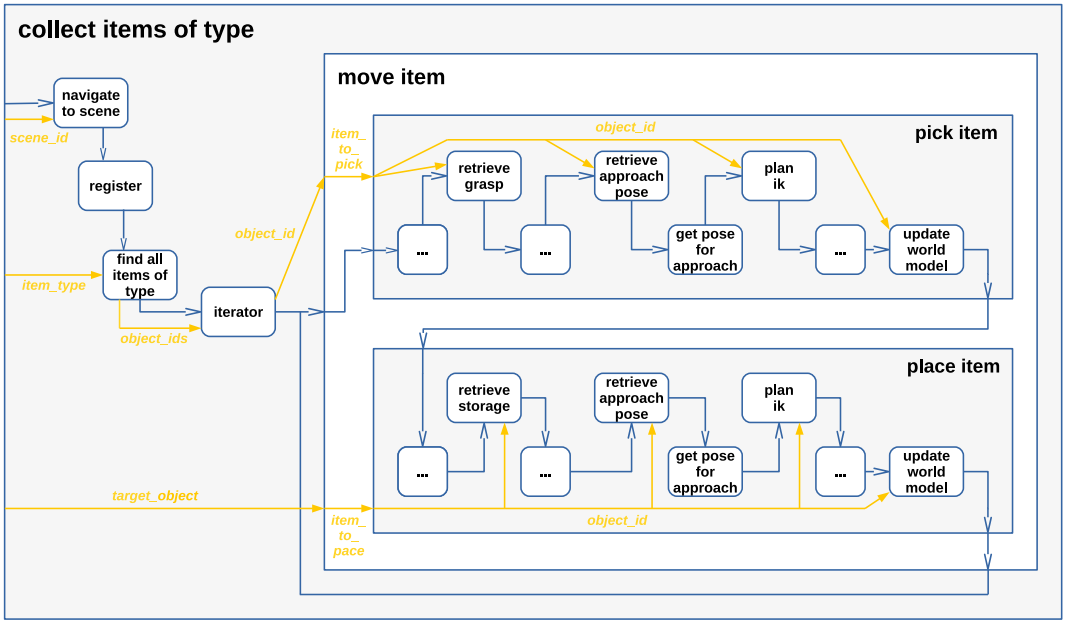


Figure 4.15: *HPFD-GL data distribution example in a robotic use case.*

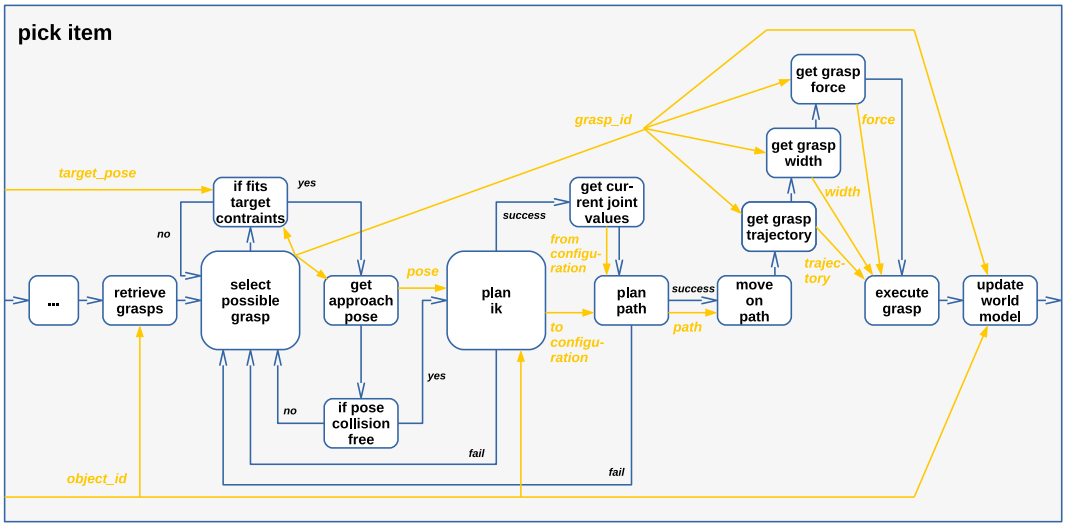


Figure 4.16: *HPFD-GL example for tracking data usages and origins in a robotic use case. The figure only shows a subset of all required data flows.*

the grasp descriptor “grasp_id”, various poses, the planned trajectory “path” and the grasp parameters. Although the control flow and the data flow are highlighted in different colors, the overall action behavior starts to become unclear as too much information is presented at once. Therefore, I propose three different views. Next to the combined control and data flow view of Fig. 4.16, a data-flow-only view helps to program and analyze the data flow (see Fig. 4.17) and a control-flow-only view helps to manage the logical part of the program (see Fig. 4.18).

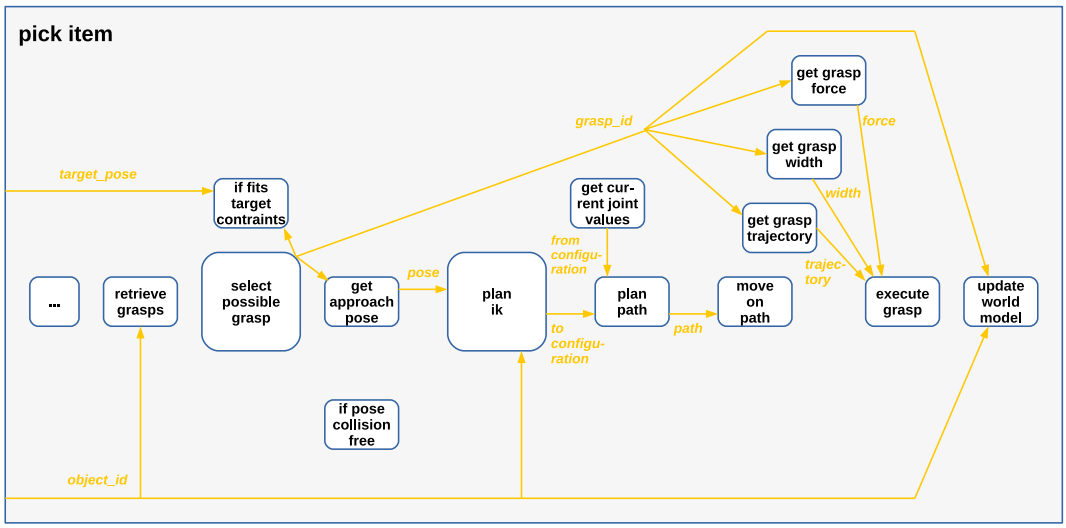


Figure 4.17: *HPFD-GL example for tracking data usages and origins in a robotic use case (data flow only).*

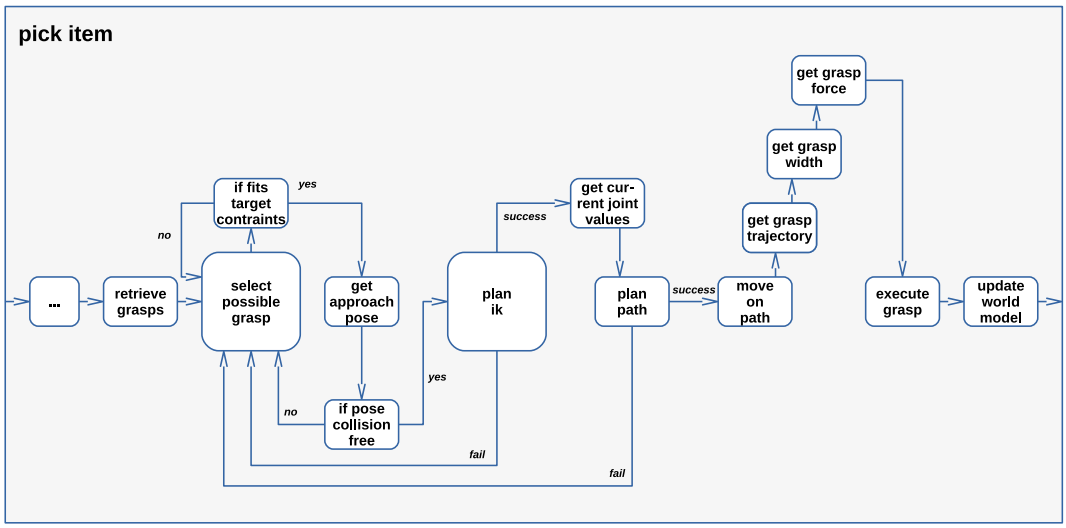


Figure 4.18: *HPFD-GL example for tracking data usages and origins in a robotic use case (control flow only).*

So far, I have primarily analyzed the two types of information flow (i.e., control and data flow) independently from each other. However, in real robotic use case scenarios they are tightly coupled. Although Figures 4.15 and 4.16 already show some kind of interaction, I will give another example. Fig. 4.19 shows the example decision tree of Sec. 4.3.1 in an even more realistic setup. The states “select RU”, “deploy location generator” and “storage selector” are shown in more detail including the data flow. The data they produce is forwarded through the “decider” HierarchyState into the “fetch RU from lander“, “deploy RU on ground” and “place RU on lander“ states.

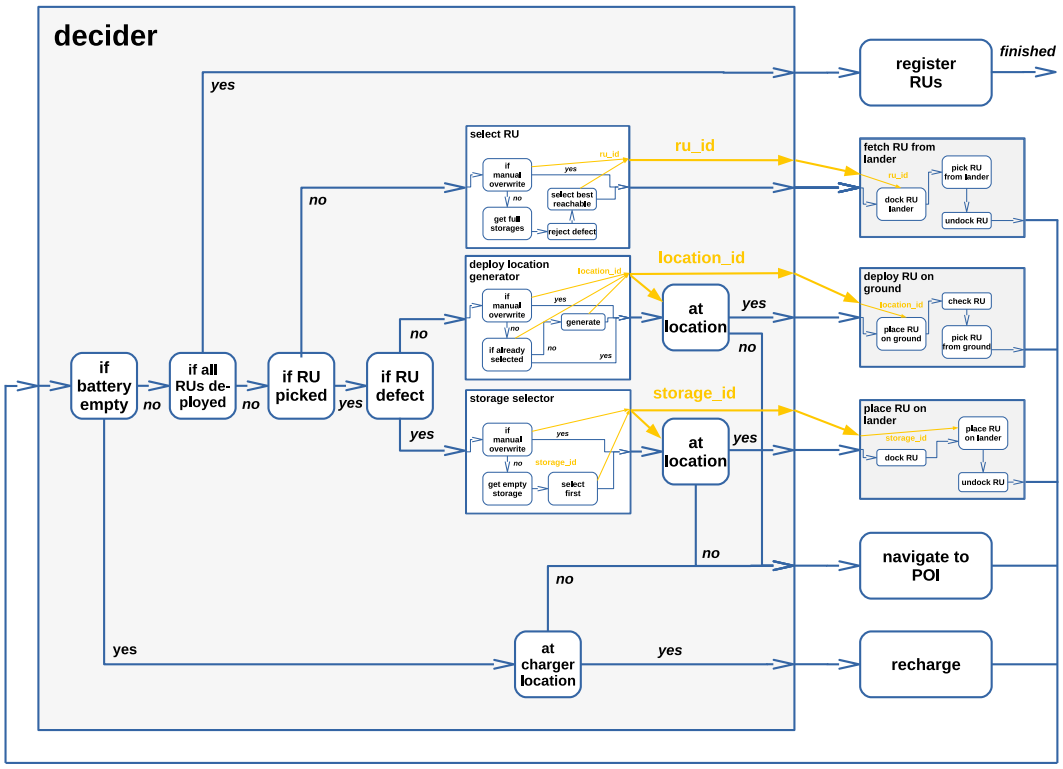


Figure 4.19: *HPFD-GL decision tree example with data flow. This figure extends Fig. 4.13 with data flow information.*

The three data-producing states are also shown in Fig. 4.20. The “select RU” state, cares about the selection of an appropriate RU to be grasped. At first, the system allows for manual intervention of the RU selection. This makes sense, if a monitoring crew already detected a defect for a RU, which was not yet observed by the robot autonomously. In this case a working RU is selected remotely. If no manual intervention is desired, and all full storages are retrieved, the ones already marked as defect (which, e.g., have already been returned by the robot, after unsuccessful deployment) are rejected. Finally, the RU is chosen that fits best into the workspace of the robot’s manipulator.
The “deploy location generator” state at first also checks for manual overwrite, then checks if a deploy location was already selected (by, e.g., a previous decider state execution, which commanded the rover to navigate to the desired RU) and finally generates a deploy location if both was not the case.
Finally, the “storage selector” state checks for manual overwrite as well, and returns with the manually selected storage id or with the id of the first empty storage.

These examples show that many decisions manifest in data (e.g., the “ru_id”, “location_id” and “storage_id”). This data is then forwarded via data flows to other

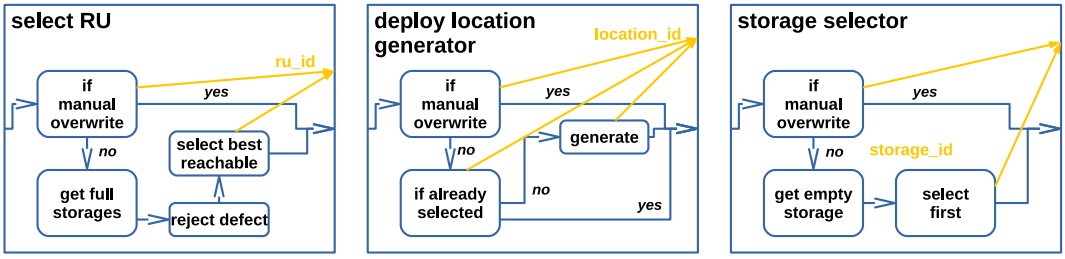


Figure 4.20: *Sub-states of the HPFD-GL decision tree example of Fig. 4.19.*

components of the system and is used to generate more decisions. This leads to one of the most important messages of my work, which will be further supported by the validation experiments: Control flow or data flow by themselves are not sufficient to model complex robotic behavior. The combination of both allows to present the most consistent picture of a complex, real world robotic behavior.

4.3.5 Abstraction Levels Coverable by HPFD-GL

The 2T* architecture introduced in Sec. 2.4 covers all software layers of a robotic system. I identified the "procedural task control" as one of the most central parts for robot control in the executive layer, which is covered by HPFD-GL. It triggers all modules of the lower level (i.e., the functional layer) and offers powerful, abstract robot skills to the higher level (i.e., the planning layer). This section is about clarifying the range of functionality the executive layer covers, i.e., what are the most low level functions HPFD must be able to trigger and what are the most abstract skills, which can be implemented with HPFDs and which are offered to the planning instance (i.e., the human or an automated task planner).

From a high-level perspective, HPFDs are able to represent arbitrary complex robotic plans. The high-level decision tree of Fig. 4.13 shows a HPFD able to solve the task of deploying a network of seismometers (see Fig. 4.12). More use cases, which include sample return tasks and fully automated workcell supply, can be found in the validation chapter (Chpt. 7). In all these use cases, HPFDs allow the robot to fulfill a task fully autonomously. Yet, they allow human supervision and intervention as well. For the industrial use case the human can decide which workcell to serve. For the ROBEX scenario, the lander selects and schedules the single missions and can define which RU to deploy. Reasonable abstraction functionalities are inherent to all major task control languages presented in Chpt. 3 and they are not the unique selling propositions of a task control language.

More interesting is how tight a task control language can interact with the underlying robotic system. How well can it cope with a huge number of functions and parameterization possibilities of full blown hardware abstraction layers. Many planning systems and task control languages focus on discretized, artificial environments such as the blocks world (see Gupta and Nau (1992)) or the Wumpus world (see Russell and Norvig (1995)). However, real robotics cannot be mapped to such simple, discretized worlds as the real world is continuous and objects move according to Lagrangian dynamics, taking masses, velocities, forces and gravity into account (see Craig (2009)). A task control language for real robotic systems needs to be able to leverage the information provided by Lagrangian dynamics. Neglecting such real world properties might include that the understanding of the robot about real world effects are lost. Some examples of these effects are:

- Interaction forces: If the robot presses against an object, forces act both upon the robot and upon the object. If the robot presses too strong, the object or the robot breaks.
- Friction: If friction is high, more force is needed by the robot to move (e.g., push) an object.
- Clamping: If two objects are not properly aligned or friction is too high, the object or the gripper might clamp.
- Overheating: If the robot carries a heavy object with a stretched arm for a long time the joints might overheat, causing the robot to drop an object.[2]

This list of physical interaction possibilities is long and does not have to be fully covered by a robot to act robustly in an unknown environment. However, the robot needs some understanding of its own domain, otherwise it cannot react to unforeseen events properly. For example, for autonomous navigation a robot needs to know that hitting obstacles implies high risks. Thus, most navigation approaches use some kind of obstacle map to avoid collision (e.g., see Schuster et al. (2017)). On top of that, sensing the real world always leads to uncertainties in the estimation of some object's or another agent's pose and other properties. Robots need a way to cope with such uncertainties. Thus, when interacting with objects the robot needs to anticipate contacts and has to use appropriate controllers which allow the robot to hit rigid objects without breaking its joints.

[2]For this reason we implemented a heat protector into the manipulator controller of LRU's Jaco arm. The heat protector observes the estimated temperature of a joint and commands the manipulator to stop and slowly remove the forces from all joints if a certain threshold has been exceeded.

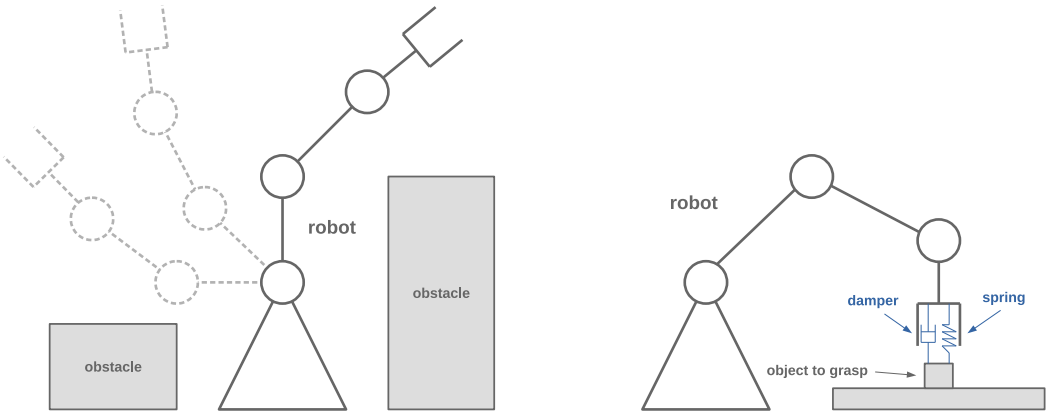


Figure 4.21: *Two basic control modes of robotic manipulators: position control (left) and impedance control (right)*

There are many types of controllers, among others position, velocity, admittance and impedance controllers (see Albu-Schäffer et al. (2007)). They can act both in the joint space of the robot or in the cartesian space. The two most important control modes are visualized in Fig. 4.21. In position control mode, the robot simply drives to the specified target configurations trying to reach certain velocity, acceleration or timing constraints. Upon encountering contacts, the robot will use its whole power to reach the specified configuration in time. The advantage of this control mode from an applications perspective is its exact path accuracy, which is imperative in narrow passage or if a target position has to be reached as exact as possible (e.g., for welding). The obvious downside is, that if the manipulator hits any non-modeled obstacles it will damage itself or the obstacle or both.

In the impedance control mode, target positions are specified with additional stiffness and damping parameters. This allows to setup up a virtual spring-damper system at each goal point, which "drags" the robot to a certain position. The stiffness parameters constrain the maximum force the robot applies in order to reach the target. Furthermore, the damping parameters constrain the velocity of how fast the robot will reach the target. The advantage of this approach is that the robot will only exert some of its power to reach its goal. If it hits an obstacle on its way, it will stop upon contact, leading to safe behavior of the robot in case of environment uncertainties. The downside is that the robot won't reach the target poses exactly as the modeling errors of the robot's joints and masses lead to wrong friction and force estimations and thus to inaccurate TCP positioning by the virtual spring-damper system.

The force, stiffness, damping and velocity parameters must not reside purely in the functional layer as the executive layer needs to reason about that. This is shown

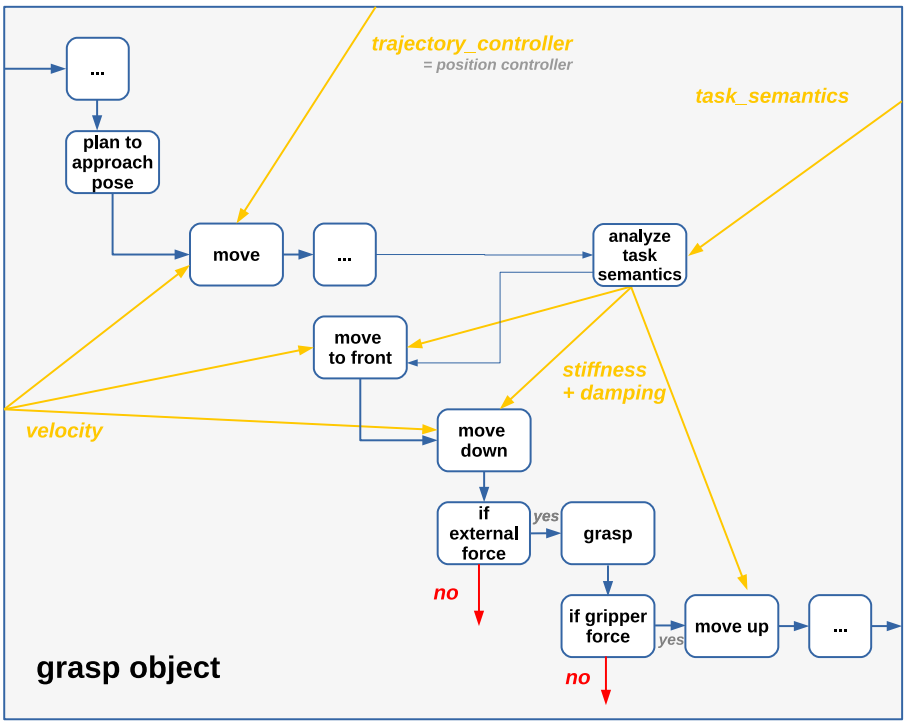


Figure 4.22: *HPFD-GL program to grasp objects taking force events and controller selection into account*

by two examples. The first shows an abstract HPFD plan to grasp an object (see Fig. 4.22). At first the manipulator is moved to the approach pose of the object, then the task semantics are queried to estimate the grasping parameters. Subsequently, the grasp trajectory is executed ("move to front" and "move down") and the object is grasped. At the end of the procedure, the object is lifted ("move up"). Two conditions check the state of success of the grasping procedure. The first checks, if the manipulator is in contact with the object before grasping. The second cares about the gripper measuring some force between its fingers. If one of the checks fail, an error handling procedure (which is not included here for brevity) is triggered. This example shows that reacting to forces is a sound solution in order to check the progress of a behavior. Moreover, many parameters are not movement specific but passed from outside as parameters such as the velocity, the trajectory controller and the task semantics. The velocity is passed as only the higher level actions are aware of velocity constraints, e.g., if a higher level observer detects the presence of another agent. The same holds for the trajectory controller. If no other agent is near the movements might be executed faster, using the position controller for improved path accuracy. However, if another agent is close the controller has to be (next to reducing the velocity) switched to impedance control in order to anticipate and handle unforeseen collisions appropriately (only possible if the task constraints allow

for it). Finally, the task semantics are necessary in order to parameterize the grasp trajectory. If the goal object or the supporting plane is delicate or high uncertainty in their position is expected, the forces and velocities have to be reduced. If they are rigid metal parts and no uncertainty present, then the velocities and the grasp forces can be increased.

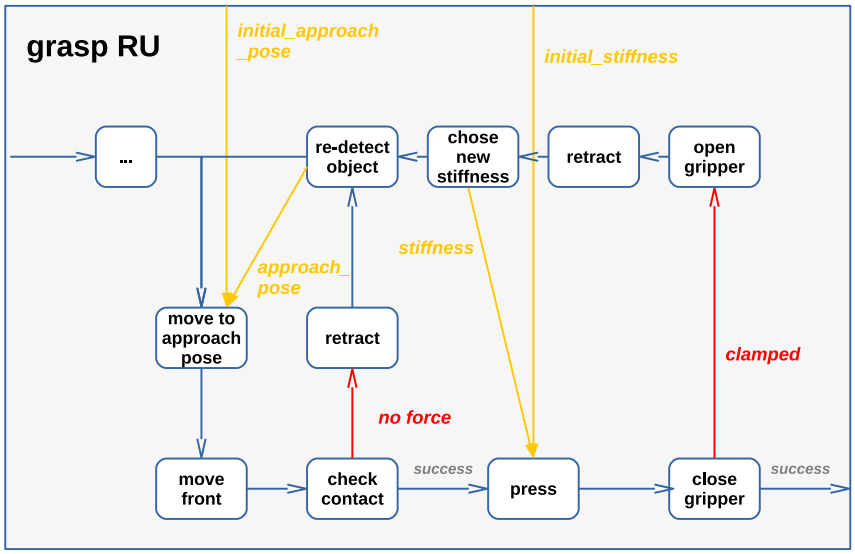


Figure 4.23: *HPFD-GL program for RU grasping taking force events for error handling into account.*

The second example (see Fig. 4.23) is based on the real use case of the ROBEX validation experiment of Chpt. 7. It is related to the abstract grasp procedure above but this time is tailored to grasp a RU and focuses more on error handling and decision propagation via data flows. In the case that the check for contact after moving towards the target object fails then the manipulator retracts, the object is re-detected and the procedure starts from the beginning. Accordingly, if, after closing the gripper, it reports clamping, the gripper is opened again. Subsequently, the manipulator retracts, another stiffness factor is calculated for the next grasping command, the object is re-detected and the procedure starts again. The error handling procedures decide to recalculate two important parameters repeatedly, which are the approach pose on the one hand, and the stiffness on the other hand. Next to being related to the physics of the underlying system, the error handling highlights the importance of the data flows. The difficulty of analyzing erroneous behavior related to the approach pose or the stiffness factors is greatly reduced as the control and data flow is recorded for each iteration of the procedure using the execution logging mechanism, which will be presented in Chpt. 5.

These examples show that in order to deal with real world use cases a task control

language has to be able to reason about the required domain properties (e.g., the Lagrangian dynamics) and that by taking the task semantics of higher level action into account proper action parameters can be inferred.

4.4 The HPFD World Model

The general necessity of keeping track of all external states the robot can observe in its environment for task control purposes was already motivated while presenting the 2T* architecture in Sec. 2.4. Next to this general motivation, I will provide deeper conceptual insights of our world model solution, which we already published in a previous publication (see Lehner et al. (2018)). As mentioned above, the robot's world model is also called its *belief state*.

4.4.1 The Benefits of a Dedicated World Model

Intuitively, one might come up with the idea to hold all the data the robot knows about its environment inside the procedural task control module, in the case of HPFDs inside the private, scoped and global data stores. For several reasons, this is not a wise choice.

On the one hand, the data of the robot's belief state is quite large. In more complex scenarios the robot has to interact with several complex objects, which have a 3D shape, normally presented in CAD models, and many properties, such as color, size, weight, center of gravity and inertia parameters etc. Furthermore, the robotic system records its surroundings and creates 2D obstacle grids, 3D voxel maps or point clouds and semantic maps for object classifications, path planning and ground classification (i.e., for navigation). Camera recordings of important points of interest and close-up images of critical objects are more examples of essential information the robot has to be aware of. Storing all this information inside the procedural task control would bloat it up enormously and would obstruct the real use of procedural task control, i.e., the specification of the robot's behavior.

On the other hand, the information of the robot's belief state is very heterogeneous and deeply interconnected. Various objects in the world are associated to each other by spatial and temporal relations. For example, an agent carrying an object will not only change its own position while driving but also the position of the object.

Furthermore, the timestamps of actions are crucial information for further reasoning on the objects position over time. Next, several objects are instances of the same base-classes and thus share the same type. If now more information of the base-class is available it must be propagated to all object instances. E.g., if a robot detects 10 boxes in a shelf, in which only resides boxes of the same type and it estimates the inertia parameters of one of them, then it knows the inertia parameters of all of them. Rather than copying this information to the object instances, the information can just be changed for the base class and the instances retrieve this information by their type-relation to their base class. As the data concepts of HPFD do not allow to store such interconnected data efficiently, the belief state of a robot must be a separate module.

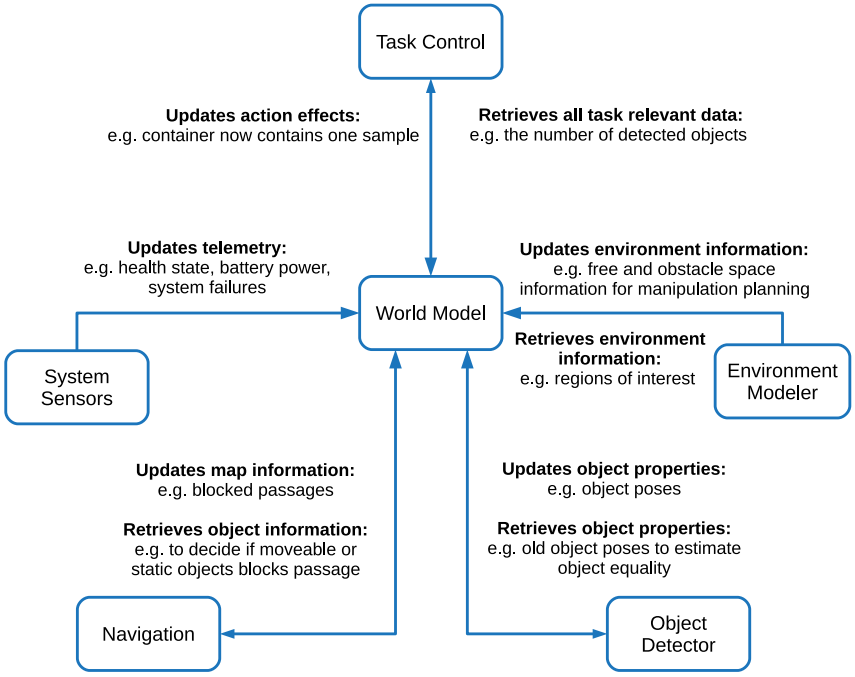


Figure 4.24: *Overview of common system modules, which access the world model for different purposes.*

Finally, though the information needed for procedural task control overlaps the knowledge stored in a world model, it is only a fraction of the whole belief state information. The world model stores information for many system modules, which can read and write the data independently of the task control layer. Fig. 4.24 shows some typical nodes that can access the world model.

Thus, the world model and the task control unit are different modules serving fundamentally different use cases, which is the reason why they have to be kept separated. Separation of Concerns (see Kulkarni and Reddy (2003)) is an old and

widely accepted pattern for complex systems and aims at dividing the systems in to several modules which can be updated and replaced independently to each other. All these reasons motivate for a world model independent to the task control module.

Another approach to tackle the world model challenge would be that each component maintains its own world model and the task control unit accesses all of them as needed. However, there is one major draw back to this approach: *synchronization*. Although it is sensible, that not every component shares all its internal states with every other component (e.g., a fine grained 2D obstacle map of the environment is only needed in the navigation module and is thus not shared) some information has to be shared and synchronized to many modules. For example, if a robot grasps an object, then not only the load data for the arm changes, but also the field of view will be obstructed and the battery power will decrease faster if it is a heavy object. Therefore, it is a wise choice to synchronize such information in a central world model component.

4.4.2 World Model Concept

As stated above, the world model data is very heterogeneous and highly interconnected. This motivates a graph based solution for storing the robot's belief state.

Nodes Nodes represent entities of the robot and it's environment. Those entities can be physical objects such as the robotic arm, one of the robotic wheels or a rock. Also, other agents are physical objects such as a lander or another robot in a planetary exploration scenario. Non-physical information can be modeled as nodes as well, such as object positions, where the robot can grasp the object (i.e., the approach pose) or the health state of the rover.

Properties Each node can have several properties, which add additional information to the physical or meta object. For example, a rock has a size and consists of several materials. For complex information, it is always possible to create a new node and connect it to other objects in order to describe them further.

Edges Edges represent relations between objects. As stated above, there are many types of relations. Possible examples are spatial, temporal and type-based connections. Edges always connect only two objects. The edges can be one-directional, bi-directional or undirected.

World Model Graph The world model graph consists of nodes and edges. Graphs can be directed, undirected, cyclic or acyclic. Having the world model in the form of a graph, allows us to use all the mature graph algorithms of information science. Those include algorithms to find shortest paths (Dijkstra), to identify circles (Depth First Traversal) or to find suitable orders for graph traversal (algorithms solving or estimating good orders for the Traveling Salesman problem). Another, much more relevant advantage is the fact that such graphs can easily be translated into triple stores (see Margitus et al. (2015)). This allows for using logical programming languages such as Prolog or query languages for graphs such as CYPHER (see Francis et al. (2018)). Finally, it enables to run queries on the data to infer new knowledge that is not explicitly encoded into the world model.

Labels Each node can have an arbitrary amount of labels, which represent categories a node belongs to. In opposite to node properties, those categories allow to classify nodes into different groups, which helps to maintain a node hierarchy and improves node retrieval when performing queries on the graph. Examples of such labels in the robotics domain could be *Physical Object*, *Robotic Part* or *Region of Interest*.

4.4.3 Application Example

Fig. 4.25 shows a part of an exemplary world model of a robot in a space scene similar to the ROBEX scenario. A rover is located next to a lander in a map. The lander, whose pose can be recognized by its markers has mounted a remote unit, which carries scientific instruments to be deployed to a remote site. There are several labels classifying the nodes into different categories. The *Robot* label classifies robotic parts, the *Physical Object* label represents arbitrary physical objects, the *Marker* class describes all kinds of markers, which can be used to localize technical equipment, and the *Grasp* label is used for meta objects, which consist of a 3D pose describing a possible grasp point for a robot in order to lift an object. There are three types of relations in the graph, i.e., *spatial*, *type* and *can-manipulate*. Spatial relations just describe relative positions between objects, type connections specify the type of a node, and can-manipulate relations are used to specify that the source of the relation can be used to manipulate the target of the relation. E.g., in the figure the defined force-controlled manipulator (FC-Manipulator) can be used to manipulate a shovel, which in turn can manipulate soil samples.

The graph structure of the world model allows us to ask various questions. For example: *Can the LRU manipulate the object called "remote-unit-1"?* Although, no

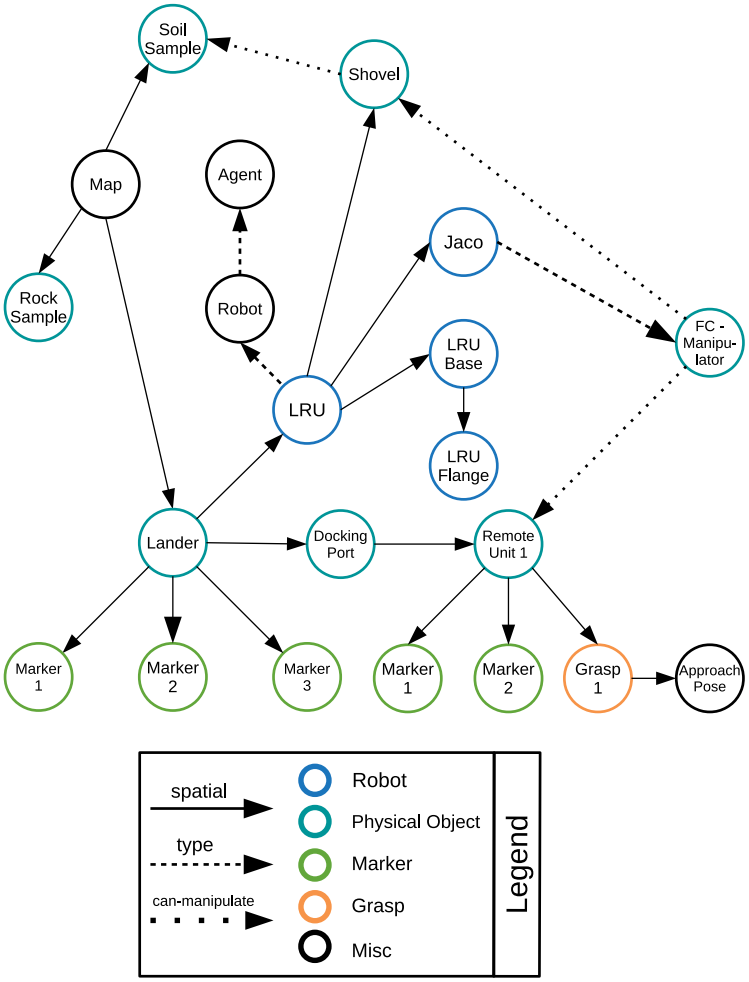


Figure 4.25: *Exemplary world model in which a robot is modeled together with a lander in a map reference frame.*

component ever inserted this information explicitly into the world model, we can retrieve the answer by using a suitable query language. Assuming that the world model information was inserted into a Neo4j database (as described in the implementation Sec. 4.6.3) we can formulate such a query in the CYPHER language.

Using the concrete query as given in Fig. 4.26 would yield the manipulator able to manipulate the remote unit object. A non-empty query result automatically means that the query successfully found a valid node, thus the answer to the question above is *yes*.

Another example question would be: *Is there an idle agent with a manipulator, able to pick up a remote unit from the lander?*

Fig. 4.27 shows the appropriate example query for the world model of Fig. 4.25. The CYPHER query would return the LRU node, which means that there is indeed an idle

```
MATCH (robot) - [:spatial*] -> (manipulator)
MATCH (manipulator) - [:can_manipulate*] -> (remote_unit)
    WHERE remote_unit.name = 'remote-unit-1'
RETURN manipulator
```

Figure 4.26: *A simple example CYPHER query to retrieve information about the rover, based on the world model information of Fig. 4.25.*

```
MATCH (agent) - [:type*] -> (robot:robot)
    WHERE agent.name = 'agent' and robot.state = 'idle'
MATCH (robot) - [:spatial*] -> (manipulator)
MATCH (manipulator) - [:can_manipulate*] -> (remote_unit)
    WHERE remote_unit.name CONTAINS 'remote-unit'
MATCH (lander) - [:spatial*] -> (remote_unit)
    WHERE lander.name = 'Lander'
RETURN robot
```

Figure 4.27: *An example CYPHER query to retrieve information regarding an idle agent, based on the world model information of Fig. 4.25.*

robot able to fetch a remote unit from the lander. The single *match*-statements can be explained as follows

- All nodes of type rovers are filtered, which are an instance of *agent*
- All idle robots are filtered from the set of rovers
- All robots are filtered, which have a manipulator able to manipulate remote units
- A lander is searched for that has mounted a remote unit, which can be manipulated by the manipulator of the robot

One aspect of this query is that to query remote unit objects, the node's name is checked for the substring *remote-unit*. This is valid if the assumption holds that the name of all remote units is created using the pattern *remote-unit-x*, whereat *x* is the number of the remote unit. Another way to model this would be to create an object-type node *remote-unit* and create a type-relationship between each remote unit node and the remote-unit object-type node (such as the type relation between the *Robot* node and the *LRU* node).

4.5 Adding Semantic Knowledge to HPFDs

By using HPFDs, complex reactive, concurrent and goal-driven robotic behavior can be programmed. Different states call different *system functions* to control the system's actuators and sensors. Based on different calculations and conditions the decision upon the robot's course of action is performed.

However, the behavior itself cannot reason about its own goals and actions, as these *system functions* are not semantically described yet. This leads us to the concepts of operational and descriptive action models (see Nau et al. (2015)). The HPFD states do perform various tasks in their own task control language (operational action model). However, the semantic description of the task (descriptive action model) consisting of the effects of the task, the preconditions that must be fulfilled in order to be able to execute the task, and other task parameters such as classification information is not yet provided.

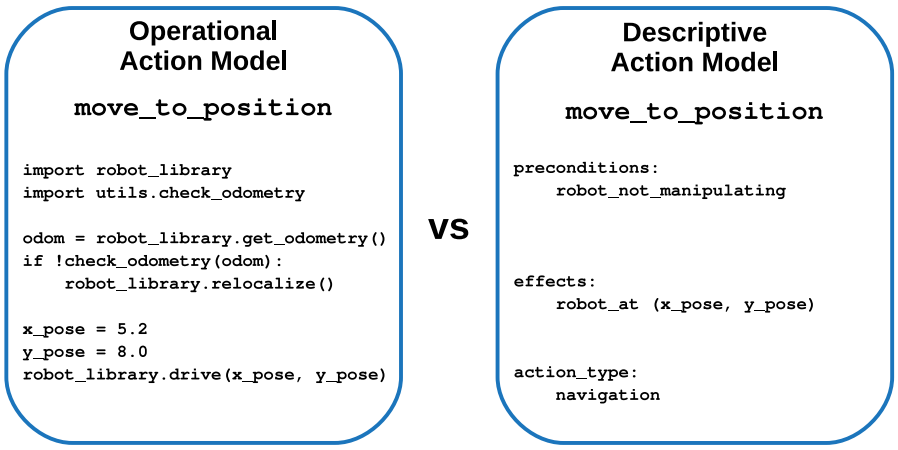


Figure 4.28: *Example for differentiating between the operational and the descriptive action model. Representative languages for the operational action model are HPFDs or PRS (see Ingrand et al. (1996)). Defining descriptive action models is possible with PDDL (see Mcdermott et al. (1998)) or STRIPS (see Fikes and Nilsson (1971)).*

4.5.1 Operational versus Descriptive Action Models

Fig. 4.28 gives an example of the difference between the action models. The left side shows the operational action model of a *move_to_position* action. It consists of task control operations which command the robot using a set of library functions. At first, the robot's odometry is checked for anomalies and if they occur the robot triggers

the re-localization procedure (as offered by an *Adaptive Monte Carlo Localization (AMCL)* navigation solution, see, e.g., Zhang et al. (2013)). Afterwards, the robot is commanded to drive to a certain position. The right side of the figure shows the descriptive action model. It consists of action preconditions, effects and classification information. One example precondition of the action is that the rover must currently not manipulate any objects before moving to another position. One effect of the action is that the robot is at the target position. Finally, it is defined that the action belongs to the class of navigation actions.

The descriptive action model serves as a semantic abstraction layer for the operational model. Abstraction always includes that some action specific information is lost, in this case the checking of the odometry and the call to relocate. At the one hand, abstraction is a necessary tool to handle complexity, as otherwise the amount of information on higher abstraction layers would explode. On the other hand, high abstraction inevitably leads to the problem that many task errors cannot be represented in a certain abstraction level (see Bohren and Cousins (2010)). For example, if friction is not modeled in a descriptive action model domain, and friction is the reason why a certain object slips again and again from the surface of another object, then a robot will never be able to solve this error with the means of its descriptive action model.

In fact, it is a difficult challenge to choose the correct abstraction level. It may vary a lot based on the available computing power of the cognitive agent, based on the agent's domain and based on the agent's goals. Different actions are needed in order to serve different jobs in different scenarios.

It is essential that the operational and descriptive action model are closely coupled. Implementation-wise they can even be maintained in the same framework (see Sec. 4.6.4). Otherwise, the danger is high that they drift apart, such that the operational and descriptive model do not represent the same action anymore.

These facts served as motivation to define RATNs, which combine operational and descriptive action models in one single model and can be designed in the same software framework (i.e., RAFCON, see Sec. 4.6.4).

4.5.2 Resource Aware Tasks Nodes (RATNs)

Resource Aware Tasks Nodes (or simply *task nodes*) add a semantic layer around HPFD states (see Fig. 4.29). They either consist of a single HPFD state, a whole HPFD state machine, or a set of other task nodes. The latter one are called child task nodes. This

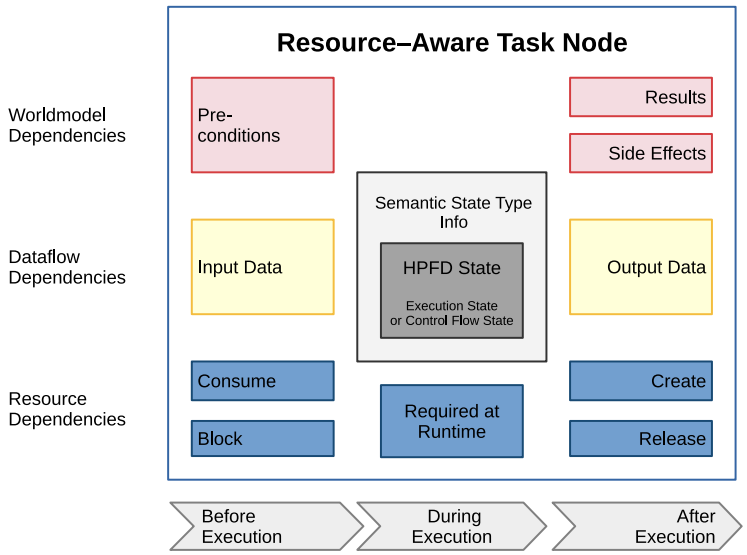


Figure 4.29: *Overview of the components of RATNs.*

encapsulation is accomplished by using the HPFD hierarchy concept.

RATNs possess *semantic state type info* consisting of an *action type* and a *control flow type*. The *action type* can be used to classify the action represented by the task node. The classification enhances the information about an action and can be used to structure and group the available set of actions a robot offers. Action types can map to complex taxonomic information and can thus be modeled using ontologies (see Chpt. 5). Exemplary action types are *navigation*, *manipulation*, *world-model* or *motion planning*. The *control flow type* adds control flow related semantic information to the already existing HPFD state types (i.e., *Hierarchy*, *BarrierConcurrency*, *PreemptiveConcurrency*, *Execution* or *Library*). The control flow type is not relevant until Chpt. 6 and, thus, will be explained there in detail.

Finally, task nodes consist of three types of dependencies: *world model*, *resource* and *data* dependencies. Their concepts are explained in following.

4.5.3 Dependency Concept

As Fig. 4.29 shows, RATNs can have *Dataflow*, *Worldmodel* and *Resource* dependencies.

World model dependencies refer to world state predicates, as commonly known from PDDL (see Mcdermott et al. (1998)). I use world model as synonym for PDDL's *world state*. Both terms describe the robot's representation of the world. When I talk about the environment state, then I refer to the robot's real external state that

the robot could have sensed correctly or not. World model dependencies consist of preconditions and effects. The effects split up in results and side-effects, whereas the former refers to the goals of the action and the latter to the side-effects which are inevitable connected to performing the action, but are not the reason why the action was executed.

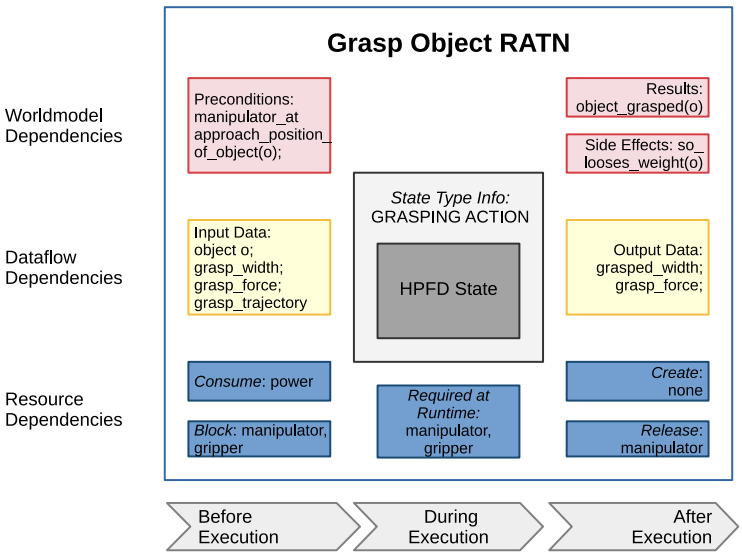


Figure 4.30: *Resource-Aware Task Node example.*

Fig. 4.30 shows common preconditions and effects of a *Grasp Object* task node. The precondition of the action would be that the manipulator has to be at the approach position of the object in order to grasp it. As a result, the object would be grasped. One of the side effects would be, that the *supporting object so* (i.e., the object on which the target object is located) looses weight. In order to get a consistent model for several application scenarios and robotic systems, consistent predicates have to be used. Otherwise, if a programmer chooses different predicates for each different robot, the HPFD actions become robot-dependent. This does not only bloat up the task control code and increases redundancy, but also renders comparison of robot performance for similar tasks very difficult. For modeling these kind of dependencies, a key-value based data store is used, which can hold all preconditions, effects and type information. Unary, binary and n-ary predicates can be stored. For more complex (e.g., hierarchical) information each entry in the data store can also hold structured data by using nested dictionaries.

Dataflow dependencies are the second type of task node dependencies, which are commonly neglected by task control languages (e.g., see Beetz et al. (2010)). They represent data prerequisites which must be fulfilled before an action can be executed. For example, the target object must be identified before the robot can pick it up. Also,

the grasp force for grasping an object has to be calculated before performing the grasp. Many factors may influence this calculation, based on the objects material, its content and the speed the robot is going to move the object from one place to another. Next to requiring certain input data for its execution, an action also returns certain output data of a specific type, which may be required by another action in turn. For modeling dataflow dependencies, HPFD data ports are used combined with a flag that the port values must not be covered by default values but by a dataflow.

Resource dependencies define all resource constraints for a task node. Next to specifying which resources are blocked or consumed by executing an action, resource effects, i.e., the release or the creation of new resources are considered. As a matter of fact, there are five classes of resources: *task, process, module, system* and *world* resources. Task resources specify task execution related resources such as: *two out of five* boxes have already been delivered. Process resources relate to running processes of the system, e.g., in order that a collision free path can be calculated a *motion planning process* has to run. Module resources are internal states of modules, on which other modules rely upon, e.g., the navigation module hast to report that the robot is *localized* in order to execute navigation commands. System resources are critical system states other task nodes rely upon such as the *battery power* or the amount of *available storage*. Finally, world model resources are a special kind of resource and represent dedicated world models. Several world models are needed in special cases such as parallel planning or simulation of future world states in order to check for possible errors or in order to pre-calculate intermediate task results. Resources are managed in a dedicated resource manager, which is described in Sec. 6.2.1. As it is the case with world model dependencies, resource dependencies can be stored using predicates grounded in a resource ontology. More information about ontologies is provided in Sec. 5.2.

The formalization of the dependency concept is given in Chpt. 6 as this is the chapter relying on RATNs in a formal way.

4.5.4 Use Cases of RATNs

In the course of this work I will use RATNs for several use cases to show their applicability and power.

Logging and Analysis: RATNs can be used effectively for generating information-rich task analysis results. This topic is covered in Sec. 5.3.

Model Checking: In critical missions some system states must never be reached. One approach to ensure that is model checking, which can be used to unburden the programmer to check for these critical states manually or with the means of testing, and proof the non-existence of specific system configurations automatically. Thereby the safety level of the robot can be increased. The topic of checking semantic properties of HPFDs is covered in Sec. 5.3.1.

Concurrent Dataflow Task Networks: RATNs are the core action representation for CDTN, which are covered in Sec. 6.1.

Planning: In Appendix D, I will give insights in how robotic plans based on RATNs can automatically be generated using a task planner.

Documentation and Debugging: Finally, RATNs help to model task-related knowledge of the programmer explicitly, which is often hidden in purely behavior-based task programming languages. For example, during the initialization phase of a robotic system different processes are booted up in a specific order but the programmer nowhere tells why this specific order is needed. A motion planner for example needs a geometric world model provider, which passes the current 3D environment state to it. The naive approach would simply be to start the world model provider first and then the motion planner. Implicitly, this order will work and the programmer (programmer A) has done his job. However, if another programmer (programmer B) updates the behavior, decouples the module boot up and restructures it, the behavior might be erroneous, as the boot order is mixed up. By using resource dependencies programmer A can clearly state that the motion planning process requires a 3D world model provider. By this, programmer B knows the reasoning behind the boot-up sequence and can perform the refactoring correctly.

By using the different kinds of dependencies properly, RATNs can clearly pay out not only for documentation purposes but also for debugging. The dependencies that are required by a task node can easily be compared with the dependencies that can be fulfilled by the current system state. The resulting difference can be a key for successful error diagnose.

4.6 Implementation: The RAFCON Framework

The RAFCON framework implements the majority of the concepts described in this chapter. It consists of two parts: the RAFCON core (for the non-graphical concepts of HPFDs and RATNs) and the RAFCON GUI (focusing on HPFD-GL). We developed the framework at the German Aerospace Center and made it open source in 2018.[3] It can be found on *https://github.com/DLR-RM/RAFCON*. The world model is not part of the RAFCON framework and was implemented in a dedicated module.

4.6.1 The Core

RAFCON is programmed in Python and currently supports the Python versions 2.7 and 3.4 - 3.7. A huge advantage of Python is its platform independence. Until now, we tested RAFCON on various Linux distributions including Ubuntu[4], SUSE Linux Enterprise[5] and OpenSuse Leap[6]. Each ExecutionState runs a Python code snippet of arbitrary complexity able to import arbitrary modules. The entry point function is the *execute*() method and has a state reference, the state's input data, the output data and a reference to the global variable manager as its parameter. Each middleware (currently tested with ROS, DDS and Links and Nodes, see Sec. 2.5) to be interfaced with RAFCON has to provide an appropriate Python interface.

We wrote both the underlying state machine representing HPFDs and the HPFD interpreter called the *RAFCON execution engine* from scratch. The interpreter supports different execution modes such as *Start, Stop, Pause* and *Resume*. During execution an execution history is created, which keeps track of all executed states including their context data, i.e., timing information, input and output data and the final outcome. The execution history is a central aspect of task analysis and profiling, and will be presented in more detail in Sec. 5.5.2.

State machine execution can not only be launched at the start state of the root state but also at a predefined state. Also, executing a state machine until a specified state

[3] 'We' refers to several of my colleagues at the RMC and me. The list of contributors sorted by their contributed number of lines of code is: Sebastian Brunner, Franz Steinmetz, Rico Belder, Matthias Büttner, Lukas Becker, Mahmoud Akl, Benno Voggenreiter, Sebastian Riedel, Michael Vilzmann, Annika Wollschläger, Christoph Sürig and Nick Walter.

[4] `https://ubuntu.com/`

[5] `https://www.suse.com/products/desktop/`

[6] `https://www.opensuse.org/`

(similar to breakpoints) is possible. Furthermore, the interpreter supports stepping through a state machine similar to common IDEs:

- Step into: executes the next state in the next deeper hierarchy level or simply the next state if execution has already reached the deepest hierarchy level
- Step over: execute the next state on the same hierarchy level. If the next state is a container state all child states are executed as well.
- Step out: the execution executes all states until it reaches the next upper hierarchy level
- Step backward: executes a state in backward manner. For doing so, backward execution does use an obligatory *backward_execute*() function, which can be defined for each ExecutionState (next to the mandatory *execute*() function). Backward execution restores all context data of the current hierarchy that was valid before the state was executed in forward mode. The necessary information is retrieved from the execution history. This is a novel feature for behavior execution and is especially useful when debugging a system with a manipulator in intricate poses. As retracting the manipulator in confined spaces may not be possible easily (and often only in combination with a path planner), the possibility to backward execute relative end-effector movements can be very valuable and avoids time consuming system state reproduction procedures.

In order to persistently store RAFCON state machines the JSON format is used, which has the useful property that it is human readable and easily versionable, e.g., via git[7]. In principle, this textual representation of RAFCON state machines defines a DSL that could also be written manually. However, without tool support this is rather tedious. Thus, in order to be able to create state machines quickly and intuitively the RAFCON GUI has to be used.

RAFCON supports collaborative state machine creation. Libraries can be created by various developers or developer teams. These libraries can then be linked in order to build more complex systems. By using a core configuration file, the user can specify the behavior of RAFCON and can configure several file paths where RAFCON can find other state machines in order to dynamically link to them using LibraryStates. The ExecutionStates' Python script is loaded and executed dynamically when the ExecutionState is started. Also, the whole state machine itself is loaded dynamically state by state, which allows to alter the state machine also during execution. Finally, RAFCON is equipped with a powerful logging library, which supports different logging

[7] `https://git-scm.com/`

levels and allows for writing log statements either to the terminal, to log files, or both.

For RAFCON, we deliberatively decided against an event-driven design. With increasing complexity, event-driven systems are error-prone (see Parent (2007)), resulting from complex topics that need to be tackled, such as event caching and prioritization, event expiration and parallel event handling (see Lamport (1978)). Therefore, RAFCON is inspired more by flowcharts (see Simons (2000)), letting states decide about the next transition to follow. On top of that, observer structures based on PreemptiveConcurrencyStates can be used to react to external events efficiently[8].

RAFCON enforces clear modularization of all state machine components. In opposite to statecharts (see Harel (1987)), transitions are not allowed to cross hierarchy boundaries. This maintains the ability to reuse arbitrary state machine hierarchies (as LibraryStates) in order to compose more complex behavior.

4.6.2 The GUI

One of the main design goals of the RAFCON GUI was to enable the developer to focus onto the creation of robotic behavior in an intuitive, flexible and efficient manner. Reducing the time for reading, understanding and programming the behavior is a major aspect we had in mind during implementation. We put high effort in adding utility functions in order to reduce the amount of boiler plate code the developer has to write (as common in other textual languages such as in SMACH, see Bohren and Cousins (2010)). By implementing HPFD-GL, RAFCON offers a concise and clear way for specifying robot behavior, without sacrificing the abilities a general purpose programming provides as it enables full access to the underlying Python language.

The graphical user interface (GUI) is a central part of RAFCON. A GUI featuring a graphical visualizer and editor is an imperative feature of modern programming tools for robotic behavior creation. However, this is not an easy feature to achieve, as bigger state machines with deep hierarchies tend to become very complex. Many visual programming tools do not support hierarchies, such as NXT-G by Kelly (2010), or show elements on different hierarchy levels in the same size, such as SMACH by Bohren and Cousins (2010). Concept-wise, we tried to make visual programming of HPFDs as efficient and intuitive as possible and thus cooperated with a professional

[8] More information about event handling can be found at: `https://rafcon.readthedocs.io/en/latest/faq.html#why-is-rafcon-not-event-based`

interface design company[9], which developed the design and layout. Implementation-wise, there are still several features missing to reach that goal. Thus, I list the most important open features in the future work section 8.2.

GUI Structure

For implementing the RAFCON GUI the GIMP-Toolkit GTK+ is used. It is an open source GUI-Toolkit for writing graphical user interface for the X Window System[10]. The GUI is completely separated from the core. By using a MVC (Model View Controller) pattern changes made to the core are directly notified to the GUI for visualization.

Several design principles concerning the alignment and arrangement of GUI components are responsible for a well structured design. We extensively used the four gestalt principles *similarity, proximity, closure* and *continuity* to improve the GUI's structure and the speed of information retrieval (see Büttner (2015)).

A high emphasis is put on modularity and flexibility. The user is able to customize the appearance of RAFCON using a yaml-configuration script. Properties such as color schemes, shortcuts, logging and notification options can be configured. Various external editors for modifying the Python source code can be configured to enable developers to code in their favorite environment.

The GUI consists of five areas. A top bar, a left bar, a right bar, a bottom bar and the central area, called the *Graphical State Machine Editor*. All side bars are foldable, detachable and resizable. The left and the right bar each contain a notebook in which the tabs can be rearranged and relocated.

Fig. 4.31 shows the single areas of the GUI, including the single widgets. The state machine editor is the central element of the GUI and is further described in the next section. Label (1) of Fig. 4.31 refers to it. The left side bar consists of several widgets with information relating to a whole state machine or the whole RAFCON GUI.

(2) *Library manager*: holds a list of all mounted library paths including all available RAFCON LibraryStates. The states can be simply drag-and-dropped into the currently selected state machine.

(3) *State machine tree*: shows all states the currently selected state machine contains

[9] Interaktionswerk, https://interaktionswerk.de/

[10] `https://www.x.org/wiki/`

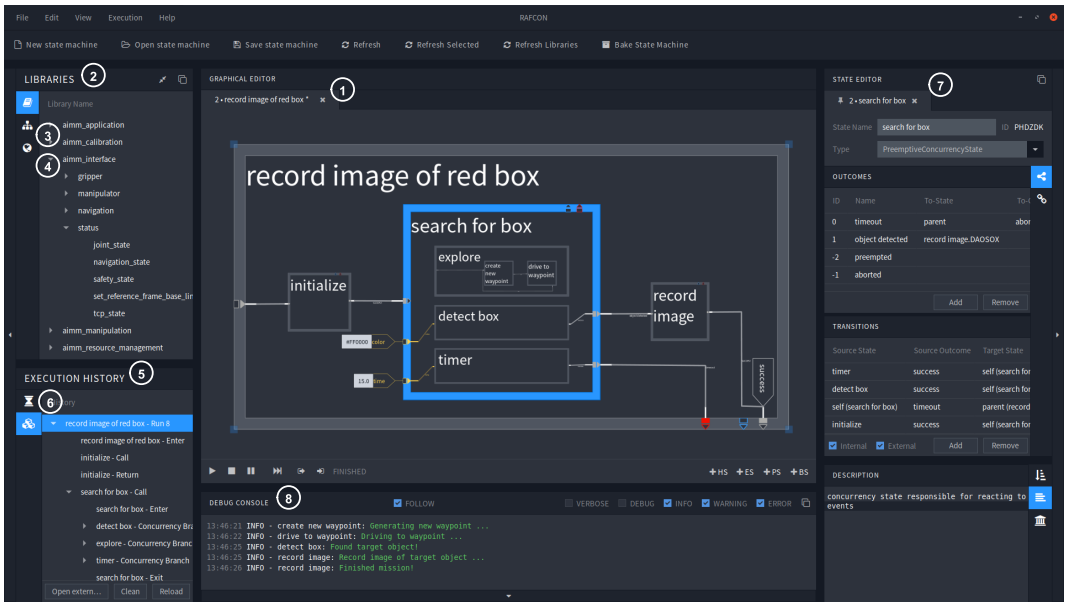


Figure 4.31: *A simple HPFD created in the RAFCON framework. The shown behavior commands a robot to explore the environment until it finds a red ball or a predefined duration is reached.*

in a tree structure. By clicking on a state in the tree the same state in the graphical editor is selected and focused.

(4) *Global variable manager*: holds all existing global variables. For each variable it shows the name, value and data type.

(5) *Modification history*: shows the modification log of the state machine. It implements the undo-/redo- functionality for RAFCON. By clicking on an entry the state of the state machine after this change is restored. A very interesting feature of the modification history is that it allows the user to access different modification branches and not only the main track as common in other graphical editors.

(6) *Execution history*: tracks the execution of the state machine in a tree structure. Each executed state is an element in the tree, each new state machine hierarchy maps to a deeper tree level. By clicking on a state its context data is shown, i.e., all timing, data and outcome information.

The right bar (see label (7) of Fig. 4.31) contains the *state editor*. It shows all relevant information of the selected state. This includes the state properties consisting of a name, a state id, a description and a state type. Furthermore, the state editor has several widgets for modifying the logical state elements, i.e., outcomes and transitions, and for modifying the state's data elements, i.e., data ports, data flows

and scoped variables. The latter ones are an aid to accelerate data distribution on the same hierarchy level and can also be modeled by using data flows only. Additionally, a *Source Code Editor* is part of the state editor. The code editor enables to modify the Python code of ExecutionStates, supports syntax highlighting and checks code automatically for syntax errors. Finally, the *Semantic Data Editor* can be used to add semantic data (as required by the RATN model, see Sec. 4.5.2) for the state into an arbitrary deeply nested dictionary. Possible semantic information is, e.g., different labels for filtering and classifying states or pre- and post-conditions for task planning as described in Appendix D.

The bottom bar has only one element, the *Logger* widget, which is referred to by label (8). The logger supports different log levels (i.e., *verbose*, *debug*, *info*, *warning* and *error*), with different color codings. Moreover, it features structured logs, which include the current timestamp and origin module of each log statement.

The Graphical Editor

The graphical state machine editor (shown by the marker (1)) in the center area of the GUI distinguishes the framework from other flow-control GUIs. It is an editor to view a state machine, modify it and monitor its execution. It allows for creating HPFDs using visual programming and HPFD-GL.

The main feature of this part of the GUI is to be able to handle huge state machines with many hierarchies, states and transitions. In Brunner et al. (2019)) I presented HPFDs with more than 1500 states and more than 2000 transitions. RAFCON renders a state machine in 2.5D, which means that elements further down in the hierarchy are smaller than elements in higher hierarchy levels.

Continuous Visual Abstraction is supported so that the user can navigate quickly and intuitively between the different hierarchy levels. A panning mechanism allows the user to move the view towards a state of interest. Moreover, a user can zoom into a state machine in order to increase the size of the states of interest. As RAFCON zooms into the direction of the mouse cursor, a user can deliberately zoom into a desired child state. By selecting a state, a state editor is created for it in which all state properties can be viewed and updated as described above. If the user zooms out fully a coarse overview of the behavior is shown. The main elements of the behavior can be identified easily as the information of deeper hierarchy layers is discarded by the intelligent zooming concept. This also comes in handy if the behavior is discussed and distributed in a developer team with several people. In summary, the

whole approach is similar to that of digital maps and thus helps to navigate complex information efficiently.

The graphical editor visualizes both, the logical and the data flow of HPFDs. It allows to move, resize and manipulate states by adding ports and connections with via-points. In general, editing state machines is very similar to the way common UML editors handle their graphical elements. The editor serves also for visualizing the behaviors during execution. All currently active states are rendered in green. If the step mode is enabled, the last executed state is visualized in yellow, whereat the final outcome of the state is highlighted in green. Thereby, a user can efficiently track the current execution flow and monitor the behavior's state.

Various viewing modes enable the user to focus on the elements of interest. In the default mode, all state information is shown, which can lead to a very cluttered view. Thus, there is another viewing mode in which all data flows are hidden. Furthermore, the visualization of the labels of the default outcomes *aborted* and *preempted* can be toggled, which otherwise take a lot of space and can thus distract the user.

Additional GUI Features

This section lists additional, important features of the task control framework, which I did not discuss yet.

One benefit of VPLs is that "VPL researchers have the opportunity to seamlessly integrate support for non-coding aspects into the coding environment such as design, testing, and debugging" (Burnett et al. (1995)). We used this opportunity to enhance RAFCON with precious design features, such as the magnet lines to arrange states graphically and movable state ports and labels, and debugging capabilities, such as the execution history view, the console bar and the life data port value introspection (therefore, also see Sec. 5.5).

RAFCON features an easy-to-use plugin concept. This allows other developers to extend the functionality of the framework in different ways. Plugins for calling custom scripts on each execution mode change (e.g., from *paused* to *running*) and for auto-layouting have been written[11]. Moreover, the model checking functionality (as given in Sec. 5.5.1) and the autonomous execution optimization (as described in Sec. 6) are implemented using the plugin concept as well.

Additionally, the flexible design of RAFCON in combination with features of the high

[11] see `https://dlr-rm.github.io/RAFCON/plugins`

level language Python allows for online state machine modifications even during execution. Both, the state machine layout as well as the source code can be manipulated graphically in the GUI or automatically using ExecutionStates. The changes take effect immediately, even without the necessity to save the HPFD beforehand. RAFCON's API can also be used to generate state machines programmatically, e.g., by using a task planner (see Sec. D) or other on the fly state generation methods (as described in Sec. 7.1.1).

Furthermore, the fact that the RAFCON core and the RAFCON GUI are completely separated yields many advantages when using the task control engine on real robots. HPFDs can be developed efficiently on a local PC. Afterwards, it can be deployed to the target system, where it can run without the GUI overhead only using the HPFD interpreter of the core. As the core is very lightweight the CPU overhead for running state machines can be kept low and is thus suitable for systems with low computation power. With a special plugin called the *monitoring plugin*[12] a remote GUI can connect to the execution of a HPFD on a robotic system. When the connection is established information about the execution status of all states is forwarded to the remote GUI. Additionally, the execution can be controlled remotely including the triggering of all mentioned step commands (see Sec. 4.6.1). Several remote GUIs can be attached to a server process, which can either run inside a GUI or using the core only.

Finally, RAFCON is scalable enough to run very complex tasks, such as the one in the SpaceBot Camp scenario (see Sec. 4.7.1), the ROBEX demo (see Sec. 7.1.1) or on AIMM (see Sec. 7.1.2). For the latter use case, we developed state machines with several thousands of states, transitions and data flows. The main reasons that such a complexity can be tackled are the 2.5D view of state machines, the clear, modular design of HPFDs and the explicit modeling of data flows allowing to track data distribution over many hierarchy levels efficiently.

4.6.3 The World Model

As specified in Sec. 4.4.2 the described world model is graph-based. Moreover, as world model information can get large, using a database is a valid approach. Additionally, most databases are optimized for low latency, which makes task control fast as there are many world model queries per task execution (e.g., see 7.4). Furthermore, batch processing features care for high throughput in the case that many requests need to be placed at the same time.

[12] see `https://github.com/DLR-RM/rafcon-monitoring-plugin`

These facts directly lead to the choice of implementing the HPFD world model using a graph database. In our case (see Lehner et al. (2018)), we chose *Neo4j*[13], which is a famous graph database running in Java. Fig. 4.32 shows a screenshot of our Neo4j world model as used in the ROBEX scenario (see Sec. 7.1.1.). Neo4j's query language *CYPHER* (see Francis et al. (2018)) is one of the most used query languages in industry.

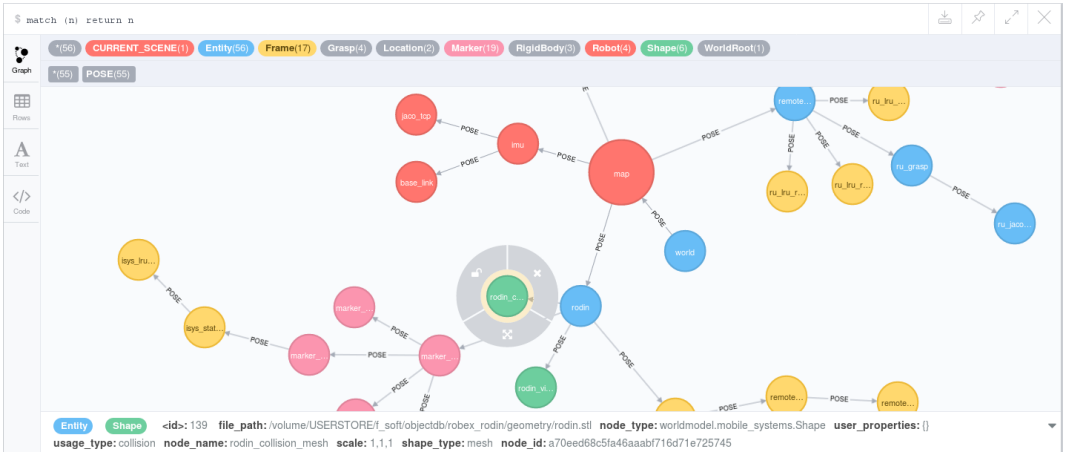


Figure 4.32: *A screenshot of the Neo4j world model as used in the ROBEX scenario (see Sec. 7.1.1.)*

Neo4j holds the data in a property-graph model, which directly maps to our definitions of abstract world model components, i.e., *nodes, properties, edges* and *labels*. Except for complex queries, our world model implementation uses the *neomodel* library[14] to access the data in the Neo4j database. Neomodel's *object-graph-mapping* (OGM) is used to create a type-system for the world model objects with both optional and required properties. For example, the node types *Physical Object* and *Marker* (according to the example of Sec. 4.4.3) can be defined using the OGM. Additionally, rules can be set up so that, e.g., each *PhysicalObject* node has a mass and all *Marker* nodes have a size. For more complex queries we rely on CYPHER, as this is the native query language for Neo4j enabling the access to the full power of the database.

We use YAML files as a human readable format to specify world model information, such as object templates, object instantiations or whole world model states. Object templates are a powerful way to define the objects in an abstract manner. These templates may even contain other objects and edges, and can thus represent small graphs on their own. Those templates can be instantiated efficiently and can be modified subsequently. Our implementation uses Python to access and read those YAML files and to create world model objects using the neomodel library.

[13] `https://neo4j.com/`
[14] `https://github.com/robinedwards/neomodel`

As the world model is a central module, which needs to be accessed by many other nodes as shown in Fig. 4.24 we wrote several adapter nodes. Commonly, different nodes of a robotic system talk via a middleware to each other. In our experiments, we used the ROS middleware for communication purposes. The adapter nodes care for retrieving a specific subset of information from the world model and passing it to other nodes. One example is the generation of the geometric scene of the environment, which is fed to the path planner in order to generate collision free paths for the manipulator. Another example is our tf-world-model-module, which retrieves all spatial information between objects from the database and publishes it via a ROS-Tf-tree[15]. As a matter of fact, the job of the adapter nodes is not only the retrieval and forwarding but also the adding of data to the database. For example, the tf-world-model-module does not only publish spatial information but also receives 3D transformations from other nodes and enters this data into the database.

4.6.4 Resource-Aware Task Nodes

As given in Sec. 4.5 RATNs are basically HPFDs enhanced with structured, semantic information. The Semantic Data Editor of RAFCON's State Editor as described in Sec. 4.6.2 plays a vital role for implementing RATNs. As I will elaborate more on RATNs in Chpt. 6 the implementation of them will be explained in that chapter in detail.

4.7 Application Scenarios

To measure the benefits of a programming language or framework, such as their modularity, flexibility or generality, in numbers is very hard. One way to do so is to compare my framework with others, which I did in Chpt. 3 using several metrics. Another way is to show the use and applicability of HPFDs and RAFCON in challenging application scenarios of different robotic domains, which I performed extensively and summarized in the following section. Next to space-related use cases, also industrial and service robotic use cases are covered. An overview of all use cases covered with RAFCON so far is given at `https://dlr-rm.GitHub.io/RAFCON/projects`.

[15] Concerning ROS-Tf see `http://wiki.ros.org/tf`

4.7.1 Space Related Application Scenarios

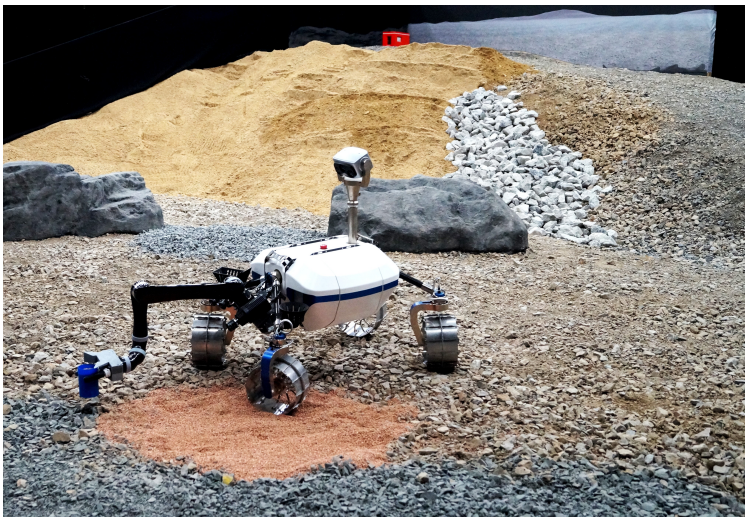


Figure 4.33: *The environment the LRU had to act in during the SpaceBot Camp 2015. A base station, where several parts had to be assembled, was located on a two meter high elevation, accessible only by a steep ramp.*

SpaceBot Camp: The first real project in which I used RAFCON to control an autonomous robot was the SpaceBot Camp, organized by the DLR Space Administration in November 2015. The mission was to built a mobile robotic system able to explore an unknown Moon-like terrain of approximately 200 square meters, to localize and pick up two objects and to transport them back to a base station. The terrain and our robot, the LRU, are visualized in Fig. 4.33. The first object was a battery and the second a soil sample. Back at the base, the battery had to be used to power a scientific device able to investigate properties of the sample. The total time limit for the mission was sixty minutes. A communication delay between mission control and the robot of two seconds (i.e., four seconds round trip time), which simulated the real signal delay between Earth and Moon (see Schuster et al. (2016)), posed an additional challenge. The delay was the main reason, why a high autonomy level had to be achieved by the participating robots.

Ten teams in total participated at the event. Our team *RM-explores* was the only team which accomplished the mission fully autonomously according to the competition constraints (see Schuster et al. (2016)). On top of that, we solved all tasks in only half of the given time limit. Next to sophisticated manipulation, computer vision and navigation modules my proposed task control framework RAFCON was a vital advantage compared to the other teams. It orchestrated all software modules in a centralized, efficient manner that allowed to easily comprehend the choices

the system made. The robotic software architecture strongly resembled the 2T* architecture described in Sec. 2.5.

The robotic behavior, which solved the mission, consisted of more than 700 states and 1200 transitions. We used eight abstraction layers (i.e., HPFD HierarchyState levels) in order to handle the mission complexity. We used RAFCON to trigger high level commands only, such as moving the robot to a specific waypoint or driving the manipulator to a certain 3D position. Single wheel or pan-tilt movements during navigation were part of the lower software modules and not triggered by RAFCON directly.

We strongly benefited from the hierarchical, modular design of HPFDs and RAFCON. Several programmers could develop the state machine at the same time independently and created state machines for solving the sub-tasks for manipulation, grasping, computer vision, navigation and exploration. We discussed the initial strategy of how to solve the overall SpaceBot Camp mission beforehand by graphically sketching the whole mission with abstract high level RAFCON states in meetings, in which experts of various fields participated. As we wrapped the whole robot's functionality in dedicated RAFCON states, the robot's behavior could be updated and improved efficiently by only using RAFCON's GUI. Especially high level mission discussions were much more fruitful by leveraging the RAFCON state machine compared to using abstract sketches or textual descriptions.

The ROBEX Project: We used RAFCON during a complex Moon-analogue campaign called ROBEX. For this project, we programmed several missions for the LRU robot. The final behaviors solving the missions consisted of more than 1500 states and more than 2000 transitions. More information about the mission is provided in Sec. 7.1.1 of the validation chapter.

4.7.2 Industrial Application Scenarios

Mobile Manipulation on AIMM: During Automatica 2018, the leading exhibition for smart automation and robotics [16], RAFCON controlled our AIMM robot to maintain an autonomous supply chain for various workcells. The main HPFD for delivering various material to the different workcells had more than 4000 states and 5000 transitions and was created by nine developers. More detailed information about this

[16] For highlights of the RMC institute at Automatica 2018 see `https://www.dlr.de/dlr/en/desktopdefault.aspx/tabid-10081/151_read-28474/#/gallery/31018`

application scenario is given in Sec. 7.1.2 in the validation chapter.

Programming by Demonstration on KUKA LWR4+: In the RACELab project, Steinmetz and Weitschat (2016) employed RAFCON to program complex skills for an industrial automation scenario with several robotic manipulators. The skills were then used in a programming by demonstration workflow, in which shop-floor worker (non robotic-experts) created various, sequential tasks with a software tool called RAZER created by Steinmetz et al. (2018).

4.7.3 Application Scenarios in Service Robotics

UAV High Level Autonomy Control: *Unmanned Aerial Vehicles* (UAV) such as Helicopters[17] or the High Altitude Platforms as shown in Fig. 4.34 have many interesting use cases such as transportation of goods into terrain not accessible by or hazardous to humans, aerial manipulation, plant inspection, establishing communication networks and data retrieval of the atmosphere for meteorology science. Many of those use cases require the UAVs to act autonomously, as remote control is not possible (e.g., in caves), the target terrains lack networking infrastructure or remote control by humans is too expensive (e.g., in agriculture applications scenarios). Next to acting autonomously, airborne vehicles have to fulfill much higher safety constraints. This is because accidents of UAVs yield much higher risks both for their surroundings (such as humans) and themselves (in a crash UAVs are often a write-off). For these reasons automatic model checking and mission verification have uttermost importance.

Thus, the Flying Robots Group at DLR RMC, which used the flow control framework for their UAVs, enhanced RAFCON by model checking capabilities. With the enhanced RAFCON it is possible to guarantee system properties like: "*While flying never deactivate the motors*" or "*Do not start the motors if there is a person near to the UAV*". More information concerning model checking with RAFCON can be found in Sec. 5.3.

Medical Assistant: In the context of the SAPHARI[18] project, we successfully used RAFCON for human-robot-interaction in a medical assistant scenario. Here, a KUKA LWR4+ could handle different tools such as scissors, scalpels or pliers to surgeons during simulated operation procedures.

[17] Flettner helicopter built by the Flying Robot Group at the RMC institute `https://www.dlr.de/dlr/presse/en/desktopdefault.aspx/tabid-10307/470_read-16914/#/gallery/22188`

[18] `http://www.saphari.eu/`

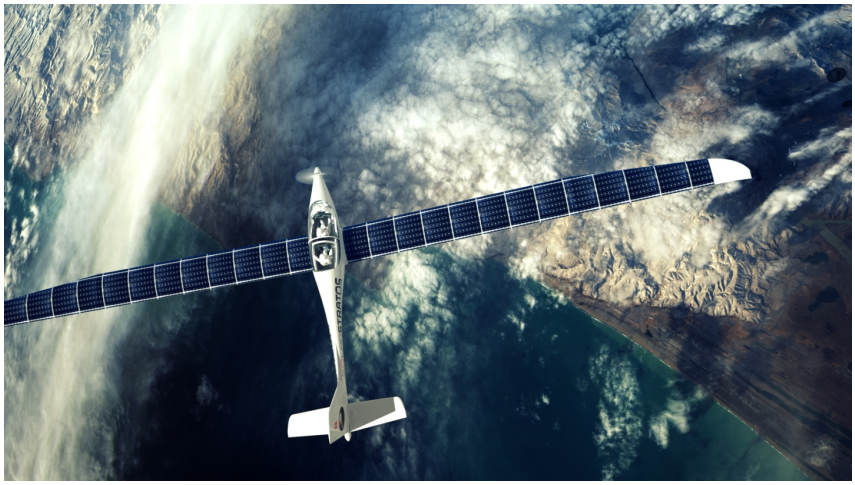


Figure 4.34: *Artistic rendering of a solar-electric high altitude platform flying in the stratosphere.*

4.8 Related Work and Discussion

Many of the task control frameworks introduced in Chpt. 3 use a state machine approach as their underlying model such as SMACH, ROS Commander, FlexBE, ArmarX, Xabsl and Fawkes. However, most of them do not offer the ability of modeling data flows in a dedicated manner (such as ROS Commander, Xabsl, SMACH and Fawkes). Other approaches bind the data flow to the state transitions (such as ArmarX), which is overly restrictive. From the task control frameworks of Table 3.1 only Modelica, FlexBE, Robot Task Commander and RAFCON allow for explicit modeling of control and data flow.

Although graph grammars (as defined by Zhang (2010)) offer an intuitive, clear way to define the meaning of a graphical program, they are rare in the context of visual behavior programming. In fact, none of the frameworks given in Fig. 3.1 offer a graph grammar for the graphical behavior representation. Only Effinger et al. (2010) proposes a graph grammar for his *Hierarchical Constraint Automaton (HCA)*. It is used to create a graphical representation of programs written in the *Reactive Model-based Programming Language (RMPL)*, a textual language for defining reactive robot behavior. On top of that, also Glas et al. (2016) show all building blocks of their graphical language (Interaction Composer) in a clear manner and map them to control flow building blocks, which is very similar to the graph grammar approach.

Modeling the knowledge of a robot about its environment is actually a huge topic, which I cover only to some extent in Sec. 4.4. There are many concepts in literature

describing how complex data can be stored in a structured manner and easily be queried (see Blumenthal et al. (2013) and Margitus et al. (2015)). Graph database based approaches are just one possibility, which, however, promise to be a scalable solution (Blumenthal et al. (2013)). In practice, many proposed frameworks do not offer a (mature) world model concept, such as FlexBE, ROS Commander, Modelica, Matlab Simulink and LabVIEW. This does not mean that they cannot be interfaced with a world model, they just do not present one in their framework description. However, there are also frameworks which offer very powerful knowledge processing capabilities, such as CRAM in combination with KnowRob (see Beetz et al. (2018)). It supports data acquisition from ontologies (also located on remote servers), from general instructions that can be found in the world wide web and from natural language instructions.

Chapter five

Design, Execution and Post-Mortem Analysis of Robotic Behavior

Next of being able to program complex robotic behavior, sophisticated task analysis concepts are needed to support different stakeholders during the different lifecycle phases of robot behaviors, i.e., during the pre-, the runtime- and the post-mortem phase.

5.1 Analysis Goals

Logging and analyzing the execution of robotic tasks has multiple goals, whereat three major areas are *debugging*, *monitoring*, *profiling* [1].

Debugging is especially necessary during designing robotic behavior. Actions of various abstraction levels have to be defined and arranged in order to create reactive, concurrent or goal driven behavior. The ability to debug occurring errors efficiently is vital for each task control framework. The programmer must be able to get deep

[1] *Learning* is another reason for logging data, e.g., to feed semantically logged images into the training of object classifiers/localizers. However, this topic will not be covered in the scope of this work.

insights into the reasons why specific choices were made during behavior execution. All the different abstraction layers of the behavior need to be easily accessible by the programmer. As for robotic systems a lot of data accumulates during task execution, the data relevant for error inspection must be extensive and efficiently accessible.

Logging data is not only useful for error debugging but also for error detection during execution, commonly called task **monitoring** (see Brunner et al. (2018)). In the case of an error it must be easily possible to pause or stop the current mission. After inspecting the robot's current situation the mission can be optionally updated and finally resumed. The more critical a mission is, the more important are proper monitoring possibilities.

In many scenarios, tasks must often not only be solved effectively but also efficiently. Especially, time constraints have to be accounted for. Thus, **profiling** concepts able to analyze the mission status and generate information about performance bottlenecks are essential. This can be achieved by analyzing log data recorded during behavior execution. As given in the introductory space rover example (see Sec. 1.2.3) the time available for mission execution can be limited, thus, efficiency is essential. Based on the identified performance bottlenecks, the robotic system can be updated selectively, e.g., by optimizing existing or creating better algorithms, and making behavior execution more efficient.

In Sec. 5.3, I present an architecture to reach these goals. Prior to that, I will highlight the log data processing benefits of grounding the semantic information of RATNs in ontologies.

5.2 Linking Ontologies to Resource-Aware Task Nodes

During task execution large quantities of log data accumulates, such as image data, telemetry and house keeping data, information about the robot's environment, decision logs of the task control process, and the internal state of various software modules. Logging can be performed in a continuous fashion, event based or triggered by the task control module. In order to interpret such data, context information is required. Semantic description of a task is essential to put the log data into context.

RATNs are a powerful semantic action model able to be used for a range of use cases (as shown in Sec. 4.5.4). So far, the programmer can add arbitrary semantic information to a RATN, such as needed resources or classification information. Grounding this semantic information using ontologies yields a huge benefit: accessing and

filtering the logged data can be performed in a standardized fashion by only using the ontologies. The difficulty of inferring context from not explicitly encoded data, or requiring information about the whole ecosystem of the robot in order to know the data types (such as 3D pose or SI unit conventions) of the logged raw data can be drastically reduced.

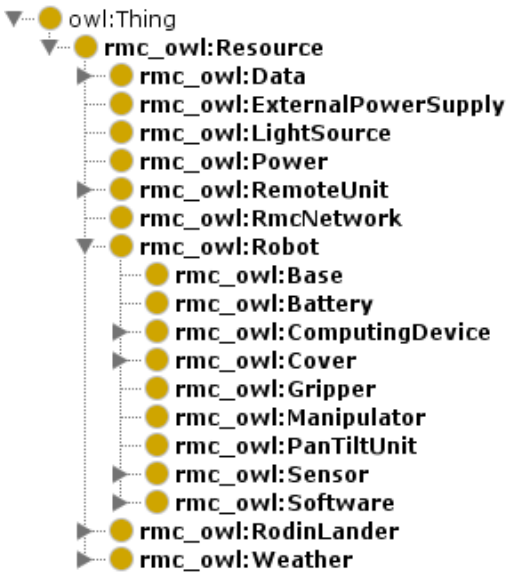


Figure 5.1: *An excerpt of the resource ontology as used in the ROBEX project (see Sec. 7.1.1). I used Protégé by Noy et al. (2003) to create the ontology.*

Ontologies can be used to create object taxonomies (i.e., relating entities only using the 'is-a' relation: e.g.: water is a liquid) or to model more complex relations between objects (e.g., 'water extinguishes fire' and 'water cools hot components'). An example ontology is given in Fig. 5.1. I used it to model all relevant resources needed by various actions during the ROBEX project on Mt. Etna. Next to the list of existing components it shows the membership relation using the tree structure, i.e., *rmc_owl:Manipulator* belongs to the class of *rmc_owl:Robot* resources. Ontologies ground abstract, machine-readable symbols to concepts and objects in the real world (see Tenorth and Beetz (2009)). Thus, filtering log data by only referring to these real world entities is possible. This allows for intuitively writing powerful data queries by people that do not have to be robotic or domain experts. Appropriate ontologies are the key for a suitable classification of actions and thereby for proper classification of recorded raw data. Example queries, visualization and graphs which can be generated using such ontologies are shown in the validation Chpt. 7.1.

5.3 An Architecture for Whole Lifecycle Analysis

Fig. 5.2 shows the overall architecture of the whole-lifecycle analysis concept, which forms the major part of this chapter. The figure is split in three rows and three columns. Each row shows one of the three lifecycle phases of the iterative development of a robotic behavior, i.e., the *design phase*, the *runtime phase* and the *post-mortem phase*. The first column shows all stakeholders involved in the respective lifecycle phase. The second column lists the actions performed by the respective stakeholders. The third column shows the concepts of how to analyze the progress and the quality of each phase. The analysis concepts aim to reduce the stakeholders' time needed for development, verification and improvement of the robotic behavior. We realized all of those concepts in RAFCON and tested the majority of them on the LRU in the context of the ROBEX project (see Section 7.1.1). In the following, all analysis concepts are explained.

5.3.1 Design Phase

In the design phase the robotic behavior is created. Basic actions are defined and more complex robotic skills of various abstraction layers are built with those actions. Furthermore, semantic information is added to these skills by using ontologies. This includes classifying skills, and defining required resources and other RATN-related dependencies. The overall mission is finally constructed out of these high level skills. As depicted in Fig. 5.2 this is done by the behavior developers. This phase can be supported by various analysis concepts.

Test (incl. Simulation): The first method for analysis is to apply state of the art testing. This includes unit tests, module tests, integration tests and system tests. Unit tests aim for checking the correctness of single functions. For module tests, small simulations covering only the scope of the respective module under test can be employed. Integration tests check the interaction of several modules and processes. System tests, require a complete, comprehensive simulator to test the system's software in the whole. As these concepts are common practice for every larger software system, I do not elaborate on this topic.

Code Syntax Checks: Syntax errors of the HPFD ExecutionStates can be revealed by static code analysis, such as linters. This can be automatically performed during editing of the code snippets or after saving. For interpreted languages (such as Python

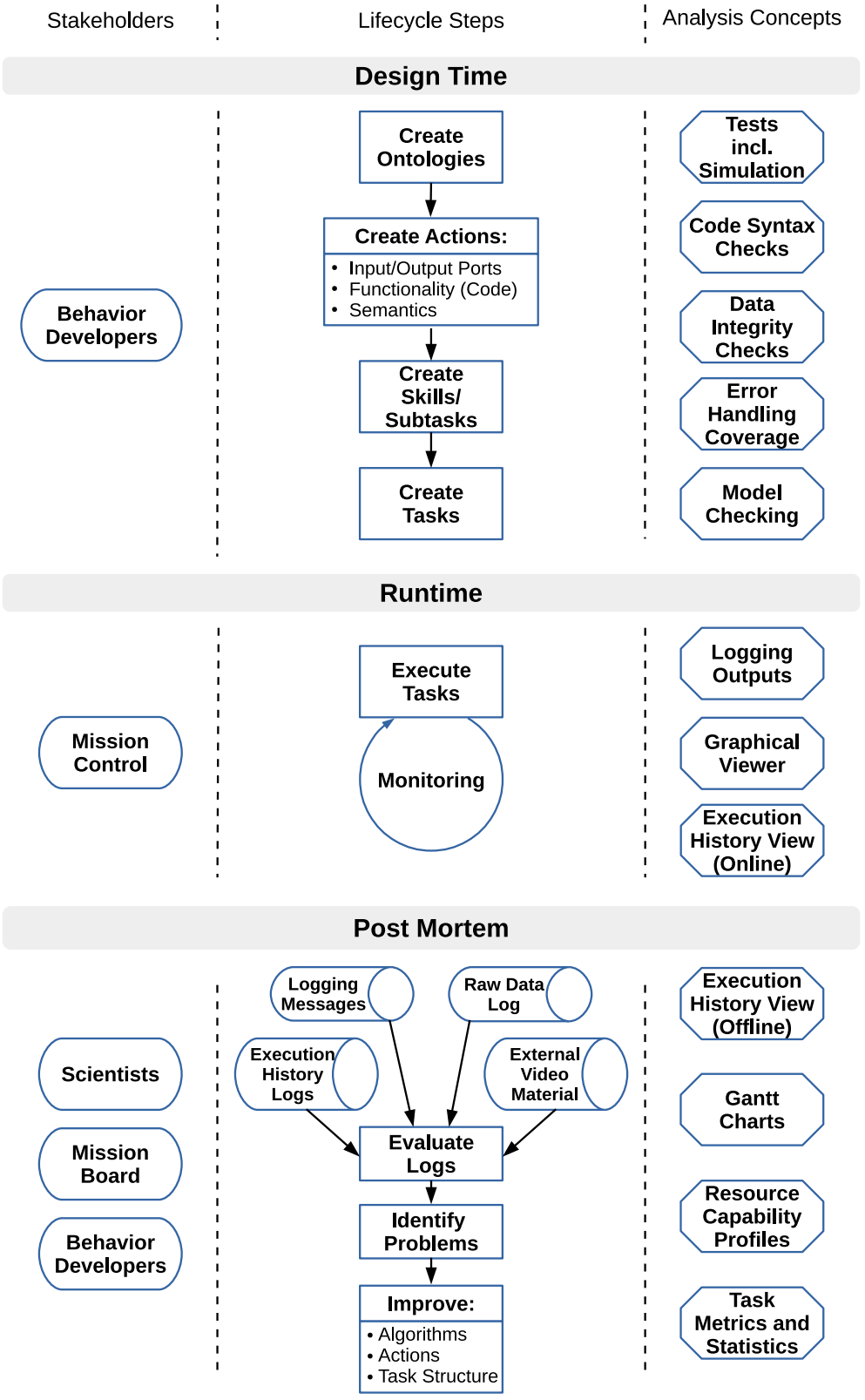


Figure 5.2: *The whole-lifecycle analysis architecture: the left column shows the responsible stakeholders for each life-cycle step, the center describes the main steps performed by the stakeholders, and the right column shows possible analysis methods.*

or Lua) this is especially valuable as typing errors are otherwise not revealed before runtime.

Data Integrity Checks: Actions, as defined in the HPFD formalism, are parameterized with various input and output ports of different data types. Connecting those ports with data flows must ensure type consistency by either requiring the exact same data type or by respecting the inheritance structure. This means that an object of a specialized class can be forwarded into an input data port with its base class as data type (but not the other way round!). Fig. 5.3 shows an overview of all important data type consistency checks. These checks are always performed before the task execution starts in order to ensure the syntactic correctness of the program.

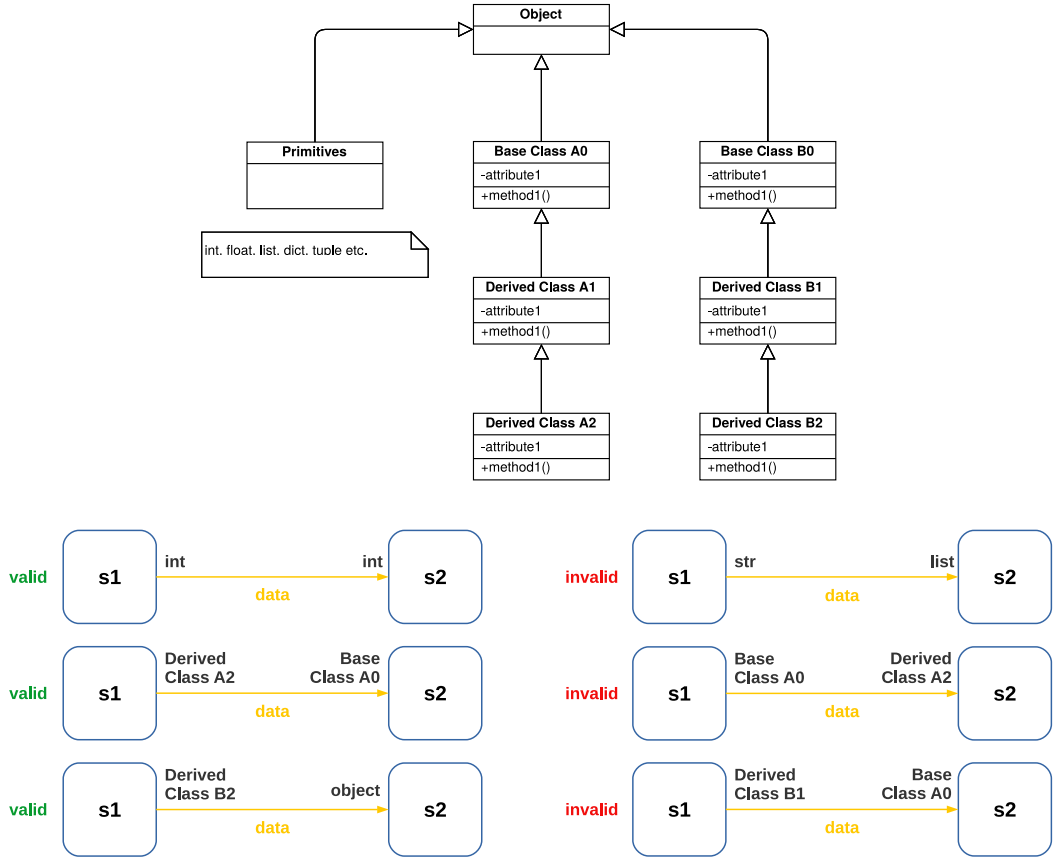


Figure 5.3: *Top: an exemplary class diagram including generalization; bottom: exemplary data integrity checks, as enforced for HPFDs by the whole-lifecycle analysis architecture.*

Error Handling Coverage: In real robotic application scenarios every action has a certain probability to fail. Thus, recovery routines and error detection have to be implemented for critical tasks. This especially holds true for missions requiring longterm autonomy, i.e., autonomous behavior for longer periods of time. The longer the system runs, the higher the probability of the occurrence of errors, which have to

be handled by the robot autonomously. In principle, there exist two types of error detection. The first type is the synchronous error response. Every robot action has to return exact information about its outcome, i.e., if the action was successful or not. In the case of an error, the task has to return the error reason, e.g., in case of a grasp action, if the contact force of the grasp was significantly off to the expected one, if the gripper could not reach the commanded position, or if the gripper driver itself failed and did not respond. The second type is the asynchronous error monitoring by using dedicated error observers. This is required for all critical environment states of the robot. Typical examples are observers checking if an object remains grasped during manipulator movements, or monitoring the robot during navigation in order to interrupt the current action if a certain distance or time threshold is met.

Next to detecting an error, a robust behavior must also be able to recover from it. Recovery procedures have to be implemented for every action that has high failure probability. For the synchronous case, error handling can be either implemented straight forward by connecting the failure outcome to a routine which handles the error correctly or chosen by, e.g., procedural reasoning or semantic planning (see Appendix D). In the case of asynchronous observers an error routine for each environment state under observation has to be defined.

In both cases, defining error handling routines for all HPFD or environment states does not scale, as HPFDs can consist of several hundreds or thousands of states (see Sec. 7.1.1). Therefore, modularity and reusability of error routines are required. A key element of a scalable architecture is that errors can be forwarded to the next higher abstraction level until a generic error handler can take care.

For developers the so called “error handling coverage” metric provides insights of how robust and error-tolerant a state machine is with respect to synchronous error handling. In order to define it formally (for the prerequisites, i.e., the HPFD formalism, see Sec. 4.1.2) the *parent set* of a state is introduced:

Definition 13 (Parent set) *The parent set of a state s consists of all states p_i that are elements of the path from s up to the root state r of a RST. The parent set excludes s and includes r. The parent set of s is written $parentset(s)$.*

Equation 5.1 provides the formula for calculating the error handling coverage $ehc(sm)$ of a state machine sm. S_{sm} denotes the set of all states of a state machine sm with root state r. $S_{eh,sm}$ denotes the set of all states s_i (with $0 \leq i \leq n$ and n being the total number of states of sm), which are provided with direct or indirect error handling. The final error handling coverage of a state machine is simply the ratio of states with error handling $S_{eh,sm}$ and the total number of states S_{em}.

$$ehc(sm) = \frac{|S_{eh,sm}|}{|S_{sm}|} \tag{5.1}$$

$$S_{eh,sm} = \{s_i\}, 0 \leq i \leq n : deh(s_i) \vee ieh(s_i) \tag{5.2}$$

A state has direct error handling, written $deh(s) = True$, if it has a state connected to its *aborted* outcome via a transition t. This is defined by the following equation:

$$deh(s) = \begin{cases} true & \text{if } \exists t : t = (l_1, l_2) : l_1 = (s, o) : o = aborted \\ false & \text{otherwise} \end{cases} \tag{5.3}$$

A state has indirect error handling, written $ieh(s) = true$, if one of its parents has direct error handling:

$$ieh(s) = \begin{cases} true & \text{if } \exists p \in parentset(s) : deh(p) = true \\ false & \text{otherwise} \end{cases} \tag{5.4}$$

In addition to the error handling coverage the average error handling depth is of interest. The error handling depth of a state is the sum of all error handling procedures of all the parent states of s up to the root state. It is calculated as shown in equation 5.5

$$ehd(s) = \begin{cases} 0 & \text{if } s = r \text{ (= root state)} \\ 1 + ehd(parent(s)) & \text{if } deh(s) \\ 0 + ehd(parent(s)) & \text{otherwise} \end{cases} \tag{5.5}$$

The average error handling depth *aehd* of a state machine *sm* if finally defined as

$$aehd(sm) = \frac{\sum_{i=0}^{n} ehd(s_i)}{|S_{sm}|} \tag{5.6}$$

In the validation chapter, several experiments are presented that calculate the error handling depth in order to make statements about the robustness of the analyzed behavior.

The error handling coverage is a quantitative metric to analyze the robustness of an autonomous behavior. In general, it only states if and how many error handlers are defined per state. If the handler is sufficient for a specific error in various contexts is not covered by the error handling coverage and depth.

Model Checking: Model checking and model driven development are essential for critical missions. Critical can either refer to system critical, i.e., if some condition does not hold the robotic system can take severe damage, or critical to the environment, i.e., if the robot fails then there can be serious injuries for humans or expensive damages to external equipment. One of the most severe and expensive accidents of mankind was the Ariane-5 launch on June 4, 1996 (Baier and Katoen (2008)). A simple software bug (a type conversion from a 64-bit floating point into a 16-bit integer value) was responsible for the mission failure, which could have been easily prevented using model checking (Baier and Katoen (2008)).

The general picture of model checking is rather simple and is shown in Fig. 5.4. Based on a system description, the system is modeled using a suitable system modeling language. At the same time, crucial system requirements are listed and formalized using a property specification language. The system model and property specification are passed to the model checker, which checks whether the system model satisfies the defined properties. If this is not the case, the model checker returns a model failure in the form of a counter example, i.e., a sequence of internal system states leading to a case in which some of the specified properties do not hold.

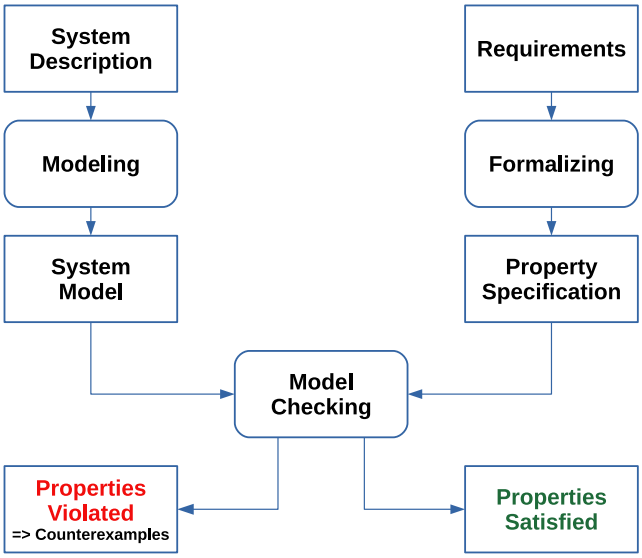


Figure 5.4: *The general model checking methodology as presented by Baier and Katoen (2008).*

For each robot system different properties are critical and must hold during the

whole mission execution. For example, the motors of an UAV must never be turned off while it is in the air. Such properties have to be ensured automatically for all possible system states. Manual checking would be much too costly because its too work intensive and error prone and (in practical uses cases) simply not possible for programs with high numbers of states.

Using model checking for system analysis has several advantages. A model checker can prove different kinds of properties for a software system, e.g., the existence or the non-existence of certain variable values, reachability of certain states, and the existence of deadlocks, life-locks, starvation, and race conditions. Model checking can also be used for partial system verification, which means that specific properties can be checked individually without the need of a complete requirements specification from the beginning. Furthermore, model checking is "not vulnerable to the likelihood that an error is exposed. This contrasts with testing and simulation that are aimed at tracing the most probable defects"(see Baier and Katoen (2008)). Finally, the fact that a property violation always returns an example for the violation (a counter-example for the specified property) is very beneficial for debugging purposes of the system model.

However, model checking has several weaknesses as well, as stated by Baier and Katoen (2008):

- It is mainly appropriate to control-intensive applications and less suited for data-intensive applications as data typically ranges over infinite domains.
- It verifies a system model, and not the actual system (product or prototype) itself; any obtained result is thus as good as the system model.
- It checks only stated requirements, i.e., there is no guarantee of completeness. The validity of properties that are not checked cannot be judged.
- It suffers from the "state-space explosion problem", i.e., the number of states needed to model the system accurately may easily exceed the amount of available computer memory. Despite the development of several very effective methods to combat this problem, models of realistic systems may still be too large to fit in memory.

The second bullet point aligns with the "the code is the design" principle. In the opinion of several authors model specification languages are only one view on the overall system design, with the source code being the only complete description of a system (see Reeves (1992), Martin et al. (2003), and Ye (2018)). As model checking only checks the part of the system that is modeled by the model specification

language, it is per design incomplete and can even validate completely irrelevant properties if the system model and the system implementation do not align. Analysis methods that check the actual behavior itself (i.e., the source code being executed) such as Code Syntax Checks, Data Integrity Checks and Error Handling Coverage do not suffer from this misalignment problem.

In summary, model checking can significantly increase the confidence level of system design, but cannot be the only way to ensure the system's integrity. This is the reason, why model checking is only one of many elements of the presented whole-lifecycle analysis architecture for HPFDs.

5.3.2 Runtime Phase

During task monitoring there are several approaches to assist the operations team and mission control. Monitoring tasks for critical systems can be very intricate as next to the housekeeping data and the environment of the robot, especially the task related data of mission execution has to be observed. The data includes all information about the progress of the task, including all detected and manipulated objects, the current state of internal modules (such as if the rover is currently localized) and the status of all goals. To support the mission control during the execution of a task, two monitoring tools are proposed apart from classical logging.

Logging Outputs: Logging debug output to the console or dedicated files is the classical way of making the current state of the system accessible. Each state can print out different information using different log levels, i.e., info, debug, verbose, warn and error. Depending on the log level, the severity of the logs can already be inferred. The clear goal is to avoid system breakdowns completely. The presence of many warnings or errors is a hint that system health degrades. In such a case, the mission control team has to interfere and at least pause the execution of the current behavior

Graphical Viewer: A 2.5D graphical viewer is able to efficiently show the internal representation of a robotic behavior in one consistent graph, including the currently executed actions and all past and future ones. The 2.5D can not only visualize the actions of the current abstraction level but also the ones at lower levels (see Sec. 4.2.2 for Continuous Visual Abstraction). In order to track the parameters of each HPFD state live data introspection is a powerful approach. Next to the input and output port of each action the current data content is visualized. As the space for the graphical viewer is limited, ports with complex data need a way to compress

or crop the depicted data.

Execution History View: The execution history view is one of the most powerful debugging and monitoring approaches in the architecture. It allows to inspect each executed action including all its context data. The context data consists of all input and output data of the state itself and its parent, the start and finish times of the action and its final outcome. The actions are visualized in a tree structure, in which each tree level maps to a hierarchy level in the HPFD. A specific action can be selected in the view, which reveals all its context data. In principle, it is very similar to the call stack visualization offered during debugging by a common IDE: The user first selects the thread and the function in the thread's call stack and can then introspect all function variables and the members of all accessible objects.

The implementation section shows the benefit of these concepts by giving examples. Furthermore, the validation section will refer to these concepts and highlight their advantages in the context of real and simulated missions.

5.3.3 Post-Mortem Phase

Most of the stakeholders are interested in the third phase of the whole-lifecycle analysis architecture. At this stage the success of the mission can be evaluated and performance bottlenecks can be identified. This information can be used in order to guide development efforts focusing on the critical aspects of new missions. The post-mortem analysis concepts make up the major part of the architecture shown in Fig. 5.2.

Execution History View: The execution history view is not only valuable during runtime but after missions execution as well. Next to reconstructing the task, all decisions made by the task control during execution can be analyzed using the tree structure. In fact, the logged execution traces are the most essential data sources for calculating task statistics. With the timing information of the start and end timestamps of each action, semantic queries can be created able to access raw data streams using the time information as index. Different parts of the raw data stream can thus be mapped to the semantics of that actions that were executed at the timestamps of interest.

Gantt Charts: Gantt charts belong to the set of essential tools for tracking the execution order and for comparing the execution times of actions. They can be generated based on the timing information stored in the execution logs. Arbitrary time windows can be defined to focus on the interesting events in the Gantt chart.

An example Gantt chart is given in Fig. 7.8 showing the execution order of actions during one of the ROBEX missions.

Resource Capability Profiles: To allow the analysis of the resource usage of all actions during task execution a special chart type similar to common *Resource Capability Profile* (RCP) is used. The resource dependencies of RATNs are the basic information needed to generate such diagrams.

The charts in Figure 5.5 show the evolution of the different RCP styles.

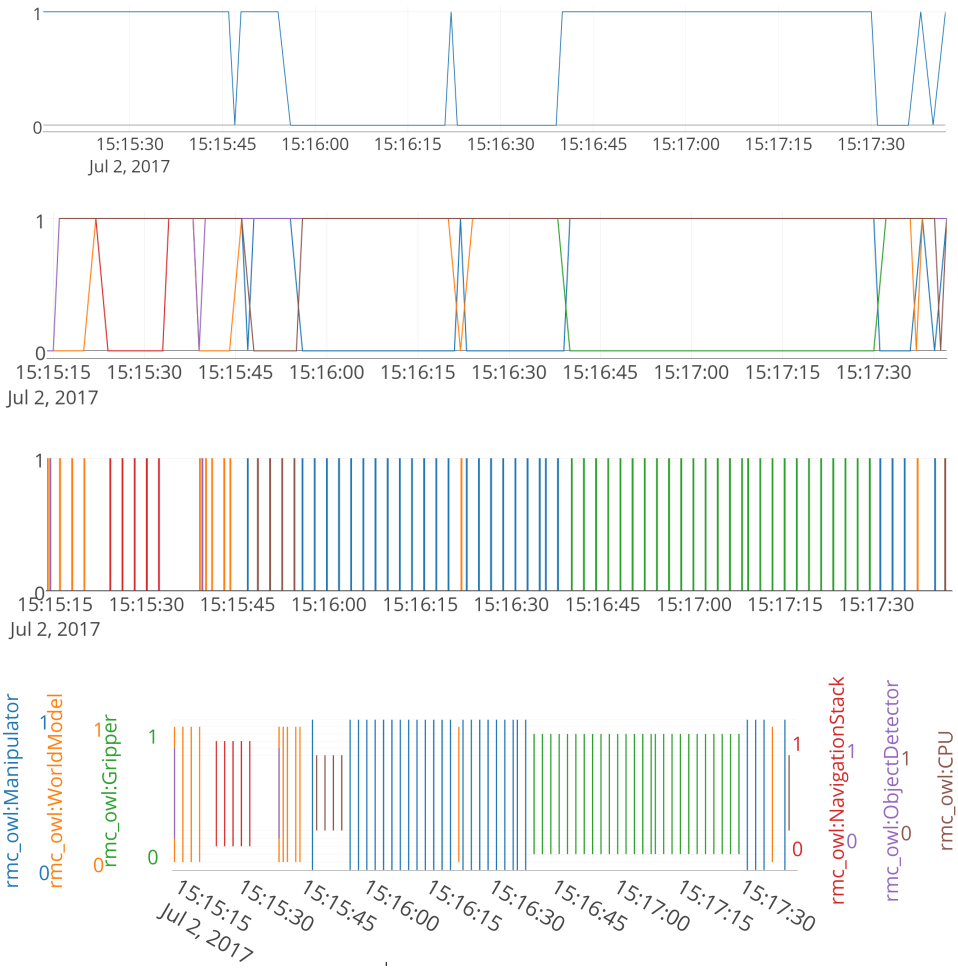


Figure 5.5: *Different types of resource capability diagrams; from top to bottom: Resource Capability Profile (RCP) with a single resource, RCPs for several binary resources, Inverted Resource Capability Profiles (IRCP) as bar plot, Inverted Resource Capability Profiles with Shifted Y-Axis (IRCP-SYA) as bar plot. The bar plots show the resource usage sampled in a 2 second discretization*

RCPs can visualize a single resource (see the **first** sub-figure of Figure 5.5) or even several resource at once (see the **second** sub-figure of Figure 5.5). It is obvious that the information retrieval efficiency of the second is low because of the overlapping

of the plot for the different resources. Thus, I give more efficient variants of how to draw several resources into the same graph in the next two sub-figures. The sub-figures in the rows two, three and four all visualize the same data. Sub-figure four eases the information retrieval significantly compared to sub-figures two and three, as the overlap of the lines is drastically reduced.

In the **third** sub-figure the graph is inverted, i.e., it shows the resources in use and not the remaining ones. The reason for this inversion is that a lot of resources just have a total capacity of one (i.e., binary resources). Thus, in most cases, showing the usage of the resource is more significant than its sole availability. The usage maps to the event of interest, whereas the availability is just the default state. For resources with a higher availability such as the power of the robot, the non-inverted approach is used.

The reason why I use bar plots instead of scatter plots is that scatter plots are not convenient for zooming (see, e.g., Plotly (2017)). As a lot of data of long time periods (tested up to several hours) can be plotted, the user must be able to zoom into the interesting spots. Unfortunately, scatter plots suffer from a lot of white space inside the graph. This makes it very hard for the user to navigate inside the graph (i.e., translate the view or zoom in even further). Thus, I rely on bar plots.

If several resources are used at the same time the bars of the bar plot overlap. Thus, resources drawn below other resources cannot be seen anymore. Visualizing the bars in different transparency levels, quickly confuses the user as transparently overlapping bars produce new colors: in the worst case colors that are already reserved for other resources. This is the reason why the resource usage is sampled (e.g., in a 2s resolution). By using a small offset for each resource, several resources can now precisely be plotted into the same graph.

On hovering over a bar (i.e., a resource sample) all context information is shown.[2] This enables the mapping between the resource sample to the respective action. By shifting and scaling the y-axis of the graph different types of resources can now be included into one graph (see the **fourth** subfigure of Figure 5.5). This final type of RCPs is called *Inverted Resource Capability Profile with Shifted-Y-Axis using Bar plots* (IRCP-SYA-Bars) (see Brunner et al. (2018)). Ultimately also classical RCPs can now be integrated into the same graph, as shown in Fig. 7.7.

Task Metrics and Statistics: Based on the log data various metrics and statistics can be computed. There are two groups of metrics.

[2] See `http://rmc.dlr.de/rm/de/staff/sebastian.brunner/task-analysis` for multiple real RCPs of different types created with Plotly (see Plotly (2017)). These RCPs are based on real data of the ROBEX mission described in Sec. 7.1.1.

On the one hand, there are efficiency metrics. The total runtime of single primitives, skills, tasks and the whole mission can be of interest. Also, the minimum, maximum and mean action execution time are significant numbers concerning their efficiency. Putting the numbers of various sub-tasks into relation with each other or with the total mission time can finally be used to identify bottlenecks.

Let a_j be an action type as defined in the RATN formalism. In the robotic behavior b there are k action types in total. j refers to the action type index (i.e., $0 \leq j \leq k$). Let d_b be the execution time (i.e., the duration) of the behavior b, and $d_i(a_j)$ the ith execution time of action type a_j, with $0 \leq i \leq n_{a_j}$. The relative execution time consumed by action type a_j, written $ret(a_j)$ is given by:

$$ret(a_j) = \frac{\sum_{i=0}^{n_{a_j}} d_i(a_j)}{d_b} \tag{5.7}$$

An individual weight for each action based on the total number of action executions needs to be taken into account, as actions only performed once in several hours are not a good candidate to focus optimization upon. The individual weight w for an action type is calculated by:

$$w(a) = \frac{n_{a_j}}{\sum_{j=0}^{m} n_{a_j}} \tag{5.8}$$

The weighted relative execution time $wret$ of an action type a_j is finally calculated by:

$$wret(a_j) = w(a_j) \times ret(a_j) \tag{5.9}$$

It is reasonable to focus the optimization on the action types $a_j s$ with the highest weighted relative execution time. All statistics can be calculated for single (sub-)tasks or the whole mission.

On the other hand, also robustness metrics can be calculated. Next to the error handling depth and coverage of the design phase, errors that actually occurred and were not caught and error handling routines that were triggered can be counted and put into relation. Let $o_i(e_j)$, with $0 \leq i \leq n_{e_j}$ and $0 \leq j \leq m$, being the ith occurrence of the error type e_j (with m different error types in total). For filtering and grouping of errors of interest a reasonable error classification is required. This means that the error types e_j need proper semantics in order to be meaningful (such as given by Beetz (2000)). The relative error severity res for an error e_j is calculated by:

$$res(e_j) = \frac{n_{e_j}}{\sum_{k=0}^{m} n_{e_k}} \tag{5.10}$$

The relative error severity statistic is only meaningful when several runs of a robotic behavior are taken into account, as an uncaught error normally means the termination of a behavior execution. However, the relative error severity statistic can also be calculated only for all caught errors, denoted as $resc(e)$. The calculation for $resc$ is analogue to equation 5.10, with the difference that only error occurrences $o_i(e_j)$ are taken into account if they were caught, i.e., if an error handling procedure took care of the error.

The presented statistics have been calculated for several scenarios given in the validation chapter and help to identify performance and robustness bottlenecks.

5.4 Online Mission Update

During the runtime phase, mission updates might be required in order to adapt to environment changes, hardware failures or changes concerning the mission objectives.

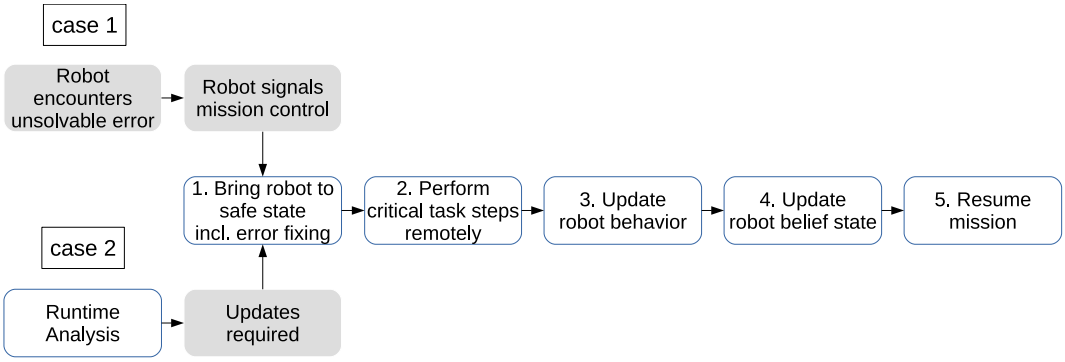


Figure 5.6: *Mission update procedure during the runtime phase. Two cases are visualized: 1. shows the case, where the robot encounters an error that cannot be solved by itself; 2. shows the case that the mission control wants to update the robotic behavior because of, e.g., changing task objectives which are based on new insights about the target area. Mission states are shown using gray boxes, actions required by mission control are shown using white boxes.*

Fig. 5.6 shows two cases in which an online mission update is required. The first case covers errors or events the robot recognizes and cannot resolve on its own. Upon detecting the error the robot contacts the mission control and asks for new

instructions. This case includes only situations, that can be modeled using the robot's semantics, i.e., situation that the robot "understands".
The second case covers situations in which either the target objectives change (e.g., because of new findings) or in which the robot faces an error which it does not detect or cannot model. Examples for the latter situation would be that the robot tries to reach a certain POI, which is located on an elevation, and because of the high slippage the rover does not reach its goal. In the first case, the robot is the initiator of a mission change, in the second case the mission control itself has to act and has to initiate the mission change. In both cases, the procedure for mission control is similar.

The **first step** consists of bringing the robot into a safe state. First and foremost, this step includes the identification and analysis of all errors by using the proposed tools for Runtime Analysis. If the identified error is software related, the respective modules have to be reset and re-configured. Otherwise, if the error is hardware related, two cases have to be distinguished. The first case includes all situations in which the robot's misbehavior is based on critical contacts to its environments, e.g., one of the robot's wheels is stuck or the manipulator is clamped. By analyzing the situation correctly the mission control might be able to resolve the problem by, e.g., retreating the robot's manipulator from undesired contacts or moving the robot's base to another spot. The second case includes all permanent hardware damages to the system (e.g., because of stellar flares, or a broken link because of a failed manipulation attempt). In this case the mission control has to adapt the mission objectives. If only the mission objectives are going to be changed (i.e., the case where the mission control initiates the change), this step simply consists of pausing the robot at a suitable point during task execution.

In the **second step**, the robot is remotely commanded to perform critical actions in step by step fashion in order not to encounter the same errors as before. This might include the repositioning of the robot in order to try a manipulation action from another relative position to the target object or analyze a target object from different perspectives. In the case that simply the mission objectives change this step might be skipped.

The **third step** consists of updating the robot's behavior (i.e., the HPFDs) such that the probability of the encountered error is reduced or the new mission goals can be achieved.

In the **fourth step**, the robot's belief state has to be adapted in order to reflect the effects of the encountered error or the effects of the modified task objectives. In the former case, this might include to constantly block a resource (e.g., the robot's

pan-tilt unit in case it is broken). In the latter case, this could mean to add further waypoints the robot has to explore or further POIs the robot has to investigate.

The **final step** consists of resuming the mission. This includes triggering the correct re-initialization procedures and loading the updated belief state. On top of that, situation specific actions might be required, e.g., the scanning of the direct surroundings of the robot in order to get an updated local obstacle map needed for autonomous navigation.

The in-depth analysis of error reasons and effects and the creation of a powerful error taxonomy, able to describe all major errors the robot can encounter during such missions, is out of the scope of this work. However, in the evaluation chapter 7, I will provide multiple examples describing common errors and error handling approaches. Particularly, I will highlight examples for all steps of the online mission update procedure in the context of the ROBEX mission.

5.5 Implementing the Analysis Architecture in the RAFCON Framework

The whole-lifecycle architecture is a domain-independent, universal approach and thus implementation-independent. Yet, it poses many challenges for the underlying target task control framework as it covers all major lifecycle phases. In the following, I describe how I have used and extended the task control software RAFCON to meet all the requirements of the analysis architecture.

Before explaining the implementation for the single analysis features, I will highlight the logging system of RAFCON which is extensively used for the runtime and post-mortem phase. The RAFCON execution engine generates logs for each state execution. A very precious source of log data are the data flows that are modeled for each state. Figure 5.7 shows various execution commands for an example state machine including fine-grained execution control possibilities. Furthermore, the figure summarizes the data logged for each state execution:

- Input / Output Data: All input and output data of the action either passed via data flows or taken from the default values including all return values
- Scoped Data: All the scoped data of the parent state, which combines all input and output values of all executed siblings

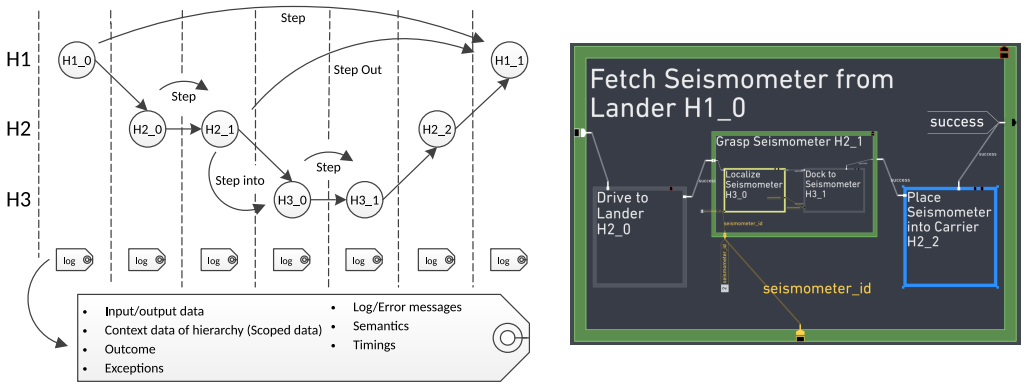


Figure 5.7: *On the left the execution architecture of RAFCON is shown by utilizing the state machine on the right, which is inspired by the ROBEX scenario (see Sec. 7.1.1) and has three hierarchies H1 - H3. Every state name starts with its current hierarchy level, followed by a unique number for its own level. On the left, the arrows show the start and end states of various execution commands. By executing the command given in the label of the arrow, RAFCON's state machine interpreter executes all states between the start and the end state. Execution steps are separated by dashed vertical lines. The table below the left figure lists all data that is recorded when executing each state.*

- Final Outcome: The final outcome of the state, which can either be user defined or built in such as the 'preempted' and 'aborted' outcome
- Exceptions: All occurred exceptions during state execution
- Log and Error Messages: Common log and error messages of each state
- Semantics: The semantic data of the task node, which consists of RATN information such as semantic dependencies and action types
- Timing information: The start and end timestamps of the state execution.

The logged data for each state is extensive and heterogeneous. Hundreds of megabyte log data accumulated every day for the ROBEX mission on Mt. Etna. Nonetheless, it is clearly structured when taking the underlying state machine and execution trace into account. Especially, the execution trace is a precious type of information and can be used to select and filter raw data from sensor stream recordings.

The semantic knowledge inherent to each action extends the knowledge of the robot about its own behavior. RATNs provide semantic information about the modeled action, amongst others the action type and the required resources, which are commonly defined in an ontology. In the beginning of this chapter in Fig. 5.1, I already gave an example ontology for resources. It defines robot, domain and task specific resources. Accordingly, I created an ontology for action types, and wrote a dedicated ontology plugin for loading the ontologies into RAFCON. The plugin ensures that the

user can, during RATN modeling, only enter resources and action types defined in the ontologies (and thus avoids un-modeled entities and typos).

The library and hierarchy concept of RAFCON leads to reuse and inheritance of semantic data. This is based on the fact, that LibraryState, which only have to be annotated once, can be reused (i.e., linked) several times into a robotic behavior. Thus, a developer only has to annotate a fraction of the whole set of states (in the state machine of Section 7.1.1, less than 5% of all states had to be annotated). Furthermore, the annotation of LibraryStates and the programming of the robotic behavior for a certain mission can be done independently by different developers.

In general, both the amount of semantic information per action and the coverage of the state machine annotation are adaptive. Thus, the developer is able to make a conscious decision concerning the tradeoff between manual annotation overhead and the degree of the semantic structure and classification of raw data. Once RAFCON is integrated into a robot, the annotation workload is kept low, as only the domain-specific semantic knowledge has to be modified. This finally leads to the fact that there is no additional annotation overhead for a robot that always works in the same domain.

5.5.1 Design Time Analysis Features

Tests (incl. Simulation): The majority of the employed high level software (including RAFCON) running on the robotic systems such as the LRU or AIMM (see the validation Chpt. 7) is covered by unit tests. We employ a CI workflow based on Buildbot[3] and Jenkins[4]. The CI system automatically builds our software module on each new commit and runs all unit tests. The workflow includes versioning with Git and the usage of a sophisticated package management system (called *rmpm*, see Schuster et al. (2017)), able to resolve complex dependencies and build highly modular runtime environments. For system simulation, we use the RoverSimulationToolkit (see Hellerer et al. (2016) and Schuster et al. (2016)), a multi-body simulator written in Modelica currently developed at DLR, as well as Gazebo. More information about how I employed Gazebo is given in the validation Chpt. 7.

Code Syntax Checks: Pylint[5] is used for RAFCON to inform the developer about syntax errors in the code. Pylint is able to check common coding standards such as

[3]https://www.buildbot.net/
[4]https://www.jenkins.io/
[5]https://www.pylint.org/

Python's PEP8 style guide[6], to find duplicated code and to create UML diagrams for Python code.

Data Integrity Checks: As our task programming framework is written in Python, data integrity is not guaranteed in a way as it is the case for statically typed programming languages such as C++. Thus, a concept for checking the data types of data passed between states is necessary. As RAFCON is designed in a way that data handling is modeled explicitly, all data flows between states are checked for matching types (see Brunner et al. (2016a)).

Error Handling Coverage: To calculate the error handling coverage of a state machine, all states whose "aborted" outcomes (triggered in the case of errors) are connected to another state are regarded as covered. The calculation of this metric is simply performed using the RAFCON API and Python.

Model Checking: We integrated the *DIVINE* model checker, created by Barnat et al. (2013), into our RAFCON framework (see Vilzmann (2016)). Fig. 5.8 shows an overview of the integration of the model checker into RAFCON. DIVINE uses partial order reduction for state space reduction and can distribute model checking tasks to multiple computation nodes. These features drastically decrease the problem of the state-space explosion problem as outlined in Sec. 5.3.1.

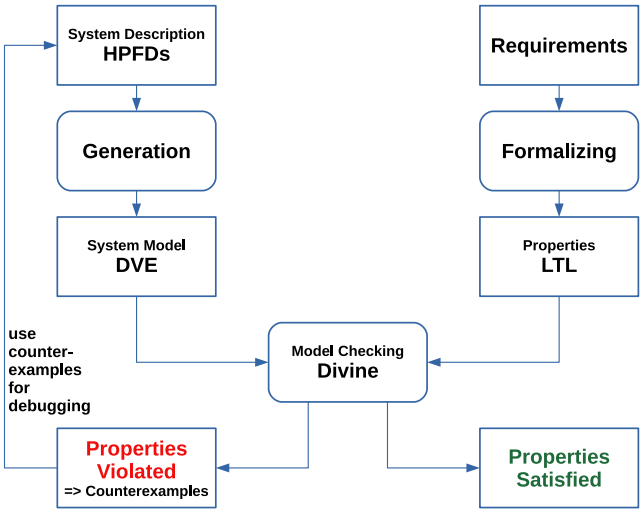


Figure 5.8: *The RAFCON model checking implementation*

Robotic behavior modeled purely with HPFDs cannot be model checked, as the information about the effects of actions practically cannot be inferred by static code analysis automatically. All code including that one of the functional layer would have to be analyzed, which is a huge amount of work considering the million lines

[6]https://www.python.org/dev/peps/pep-0008/

of code of all modules running on, e.g., the LRU robot. Thus, we performed model checking on symbol level by annotating the HPFDs states using the RATN concepts. The annotation does not have to be performed for all HPFD states, but only for those related to the system properties of interest. Our converter automatically generates a DVE transition system out of an annotated HPFD. DVE is the native modeling language for DIVINE[7] and was created by Barnat et al. (2013) for modeling asynchronous systems and protocols. The following RATN properties are mapped onto their DVE specific counterpart:

- states: map to DVE processes
- transitions: map to DVE transitions
- outcomes: map to DVE synchronization channel variables
- world model and resource effects: map to DVE variables by using transition effects
- world model and resource preconditions: map to DVE guards (which access the DVE variables in turn)
- data flows: are not mapped, as they affect the internal behavior of ExecutionStates (i.e., Python code), which is not verified by our current DIVINE pipeline

The execution engine of RAFCON was mapped to a simple interaction model based on DVE channels, states and transitions, which was able to start and preempt DVE states based on the HPFD execution semantics.

Listing 5.1: *Example LTL property*

```
#define motor-running (motor_running == 1)
#define flying (is_flying == 1)
#property [] (flying -> motor-running)
```

Given a transition system in DVE, DIVINE can validate arbitrary system properties, which can be defined via *Linear Temporal Logic (LTL)* (see Barnat et al. (2013)). One example of a LTL formula is given in Fig. 5.1. Here “motor-running” and “flying” are two atomic propositions of a system model (the according system model is given in the work by Vilzmann (2016)). The LTL property states that for the whole system runtime the propositional formula “(flying -> motor-running)” must always

[7] see `https://paradise.fi.muni.cz/~xstill/darcs/divine31a/gui/help/divine/language.html` for the DVE reference.

be satisfied. The formula reads as: "The *flying* property must always imply the *motor-running* property". Thus, if *flying* holds, than *motor-running* must also hold.

In the case that the model checker has found a property violation it automatically generates a counter example, which can be used to debug the HPFD. This process is iteratively performed until no counter examples are found anymore.

5.5.2 Runtime Analysis Features

Logging Outputs: For logging state machine execution relevant data Python's logging library is used. It can be easily configured to log data either to the standard output, to a specified file, or both. Furthermore, various logging levels can be configured, which can be used to highlight the severity level of logs. Moreover, each module has its own logger. Next to the time, the module's name is prepended to each logging statement, which allows to easily track the origin of the statement.

Graphical Viewer: The implemented viewer shows the current values of all data ports directly next to the states. This live data introspection is especially valuable to, e.g., quickly check the index of a loop or the id of a target object (e.g., see the "seismometer_id" data port of Fig. 5.7). For more complex data values the data is cut. Thus, proper plugins must be written to visualize the more complex data in a clear manner.

Execution History View: RAFCON completely tracks the execution history of all state machine runs. A foldable *Execution History Tree View* shows all executed states in their respective hierarchies together with all the context data before and after a state execution (see widget "6" in Fig. 4.31). The execution history is also completely accessible after runtime, but in a lightweight GUI optimized for large data sets.

An overview of the context data, which is logged per state, was already explained in Fig. 5.7. One central element of the context data is the input and output data per state, i.e., the parameterization of an action. The Python implementation stores the context data by marshaling the complete input and output objects. Compared to just logging the string format of objects to some log file, complete marshaling has several advantages.

First and foremost it allows for fully recovering the objects in the form they had during execution time. This assures that all information is present, not only the information used to create the string representation of the object. Next, the objects can be used for further calculation or can be transformed using their builtin class

methods. Finally, as we rely on the pickle library of Python, data pickled with an old EOL Python versions (e.g., Python 2.7 or Python 3.4) can be analyzed using newer Python versions (e.g., Python 3.7).

5.5.3 Post-Mortem Analysis Features

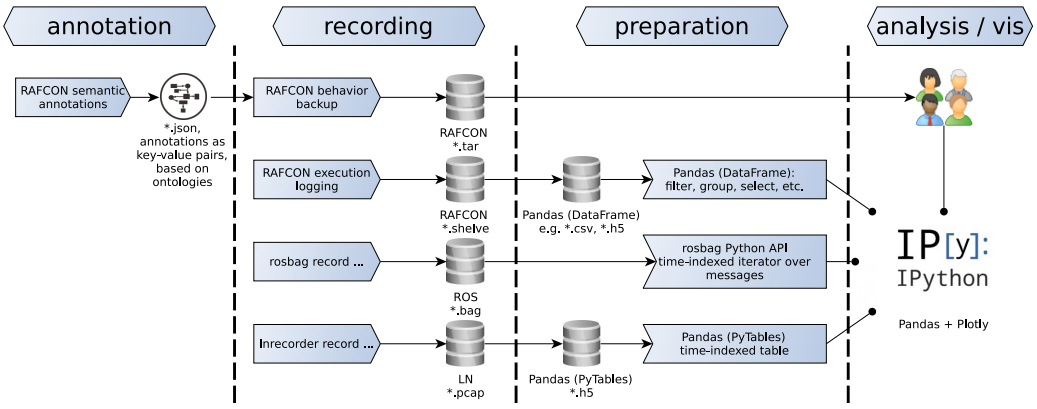


Figure 5.9: *The pipeline for generating data for post-mortem analysis (see Riedel (2018)).*

Task Metrics and Statistics: The pipeline of how the data for post-mortem analysis is generated and analyzed is shown in Fig. 5.9. In the annotation phase all skills of the target HPFD have to be semantically annotated (i.e., by using the RATNs methodology). In the recording phase, the RAFCON state machine, the RAFCON execution history, and selected middleware raw data are recorded. In the preparation step, the data is edited such that it can be processed more efficiently. The **.h5* file ending refers to the *HDF5* file format[8], which has the advantage of fast data access times (much faster than SQL data bases, see Folino et al. (2009)), and thus providing high data throughput for big data applications. Another advantage of HDF5 is its ability to work well for time series data, which is important for this use case as all data is queried on a time-index basis. Python's Panda library (see Harrison and Prentiss (2016)) is then used to cache the data loaded from *.h5 files. Finally the Panda library and Python's *Plotly* library (see Plotly (2017)) are used for the calculation and visualization of task metrics and statistics.

Concept-wise my analysis pipeline is similar to *Narrative-Enabled Episodic Memories* (NEEMs) as defined by Beetz et al. (2018). They are used to store robot experience

[8] https://www.hdfgroup.org/solutions/hdf5/

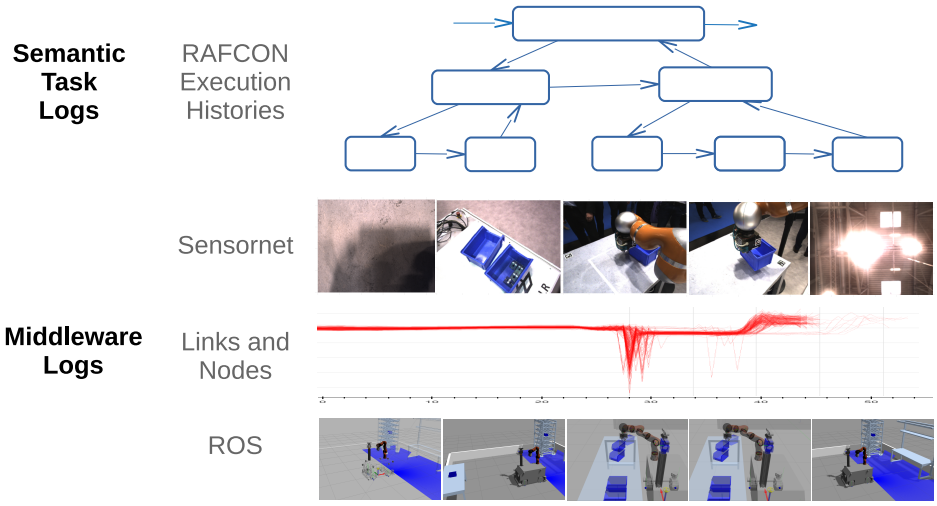


Figure 5.10: *A schematic overview of the building blocks of NEEMs. The upper part shows a task tree, with different rows mapping to different semantic task levels. The lower part shows the data of different middlewares with the x-axis showing the time.*

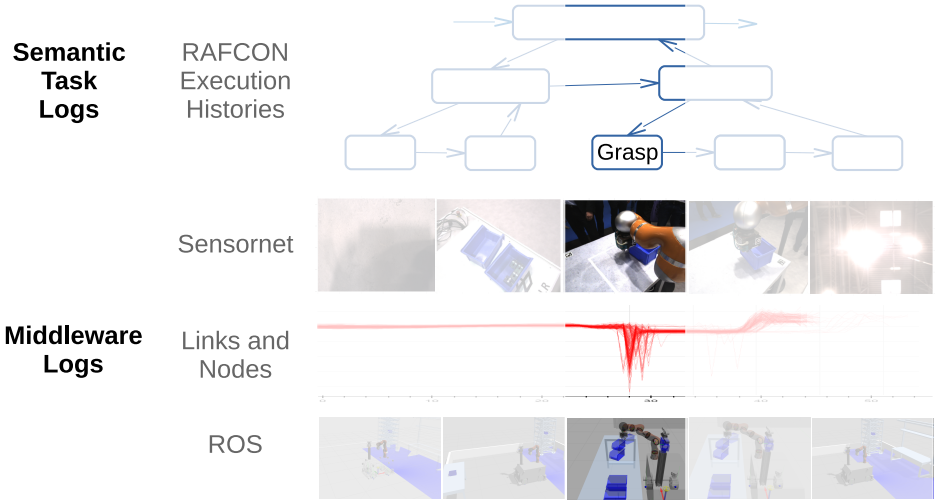


Figure 5.11: *A schematic overview, how annotated RAFCON states can be used to query (based on timestamps) robotic experience data distributed to data streams of several middlewares.*

linked with semantic information about the task, the robot and its environment. Fig. 5.10 and 5.11 show a schematic overview of how the RAFCON execution histories can be used to query raw data related to specific actions (using the AIMM example of the validation chapter). Using API functions provided by RAFCON, the start and end timestamp of specific actions can be queried. These timestamps can then be used to query into the time-indexed raw data streams of the robot. The raw data includes images, poses, force profiles (e.g., of in-contact motions) and scene descriptions (see bottom data stream) based on the robot's belief state.

This pipeline enables the automatic generation of labeled raw data. This is a promising approach to automatize the creation of machine learning models such as neuronal networks. In fact, during Automatica 2018 (see validation Chpt. 7) we auto-generated a neuronal network for in-hand detection of grasped objects using the labeled raw images.

Gantt Charts and RCPs: As the runtimes for autonomous tasks can be high (the validation Chpt. 7 shows experiments with a duration of several hours, executed several days in a row) and the number of action types and resources can be numerous, scalability problems quickly occur. Thus, I employ the plotting library Plotly[9] to generate various plots. Since the user can zoom into arbitrary sections of the graphs, translate the view to interesting spots and zoom out again to gain an overview, the presented plots scale extremely well. Moreover, arbitrary action types and resources can be dis- and enabled with a simple click into the legend. Ultimately, arbitrary information can be rendered at the mouse position while the mouse is hovering over a certain part of the chart. This is used to show the context data of the selected state. Multiple state machine executions over several sessions can be grouped and analyzed together.[10] As future work the time-synchronized visualization of multi-robot experience data of several runs could be performed.

5.6 Application Scenarios

In the validation Chpt. 7, I present the results of applying the whole-lifecycle analysis architecture in two application scenarios: The first is taken from the space domain, i.e., the final ROBEX demonstration missions. The second shows the industrial use case of maintaining an automated supply chain with the AIMM platform.

5.7 Related Work and Discussion

In this section I will highlight and compare relevant task control frameworks using their possibility to analyze robotic behavior during all three lifetime phases. Specifi-

[9] `https://plot.ly/`

[10] See `http://rmc.dlr.de/rm/de/staff/sebastian.brunner/task-analysis` showing example plots created with Plotly.

cally, this refers to behavior analysis during design-time, during runtime and after execution (i.e., the post-mortem phase). The first phase focuses on possibilities to inform the behavior developer about the quality and robustness of the programmed tasks before executing those tasks. The runtime analysis features enable the mission control to monitor the execution of the tasks, to get deep insights into the current state of the robot and to get feedback of the modules of the functional layer. The last type of the analysis, the post-mortem analysis, helps to investigate special events occurred during the execution, error reasons and performance bottlenecks using semantic task logs.

Table 5.1 shows an overview of various analysis features for a selected number of frameworks. The life-cycle analysis of robotic tasks strongly depends on the data collected by the underlying task control framework. This is probably the reason why all theoretical work concerning this topic I found in literature was published alongside such frameworks. The table does not contain as many entries as Table 3.7 as not all frameworks provide analysis features of the proposed kind.

Table 5.1: *Analysis features supported by different task programming software.*

Task Control Framework	Data Integrity	Error Handling Coverage	Graphical Live Data	Execution History Online	Logging Outputs	Semantic Logging	Execution History Offline	Task Replay in Simulation	Gantt Charts	RCPs	Task Statistics
Smach			x		x						
ROS Commander			x		x						
Xabsl	x		x		x						
FlexBE	x		x	x	x		x				x
CRAM + openEASE					x	x	x	x	x		x
RAFCON	x	x	x	x	x	x	x		x	x	x

For example, *CRAM* (see Beetz et al. (2010)) in combination with *openEASE* (see Beetz et al. (2015)) offers powerful semantic task logging features that enable to replay the high level actions executed by a robot. The semantic knowledge is, as in my approach, defined in ontologies. openEASE is not only used for visualization but

also as a mean to retrieve and infer knowledge via Prolog. However, this framework does not support as many design time features, runtime analysis possibilities and debugging support related to data handling and data integrity.

ROS Commander by Hai Nguyen et al. (2013), *XABSL* by Loetzsch et al. (2006), *SMACH* by Bohren and Cousins (2010), and *FlexBE* by Schillinger et al. (2016) are also examples for behavior programming frameworks that offer several analysis concepts. Unfortunately, the development of ROS Commander and XABSL discontinued several years ago. Furthermore, they lack important post mortem analysis features. In SMACH, although widely used, the task logging is restricted to recording console output only. FlexBE on the other hand offers quite powerful logging and analysis features but does not include the recording of semantic knowledge based on ontologies.

ROS diagnostics (see Fernandez et al. (2015)) is a very powerful monitoring tool for robotic systems programmed in ROS. Using my terminology it is mostly a tool for runtime task analysis. It can be used to characterize the functional state of a robot and to monitor various states inside ROS topics and make sure that the observed properties are valid or inside a certain numeric range. On top of that, it offers visualization tools to access detailed information quickly. Finally, it can be used for long term logging and historical analysis. However, the real gain for behavior analysis is limited as the logging lacks synchronization to semantic task data. Moreover, monitoring rules are hand-coded and update rates and history length can only be set to fixed values, which does not always account for the dynamic properties of complex robotic systems. Although it is widely used in the robotics community its functionality is too limited from a semantic behavior's perspective and thus not covered in Table 5.1. Interesting work by Kirchner et al. (2013) motivates to integrate Bayesian Networks and Bayesian Inference to dynamically configure the monitoring (i.e update rate and history length) of the systems based on semantic events. Unfortunately, I cannot add this work into the comparison table as no open source software was released for this extension.

As the majority of my use cases are space related, I would like to highlight NASA's effort in this field as well. Logging and analysis software for mission critical spacecraft data, such as logging frameworks with online modifiable severity levels (see Dvorak et al. (2000)) and timeline and resource capability profile plots (see Chien et al. (1999)), always played a central role in their software architectures. Recently, NASA also enhanced their *Europa* framework (see Barreiro et al. (2012)) for integrated planning and scheduling with features to visualize *Gantt* charts, action details, action violations and solver statistics, i.e., statistics of (generated) action sequences. The framework focuses on the creation of plans and does not provide much support

for runtime and post-mortem analysis. Finally, NASA recently created a telemetry visualization GUI called Open MCT (see NASA (2017)). However, it only aims at visualizing logged telemetry data, not the logging itself.

The discussion of the related work shows that, except CRAM, no other framework supports as fine-grained task logging, including semantic data, as RAFCON. Most of the times, none or only some of the features mentioned above are supported. Especially concerning fine grained logging capabilities and error statistics, RAFCON, including its extensions, is more mature than the other analyzed frameworks.

Subsequently, I give a comparison between the profiling features of our task programming framework RAFCON and those of high-level-programming-language profilers.

Table 5.2: *Feature support of different profiling frameworks.*

Profiler	Total Call Count	Total Spent Time	Time per Call	Time per Total Spent Time	Max / Min Time	Total Exception Count	Exception Proportion	List View of Called Entities	Call Graph	Memory Leak Identification	Suitable for Task Profiling
Valgrind + KCachegrind	x	x	x	x	x	x	x	x	x	x	
Python "profiling"	x	x	x	x				x			
cProfile + graphViz	x	x	x	x				x	x		
JProfiler	x	x	x	x	x	x	x	x	x		
RAFCON	x	x	x	x	x	x	x	x	x		x

When designing and implementing the architecture for whole life-cycle analysis, I tried to keep close to the concept of profilers for high-level programming languages. Well known examples in this area are *Valgrind* (see Nethercote and Seward (2007)), *JProfiler* by Gousios and Spinellis (2008), *cProfile* (see Python Software Foundation (2017)) together with *Graphviz* by Ellson et al. (2002), and the *profiling* package of Python (see What! Studio (2017)). Although all of them are more powerful than RAFCON for their respective programming language (and Valgrind being specifically useful for identifying memory leaks), I applied many of their features to robotic task profiling. In Table 5.2, I give a feature list of what common profilers normally offer

and compare different profiling frameworks using these features. The table shows that RAFCON exhibits all necessary profiling capabilities. However, to the best of my knowledge, nobody did profiling on pure task level for robotic applications before.

The experiments in the validation chapter show that next to the actual manipulation actions, (motion) planning and computation actions can consume a huge amount of the overall behavior runtime. In some cases even more than the acting time of the robot, i.e., the time in which the robot moves its base, manipulator or gripper.

Now we have to critically analyze the reasons for this. Robots are very complex mechatronic systems. In order to handle this complexity and let the robot do sensible things, we stick to a sense-plan-act routine: at first, the robot senses its environment, then, it plans its next actions including their parameterizations and, subsequently, it acts in order to change its environment. For all of these three steps (computer vision, planning, manipulation and control) there exist many methodologies and algorithms. However, they all use up time. From an economic perspective, all the time the robot does not manipulate its environment is lost time. A robot being able to act all the time and does not have to pause until some internal calculations are finished is superior to a robot that stands still significant amounts of time.

How can we achieve **more effective robots**? There are basically two possibilities. Either the algorithms for planning and computer vision get better and their runtime can be reduced. Or we directly tackle the sense-plan-act loop that is most often simply executed sequentially. This sequential execution is not needed in many cases, as many actions can either be parallelized or pre-calculated. The next and final theoretical chapter tackles exactly this problem.

Chapter six

Autonomous Optimization of Robotic Behavior

Programming robots that require prolonged phases of autonomy to solve their tasks is still very challenging. Robot task programming frameworks facilitate this process (as outlined in Chpt. 3). However, in practice, the execution of such programs is often slow due to high software run-times. I identified two reasons for this.

First, path planning and object pose estimation are usually computationally intensive operations, and only by applying probabilistic or learning-based algorithms sound solutions can be found at all. This holds true in particular for mobile manipulation, which requires many calls to such expensive operations for perception, navigation, and planning.

Second, actions are often programmed only sequentially, and opportunities for parallelization and pre-computation are not exploited. Users often program *a* task solution that works in principle, but in the space of *all* possible task solutions, is hardly ever the most efficient one.
See for instance the mobile manipulation use case in Fig. 6.1, where the task is to deploy several scientific instruments, i.e., boxes with different measurement devices such as seismometers. For each box, the executing rover has to drive to a suitable location, has to register to the environment and has to detect the box in its carrier and the target location where the box has to be placed. After picking the box from its carrier the robot places it at the target location.

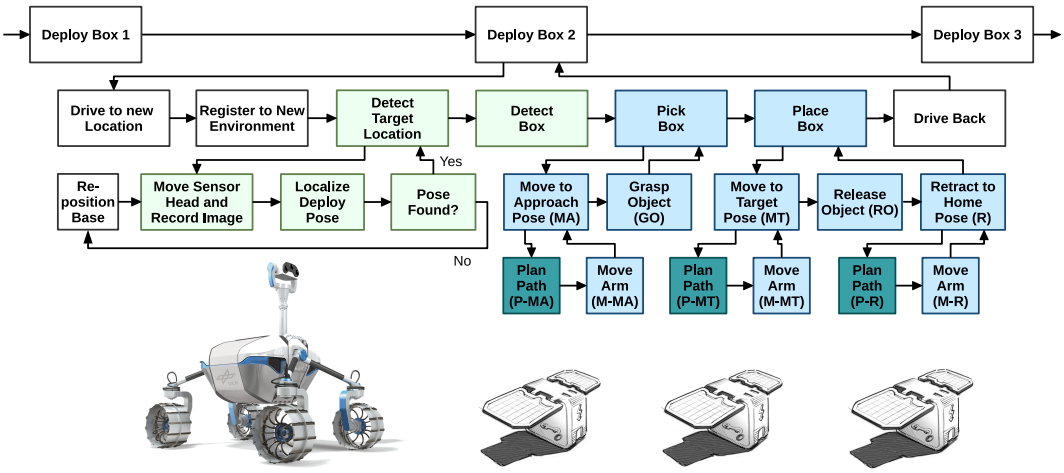


Figure 6.1: *An exemplary HPFD representation for a mobile manipulation task, in which a robot has to deploy several payload boxes. The rows represent the different abstraction layers of the tasks. Vertical lines represent* consist-of *relations, horizontal lines* next-state *relations. The* Detect Box *state, for example, consists of the states* Move Sensor Head and Record Image (MaR1), Localize Object (LO1), *and* Object Found (OF).

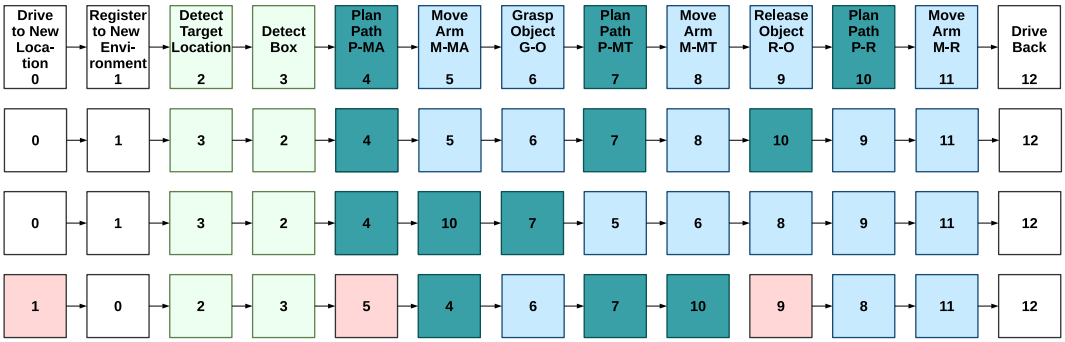


Figure 6.2: *This figure shows four execution example traces for the mobile manipulation task given in Fig. 6.1. Horizontal lines show "next action" relationships. Invalid actions are rendered in red.*

As Fig. 6.2 visualizes, there are several possibilities concerning the execution order of the tasks. For example, detecting the target location could be performed after detecting the box (see row 2), and thus also in parallel. Moreover, the planning actions could be executed before executing any of the manipulation skills. This shows that the initial order of the actions, as specified in the state machine, is over-constrained. A second aspect of the example worth highlighting is that all actions are required to solve the task, but not all of them affect the environment state (i.e., the state of the real world). For instance, computing a path with a path planner does not affect the objects in the environment, but this computation is necessary before the arm can be moved to follow this path. If this computation could now be pre-computed, precious execution time could be saved.

This leads to the following scientific questions: How can opportunities for parallelization and pre-computation in such over-constrained task descriptions be exploited? How can we leverage the knowledge and respect the constraints encoded in programmed task descriptions, whilst still automatically determining opportunities for parallel execution, pre-computation and caching of results?

6.1 Concurrent Dataflow Task Networks (CDTNs)

In this chapter I will propose a conversion of HPFDs to so called *CDTNs*, whose execution can be optimized autonomously. For humans, highly concurrent, interleaved implementation of tasks is not easy to understand, modify and maintain. I argue that the optimization of programmed tasks should thus be automated, rather than leaving it as a tedious task for the user.

6.1.1 Resource-Aware Task Nodes Revisited

In this chapter I will employ the dependency model of RATNs in order to represent partial ordered set of actions that can be subject to autonomous execution optimization. RATNs have already been introduced in Chpt. 4. As they play a central role for the presented optimization methodology, they are illustrated here in more detail and with additional examples. Additionally, a formalization of RATNs will be given, which is needed for explaining subsequent concepts, e.g., the CDTN execution algorithm.

6.1.2 Dependencies

RATNs feature three types of dependencies: *world model dependencies*, *data dependencies* and *resource dependencies*.

World Model Dependencies World model dependencies represent action preconditions, whereby actions may only be executed if certain facts in the world state hold, as in PDDL (see Mcdermott et al. (1998)).[1] Formally, world model de-

[1] I use "world model" as synonym for "world state". Both describe the robot's model of the real world it acts it. To refer to the state of the real world I use "environment state".

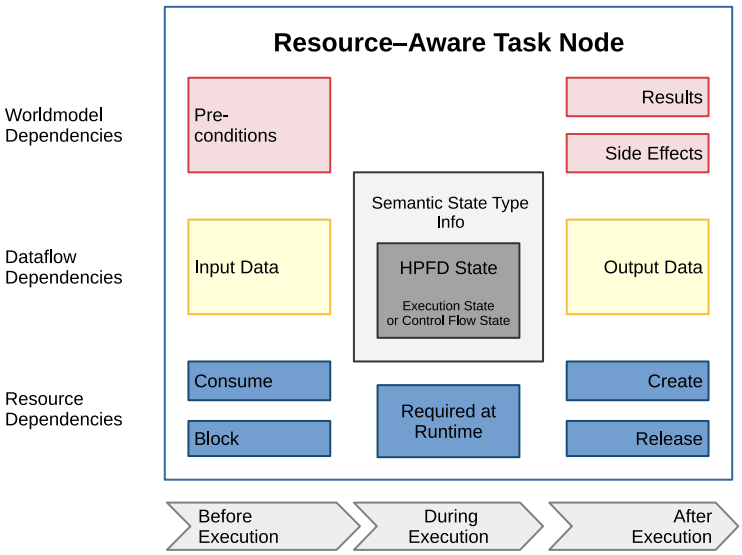


Figure 6.3: *Overview of the components of RATNs.*

pendencies are defined as parameterized predicates, which can be grounded in ontologies. For instance, the action $grasp\_object(o)$ can only be called if the fact $manipulator\_at\_approach\_position\_of\_object(o)$ holds. As it can be seen in this example, world model facts can be bound to specific objects. Action effects, on the other hand, add, modify or delete facts of the world model and thus satisfy the preconditions of other actions. For implementation details see Fig. 6.10 in the implementation section.

Data Dependencies In many robotic task control languages, data, signals and events are modeled in a global variable structure (such as a global world model, see Beetz (2000)). In contrast, dataflow modeling defines clear scopes in which data is accessible and valid. Furthermore, it ensures that data interfaces between processing nodes are consistent. On top of that, the concept strongly facilitates parallelization possibilities. In terms of debugging, this leads to a clear data dependency overview for the whole system. Another advantage is the ability to modularize and reuse functionality (see Wulf and Shaw (1973)).

A data dependency represents a necessary datum for an internal computation, but it does not influence the world model of the robot. Thus, data dependencies are modeled as data flows of HPFDs. They are formally defined as a quadruple: *(source_action, source_data_port, target_action, target_data_port)*. Additionally, they are associated with a defined data type.

The main reason for employing data dependencies in CDTNs is to enable easy projection of action effects and finally the pre-calculation of intermediate task data.

Resource Dependencies Planning for robots must take into account the resources that are available to a robot: Is there an idle motion planner for computing a motion plan? Is the planning process running, or has it crashed upon startup? Does the robot have enough battery power? Is a world model currently available (e.g., for forward simulation), and is it up-to-date? Such questions are often not taken into account in high-level planning and scheduling algorithms, but are essential in order to achieve robust autonomy.

The resource manager, presented in detail in Section 6.2.1, is a component which administrates all resources in a resource pool. As a concrete example, if the action $plan - path$ depends on the path planner process to be running but it has crashed, the resource manager ideally signals the process manager to create this resource by re-starting the process.

In my resource model, there are five classes of resources: *system, process, module, world* and *task* resources. System resources are created by hardware interfaces (e.g., by monitoring the battery power), process resources by a *process manager* (e.g., if the object detection process is running), module resources by the modules themselves (e.g., stating that the internal module state of the navigation module is 'unlocalized'), world model resources by the world model pool (i.e., copies of the root world model) and finally task resources by the task control module (e.g., two blue boxes are already delivered). Task resources help to track the progress of the final goal, e.g., if several blue boxes have to be delivered.

Thus, hardware and software resources can be represented in the same model. Almost any component can be modeled as a resource, independent of if it is a software process (e.g., a constraint geometric motion planer), a semantic resource (e.g., two objects left to deliver) or the whole robot. The latter point also makes the model attractive for multi-robot scenarios.

Formally, I define resource dependencies, such as world model dependencies, as parameterized predicates grounded in an ontology (see Brunner et al. (2018)). In practice, resource dependencies are modeled in RAFCON's Semantic Data Editor, shown in Fig. 6.10.

6.1.3 Dependency Graphs

A CDTN is created by converting the actions of a HPFD into RATNs and by defining dependencies between those task nodes. During CDTN execution, a dependency graph (see Fig. 6.4) is generated, in which the nodes are task nodes, and the edges

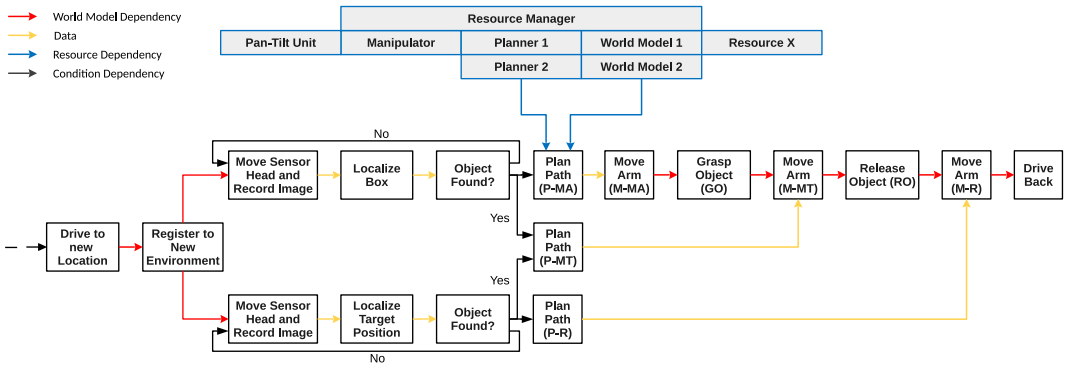


Figure 6.4: *Exemplary dependency graph for the mobile manipulation task shown in Fig. 6.1. Different colors model different kinds of dependencies between the actions. Since there are many resource dependencies, only two exemplary dependencies have been plotted. Condition dependencies are explained in Section 6.3.2.*

are task node dependencies. World model dependencies (red) are drawn between two actions a and b if a makes a world-model fact become True, which is required by b. For example, grasping an object (GO) is only possible if the manipulator is at the approach pose for the specific object, which it achieved by executing action M-MA. Resource dependencies (blue) connect a resource to an action that needs the resource. For example, to plan a collision free path for the manipulator the path planner action P-MA needs a planner and a world-model resource. Finally, data dependencies (yellow) connect actions that compute a certain data structure to actions that require this data as an input. For example, the action for moving the arm M-MT requires a trajectory as a data input.

Generating this graph happens in two steps. First, *Iterator* nodes of the CDTN are expanded (see Sec. 6.3.1). Then, all dependencies of the original and newly created task nodes are inserted as edges into the dependency tree.

Note that several actions are no longer constrained to be executed sequentially (e.g., the path planning actions P-MA, P-MT, P-R), as was the case in the HPFD in Fig. 6.1. Therefore, they can be executed in parallel even on different CPUs. In Sec. 6.3.2, I present an algorithm that is able to **exploit** these **potential parallelizations**, without violating any of the dependencies.

6.1.4 Formalization

In the following, a formalization of RATNs is given. The formalization allows for an exact understanding and is a reference guide for subsequent sections.

Definition 14 (World Model) *The world model consists of a set of facts f_i as defined in the HPFD formalization. Every fact has a knowledge identifier ki_f. The term $world\_model(ki_f)$ retrieves the (truth) value of a fact f represented by the knowledge identifier ki_f.*

Additionally, a world model must have the following properties: "persistent", "versionable", "reproducible" and "knowledge-inferable". The property "persistent" refers to the fact that a world model is not volatile and still present if the robot reboots. "Versionable" means that for every point in time a dump of the world model can be made in a versionable (i.e., non binary) format. "Reproducible" refers to the ability to create a copy of a world model, which behaves identical to the original world model. The property that "knowledge is inferable" refers to the ability of knowledge bases to infer new facts from existing facts and relations, e.g., via a query languages or first-order-logic.

Definition 15 (Resource) *A resource r is an entity in a robotic system with a certain resource type rt: $rt(r) \in \{system, process, module, world, task\}$. The following operations are defined for resources: $\{create, consume, block, release\}$.*

For more information about the operations on resources see Sec. 6.2.1.

Definition 16 (Resource-Aware Task Node) *A task node tn is a HPFD state with an additional set of information. It has an action type (written $action\_type(tn)$) and an optional control flow type (written $control\_flow\_type(tn)$), whereas the action-type is based on an action type ontology and the control flow type is one of the following values: Condition, Loop or Routine.*
Each task node has preconditions, results, side effects (which map to world model dependencies) and can perform resource operations (which map to resource dependencies). Data dependencies refer to the data ports of the underlying HPFD state. The set of all dependencies of tn is written as $dependencies(tn)$ (see Def. 19). Results, side-effects and the resource operations can be bound to the outcomes of the underlying HPFD state, i.e., there can be different effects for different outcomes. Such effects are called conditional effects.
A task node tn may not only consist of one HPFD state but also of several HPFD states and other task nodes, which are called child nodes $child\_nodes(tn)$.

The overview of RATNs is shown in Fig. 6.3. The use of *control flow types* is explained in the context of the CDTN execution algorithm (Sec. 6.3.2).

Definition 17 (Action) *All HPFD states and task nodes are called actions. All actions of a task node tn are defined as $actions(tn)$.*

Definition 18 (Parental) *For actions a, b the term $parental(a, b)$ is $True$ if b is an element of the set $parental\_nodes(a)$. The set $parental\_nodes(a)$ is the transitive hull of the parent relation defined in Sec. 4.1.2.*

Definition 19 (Task Node Dependency) *A task node dependency d is a dependency between a source task node tn_s and a target task node tn_t. Every dependency d has a truth value, $truth\_val(d) \in \{True, False\}$. Per default $truth\_val(d)$ is $False$. There exist three types of dependencies.*

- *A data-dependency dd always has a data flow df associated with it (including a data flow source and target port, see the HPFD formalism in Sec. 4.1.2). All data-dependencies have a dependency source ds and a dependency-target dt. ds maps to tn_s and dt to tn_t. If, by executing tn_s, a value is assigned to the data flow source port of df then $truth\_val(dd)$ is $True$.*
- *A resource-dependency rd always has a resource r associated with it. If some tn_s is able to create (or unblock) resource r and tn_t requires r (by either blocking or consuming r), then rd belongs to the dependencies of tn_t. If tn_t could successfully block or consume r then: $truth\_val(rd) = True$. If r is created or unblocked by tn_s based on a conditional resource operation the dependency is also called conditional-resource-dependency.*
- *A world-model-dependency wd depends on a world model fact f with knowledge identifier ki_f. If some tn_s creates f and tn_t requires f as precondition, then wd belongs to the dependencies of tn_t. It holds: $truth\_value(wd) = True \implies world\_model(ki_f) = True$. If f is created by tn_s based on a conditional world model effect the dependency is also called conditional-world-model-dependency.*

Definition 20 (Potentially Irreversible Action) *A Potentially Irreversible Action (PIA) is a task node with a special action type at. This type at must have the property $potentially-irreversible$. Potentially irreversible actions are actions that can potentially not be undone.*

In the field of robotics this is the case for the action types *Manipulation*. This means that every action of type *Manipulation* is potentially irreversible, e.g., if a robot manipulates a delicate object and smashes it, the action cannot be undone.

Definition 21 (Dependency Graph) *A dependency graph is a graph, in which the nodes are task nodes, resources, facts and input or output data and the directed edges are task node dependencies.*

Example dependency graphs are shown in the implementation section of this chapter.

Definition 22 (Executability) *An action a is called $executable$ if:*
$\forall d_i \in dependencies(a) : truth\_val(d_i) = True$. *For an action a the term $executed(a)$ is defined as $True$ if all its child actions are executed (see Def. 12). Otherwise, $executed(a)$ is $False$.*

Definition 23 (Dependency) *An action a_x is called dependent on another action a_y if the execution of a_y leads to a direct effect that the truth value of a dependency $d \in dependencies(a_x)$ becomes $True$. Effects can either write a value into the data dependency source of a_x, create or unblock a resource a_x requires or update the world model with new facts a_x has a precondition on.*

Definition 24 (Connectivity) *Two actions a_x and a_y are called connected if:*
$\exists a_z : (dependent(a_x, a_z) \wedge dependent(a_z, a_y) \vee (dependent(a_x, a_z) \wedge connected(a_z, a_y)) \vee (connected(a_x, a_z) \wedge dependent(a_z, a_y))$. *Thus, the connectivity property is transitive.*

Definition 25 (Intermediate Action) *All actions that are a part of the path between two connected actions a_x and a_y in the dependency graph dg are called intermediate states of a_x and a_y. This is written as $intermediate\_state(a_x, a_y)$. The $intermediate\_state$ operator is not commutative, e.g., $intermediate\_state(a_x, a_y) \neq intermediate\_state(a_y, a_x)$ as dg is a directed graph.*

Having defined all prerequisites, CDTNs can finally be formalized.

Definition 26 (Concurrent Data Driven Task Nodes) *Concurrent Data Driven Task Nodes (CDTN) are sets of task nodes including their world model, data and resource dependencies. All task nodes of a CDTN $cdtn$ can be retrieved by $task\_nodes(cdtn)$, all actions (i.e., all the task nodes and the HPFD states they consist of) can be retrieved by $actions(cdtn)$. A CDTN is executed if all its actions are executed.*

Based on these formalisms and the semantic information present in RATNs, CDTNs can now be used to replace the rigid control flow of HPFDs with a more optimal execution strategy. Before describing these concepts in Sec. 6.3 the remaining prerequisites are explained.

6.2 Prerequisites of Autonomous Task Parallelization

Next to RATNs the autonomous optimization is based on three concepts, which are the *Resource Manager*, *Process Pools*, and *World Model Projection*.

6.2.1 Resource Manager

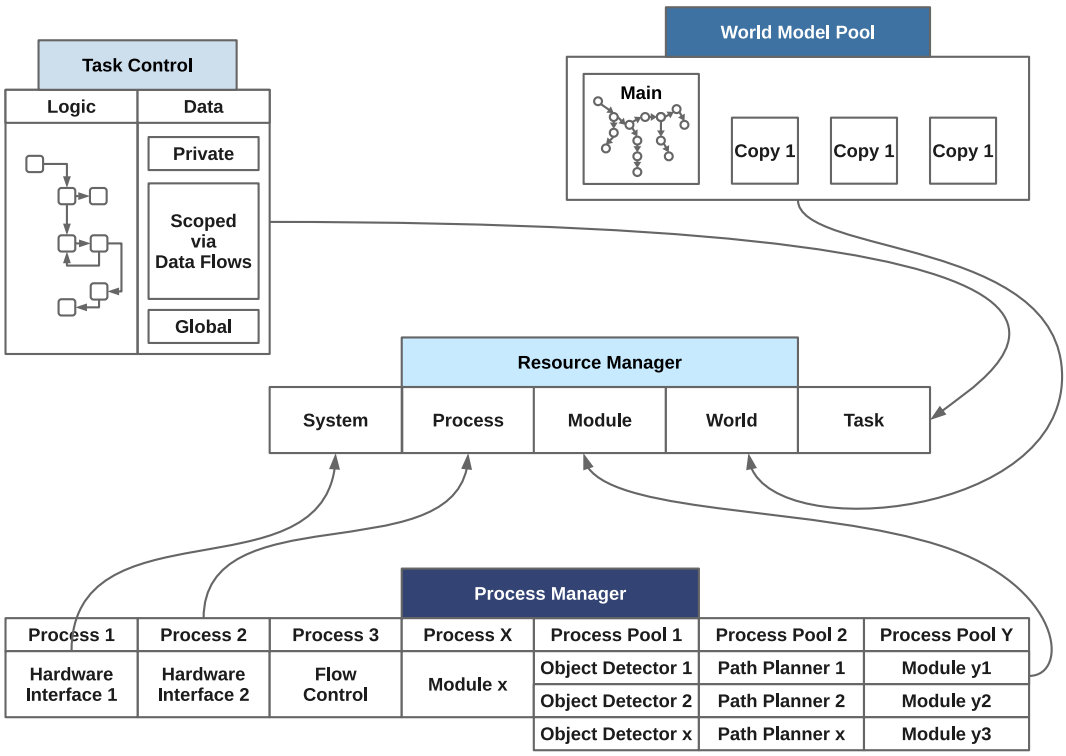


Figure 6.5: *Overview of different resource classes and their relationships: The 'Resource Manager' in the center holds all resources, which are created and forwarded to the 'Resource Manager' by the 'Process Manager', the modules administrated by the 'Process Manager', the 'World Model Pool' and the 'Task Control' module.*

The resource manager holds all resources of the robotic system and manages which of them are available. It cares about concurrent resource requests and resource assignments to task nodes. The following operations are defined on resources:

- create: adds a new resource to the resource pool
- block: blocks a resource for a certain amount of time; if a resource is blocked no other task node has access to the resource until it is unblocked

- release: unblocks a resource
- consume: consumes a resource (can only be applied to consumable resources, such as power)
- exists: asks for the existence of a resource

The resource manager ensures that only those actions are executed in parallel that do not interfere with each other in their resource claims. On the one hand, this is done by thread safe access to all resources. On the other hand, the resource manager only allows actions to either block all of their resources that they need for execution or None of them. Circular waits by partially blocked resource sets are thus avoided.

6.2.2 Process Pools

Process Pools are used to manage several software processes of the same type. They consist of a set of identical processes that only differ by their configuration and allocated task, and are maintained by the process manager. To get a new process of a certain type, the process manager can be asked to create a new process by providing the specification for the process. The manager adds the newly created process to the pool and returns a handle to the process. In Fig. 6.5 an example of two process pools is shown: one pool for path planning processes and one pool for object detector processes. Being able to spawn a configurable amount of processes of a certain type is a key aspect of executing a CDTN.

6.2.3 World Model Projection

During CDTN execution, a pool of world models is maintained. One world model represents the robot's current belief about the actual state of the robot's environment, and is thus called the robot's *root* world model. The other world models are copies, which are used to project the effects of task nodes into the future. They represent possible future world states arising from the execution of task nodes modifying their passed world model instance. Such a task node tn is called *world model task node* and it holds: $action\_type(tn) = World\ Model$.

Similar to PDDL, the world model is a set of facts (see Def. 14). As shown in Fig. 6.5, world models are also resources and are managed in the *world model pool*. The initial

world model is classified as the root world model. The root world model is always the model of the robot's real environment, that resembles it as close as possible.

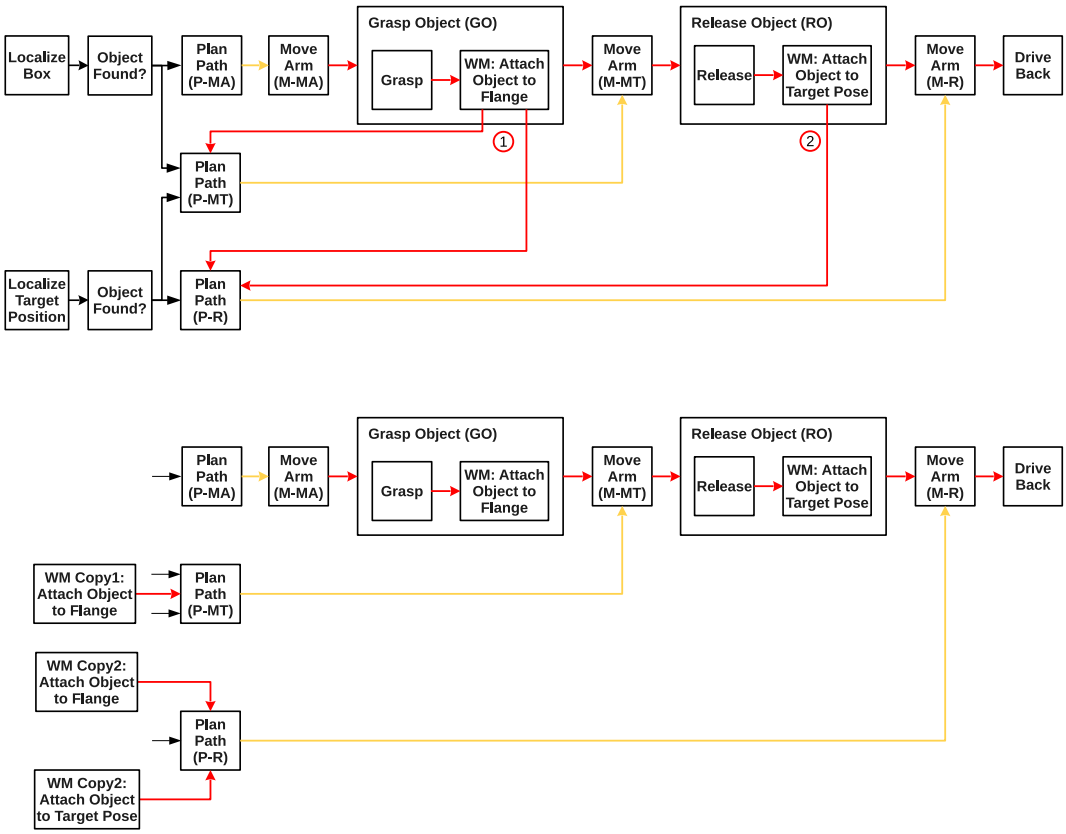


Figure 6.6: *World model projection in the context of the action set shown in Fig. 6.4. The top sub-figure shows an enhanced version of the dependency graph of Fig. 6.4. The bottom sub-figure shows the resolved dependencies for the planning actions Plan Path (P-MT) and Plan Path (P-R) after world model projection. In the bottom sub-figure, the object detection actions are dropped for layout purposes.*

Many task nodes require a world model as a resource, e.g., a path planning node needs the geometric representation of the world state to plan a collision free path to some goal pose. During algorithm execution several world models can be created, modified and deleted. The root world model itself must not be deleted or used and modified for any projection actions. If a task node t requires a world model, a copy of the root world model is created and all *intermediate* world model task nodes between the latest executed PIA and t are executed on the world model copy. As given in Def. 25, these intermediate nodes represent all nodes, which are on the path from t to the last executed PIA inside the dependency graph. This implies that all future events that would affect the world state before t is executed are applied to the world model copy. The modified world model copy is then passed to t as a parameter.

Fig. 6.6 shows the world model projection for the example given in Fig. 6.4. The top

of the figure shows an enhanced version of Fig. 6.4, as it shows some child states of *Graph Object (GO)* and *Release Object (RO)*. As can be seen, the planning actions *Plan Path (P-MT)* and *Plan Path (P-R)* need now more dependencies to be satisfied, which currently renders parallel execution impossible. However, using world model projection the required world model actions are now executed on world model copies in order to satisfy the path planning action constraints. The world model dependencies from the original world model actions (see red labels *1* and *2*) can be neglected as the planning states (*Plan Path (P-MT)* and *Plan Path (P-R)*) now receive their own world model copies on which the required actions have already been executed. The data dependencies of the world model actions must, of course, not be neglected, as otherwise the numeric values for, e.g., the target position would be wrong. In our case, the world model actions get their data from the object localizers, which have to be executed before the planning states can be started.

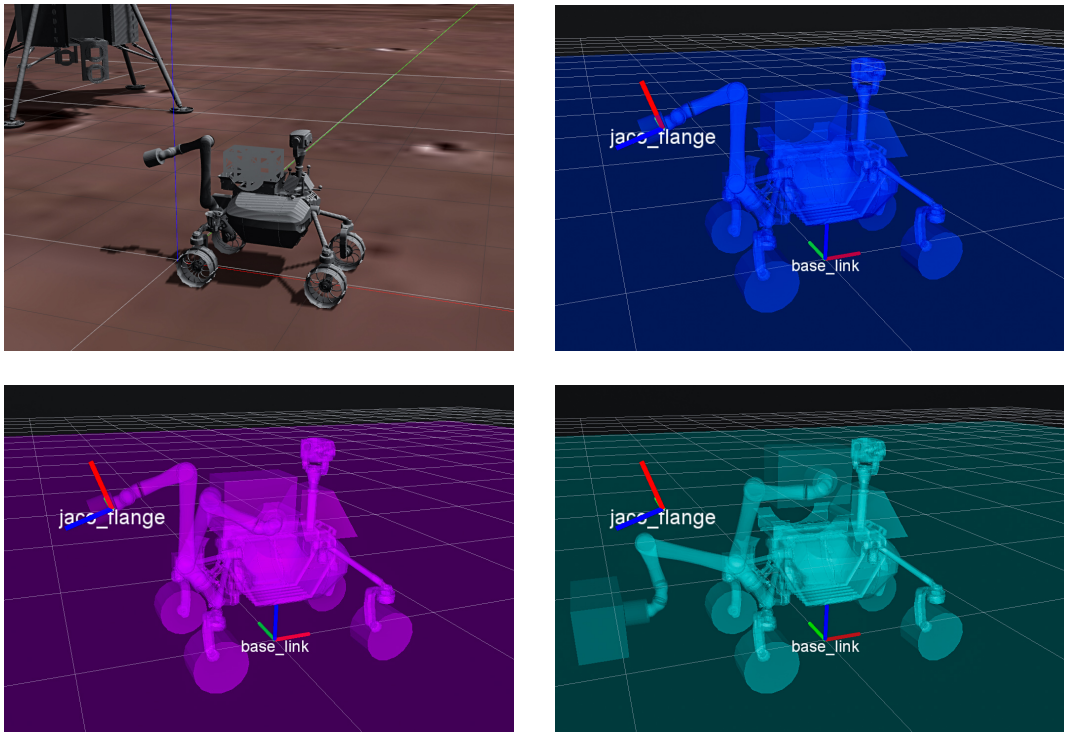


Figure 6.7: *World model projection example using the simulated LRU. Top left: the current real state of the LRU; Top right: the root world model of the robot; Bottom left: projected world model after planning the first path in order to grasp the payload box (action "P-MA"); bottom right: projected world model after planning the path to place the payload box on the ground (action "P-MT"). For the sub-figures with the projected world models both the world model state before and after the planning step is shown.*

Fig. 6.7 shows the projected world model for some of the actions of the behavior example of Fig. 6.6. Next to the physical (simulated) robot environment and the

root world model, the projected world model states (in purple and teal) for the actions “P-MA” and “P-MT” are shown. For both actions, the start and target pose of the planned trajectory are shown. Additionally, each world model shows two transformations, which reflect the current base and end-effector pose of the robot (i.e., the robot manipulator is still in the transport position and did not start to manipulate the payload box yet; however, it already plans all required motions for the future actions ahead).

6.3 CDTN Execution Semantics

As shown at the beginning of this section, graphically programmed HPFDs, as illustrated in Fig. 6.1, are often over-constrained, and do not exploit opportunities for parallelization. CDTNs provide different execution semantics able to optimize behavior execution based on the semantic information of RATNs.

6.3.1 Converting HPFD Control Flow to CDTN Execution Semantics

Converting the HPFD control flow into CDTN execution semantics happens in two steps. At first, all the HPFD control flow elements are dropped (except for *Routines*, see Fig. 6.8). This leads to the fact that the majority of all task nodes will be executed in parallel and only the data flows between the task nodes maintain some notion of action dependencies (such as in dataflow programming, see J. B. Dennis (1980)). Of course, this will violate the logical control flow of the HPFD. As the control flow given by HPFDs is often over-constrained some of the violations are uncritical. However, other violations are critical, i.e., when essential world model and resource dependencies are ignored. Thus, in the second step, a general executability rule is added: only task nodes may be executed which have all their dependencies satisfied. This will leverage the semantic task node information in order to optimize the behavior control flow, without violating the essential task constraints.

Fig. 6.8 shows a summary of all HPFD control flow elements and their conversion to CDTN execution semantics. They are explained in detail in the following.

Sequences and Concurrencies All HPFD states in a sequence are simply mapped to a set of independent RATNs. This means that they can, in principle, be executed

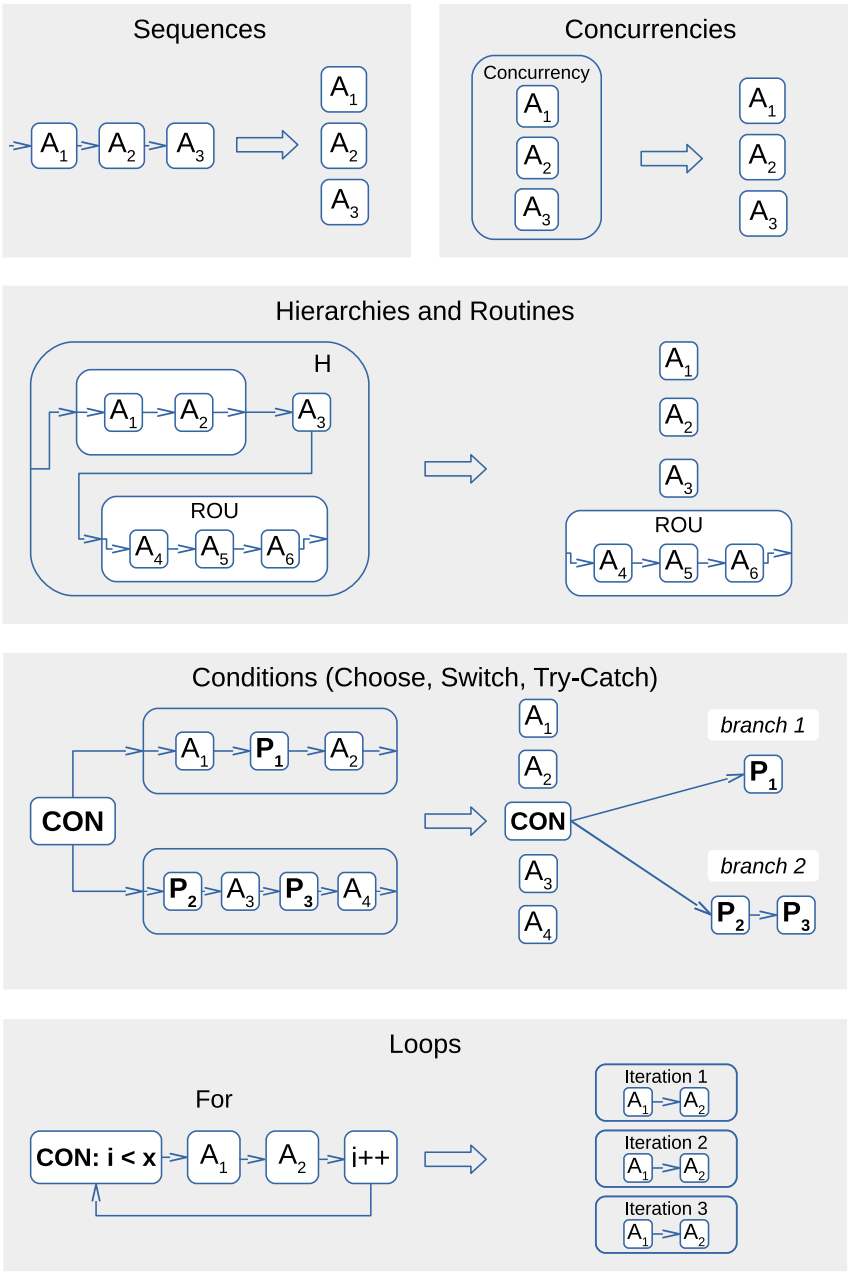


Figure 6.8: *Basic control flow building blocks in the HPFD (left of the arrow) and the corresponding CDTN representation (right of the arrow). Only for-loops are shown; do-while and while-do loops can be rolled out analogously. Nodes marked with* ***Px*** *are PIAs (see Def. 20), nodes marked with* ***ROU*** *(see 6.3.1) are routines and nodes marked with* ***CON*** *are conditions (see 6.3.1).*

in parallel. In practice, this may not be the case due to the RATN dependencies (i.e., data flow, world model and resource dependencies); the algorithm explained in Sec. 6.3.2 ensures that no dependencies are violated during execution. If BarrierConcurrencyStates are defined in the HPFD, they are treated the same as sequences and

are thus automatically parallelized.
The translation of PreemptiveConcurrencyStates to CDTN execution semantics is straight forward, as well. All task nodes in all of the children of a PreemptiveConcurrencyState *pc* are executed in parallel. If one of the ConcurrencyState's children finishes execution, all of its siblings (including their children) are removed from the dependency graph and cannot be executed anymore.

Routines (ROU) The user can define a task node to be a *Routine*. This indicates that the child states in that task node must never be executed in parallel, as they are tightly coupled. A Routine is thus a label for a task node to indicate: "do not attempt to parallelize the states inside of me". Routines are treated differently from other task nodes in the execution algorithm in Section 6.3.2. In fact, Routines are just executed as normal HPFDs.

Hierarchies Hierarchies are used by the programmer to define abstraction layers. If all dependencies of the hierarchies are satisfied, all child nodes of a hierarchy are executed in parallel (as long as none of their dependencies constrain the nodes in another way).

Conditions (CON) and Condition Dependencies Condition dependencies are used to model conditions in a CDTN, i.e., two mutually exclusive branches of execution. Before a condition is reached, idle processing power can be used to pre-compute results required in any of the branches. Upon execution, results are then discarded for branches that are not reached. In many cases, this is more efficient (from the point of view of execution time, not computational time) than waiting until it is known which branch will be entered.

Condition dependencies have not been explained thoroughly before as they do not represent a new dependency type per se. In fact, they are represented by conditional effects, i.e., if the HPFD state of the task node has several outcomes than the effects can be different for each outcome. Thus, the internal calculations of the HPFD state decide upon the outcome and the outcome maps to its own set of conditional world model or resource effects (see Def. 19). All states featuring several outcomes are automatically labeled as *condition task nodes*.

Fig. 6.4 shows an example condition dependency. The *Object Found* task node, which is a condition node, is connected to subsequent task nodes with condition dependencies (black arrows). If the detection did not find the target object (i.e., the world model fact *target-object-found* is *False*), the sensor head moves and records a new image. Otherwise the condition dependency to the planning task node is satisfied, which can

then generate a collision free path for the manipulator.

When parallelizing states with condition dependencies, care must be taken to not execute task nodes that cannot be undone. It is easy to undo the pre-computation of a result, i.e., it can be simply discarded from memory, and an outside observer would not notice. But grasping a cup changes the robot's environment state and can potentially fail. As given in Def. 20 such a task node is called PIA.

Loops *Loops*, such as for, do-while and while-do loops are rolled out. This means that a copy of the loop body is created for each iteration. If dependencies allow it, the different iterations of the loop can thus be executed in parallel.

In order to guarantee proper loop unrolling all RATN loops have to comply with the following structure. A loop task node must have exactly two outgoing transitions: t_1 for the case that the loop condition does not hold anymore (i.e., the loop has finished) and one transition for the loop body. The loop body has to be a combination of sequences, concurrencies and loops only. There must not be a condition that allows the loop body to break out of the loop without leaving the loop via the *loop* task node. Furthermore, if all preconditions and data dependencies of the loop are fulfilled, it must be able to generate the amount of iterations. This is required for loop unrolling an pre-computation of task nodes inside the loop bodies. If the condition for another loop iteration cannot be calculated beforehand and only during execution of the loop body, then the loop cannot be unrolled. This means that the parallelization benefits of CDTN execution can only be exploited locally inside the current loop iteration and not in advance for all iterations.

6.3.2 CDTN Execution Algorithm

So far, I have "dissolved" the rigid, over-constrained structure of the HPFD into a set of RATNs (along with their dependencies) that represent the same task. In this section, I present an algorithm for executing these task nodes. It is able to exploit parallelization and pre-computation opportunities, but also ensures that all dependencies are met.

The inputs for the algorithm are a CDTN and an initial world model. The CDTN can, assuming the initial world model having a certain structure, reach a goal using the plan encoded inside its task nodes. The goal is explicitly encoded as effects of the task nodes of the plan.

At the beginning of the main function, the CDTN and the initial world model

Algorithm 4 The CDTN execution algorithm in pseudo code.

```
 1: function MAIN
 2:     load_cdtn_and_world()
 3:     do_in_separate_thread(handle_task_node_termination())
 4:     while ¬task_finished do
 5:         dep_graph ← generate_dependency_graph()
 6:         execs ← extract_executable_task_nodes(dep_graph)
 7:         for all task_node t in execs do
 8:             do_in_separate_thread(execute_task_node(t))
 9:         end for
10:     end while
11: end function
12:
13: function EXECUTE_TASK_NODE(t)
14:     request_resources_of_task_node(t)
15:     if is_loop(t) then
16:         roll_out_loop(t)
17:     else if is_routine(t) then
18:         execute_with_default_execution_algorithm(t)
19:     else if is_PIA(t) then
20:         execute_PIA(t)
21:     else
22:         execute_task_node(t)
23:     end if
24: end function
25:
26: function HANDLE_TASK_NODE_TERMINATION
27:     while ¬task_finished() do
28:         t ← wait_for_next_task_node_to_finish()
29:         forward_data_of_task_node(t)
30:         propagate_effects_of_task_node(t)
31:         release_and_create_resources_of_task_node(t)
32:         if condition_task_node(t) then
33:             enable_PIAs_of_branch(get_active_branch(t))
34:         end if
35:     end while
36: end function
```

are loaded. In line 3 a separate thread is started, which calls the *handle_task_node_termination* function. The following while loop is the main driver of the CDTN execution algorithm. The loop is executed until the whole CDTN is executed. Inside the loop, the current dependency graph is calculated before all executable task nodes are extracted thereafter and finally started in their own thread. A task node is executable if all its dependencies are satisfied.

The *execute_task_node* function defines how task nodes are executed. In line 14 all resources of the target task node t are requested (and either blocked or consumed). If t requires a resource of type world model, then a dedicated world model has to be prepared by means of world model projection (as explained in subsection 6.2.3). If the task node is a loop state, the loop is unrolled. This means that for each loop iteration the loop body is copied, all input data of the loop is copied as well and

distributed to each iteration (for more information about loop unrolling see 6.3.1). If the task node has the control flow type *routine*, then the task node is executed with the default HPFD execution algorithm. If the task node is potentially irreversible, then a dedicated function to execute the PIA is triggered. All other task nodes are just executed as if they were simple HPFD states.

The last function *handle_task_node_termination* defined in line 26 was triggered in the main function inside a separate thread. Until the CDTN has not finished it waits for the next task node t to be executed, forwards its data outputs to the other states and propagates the task node effects into the world model. A hierarchical task node has been executed (also terminated) if all its child nodes have been executed. The CDTN unblocks all resources of the terminated task node and creates all resources specified in its effects. If t is a PIA, the root world model is updated with all intermediate world model task nodes between t and the last executed PIA.

If t is a condition, only the conditional results (mapping to the chosen outcome of the underlying HPFD state) are enabled. By using conditions, recurrent behavior can be created, i.e., action structures that lead back to previously executed states. If a state is executed that is connected to such an already executed state s via a transition, then s and all states accessible by s are re-enabled and can be executed again in order to reach the goal world state of the CDTN.

Unrolling loops may lead to a significant task node expansion, i.e., an increasing number of task nodes in the CDTN. These expanded nodes add up to the total number of nodes, which can be huge (several hundred tasks nodes with each of them having several hundred HPFD states, adding up to behaviors with more than 5000 states; see validation Chpt. 7). Checking the dependencies of all nodes of a CDTN would result in a huge computational effort, slowing down the whole behavior execution or even the whole system. Therefore, during dependency graph generation (line 5), a task horizon number *thn* limits the total number of nodes, for which the graph is generated and the dependency satisfaction check is performed. Every logical branch (in the case of the existence of *condition task nodes*) only gets a portion of *thn*, which depends on the number of conditions in the future task node set.

CDTN execution does in general not rely on the transitions defined in the underlying HPFD. Nevertheless the transitions are needed in the following three cases:

1. For the determination of the task nodes for the next dependency graph generation step; i.e., starting at the last executed PIA, the next *thn* nodes, which are not executed yet, are considered for the next step

2. Inside routines

3. In the case of recurrent behavior in order to re-enable already executed nodes; this can either happen via conditions or via loops, whose termination conditions are computed inside the loop bodies

6.4 Implementation

I implemented CDTNs in the task control framework RAFCON (see Brunner et al. (2019)). Fig. 6.9 shows the *deploy remote unit* action of the introductory example of this chapter implemented in RAFCON. This state machine is used to explain how the semantic information of RATNs can be modeled.

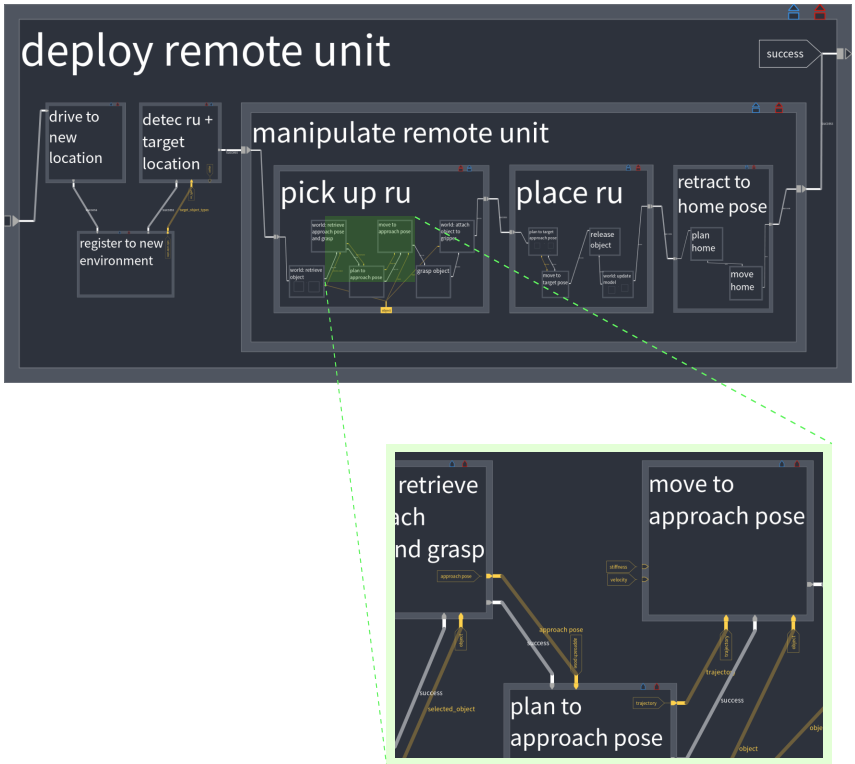


Figure 6.9: *RAFCON screenshot of the hierarchical state machine of the deploy remote unit action introduced in Fig. 6.1; The inset highlights some data flows (connections labeled approach_pose, trajectory, object etc.), which are used to model data dependencies of RATNs.*

Fig. 6.10 shows an overview of how the dependencies of RATNs are implemented. The figure shows an inner part of the CDTN visualized in Fig. 6.9. The data flow dependencies are shown in yellow. In the example of the figure the *move to approach*

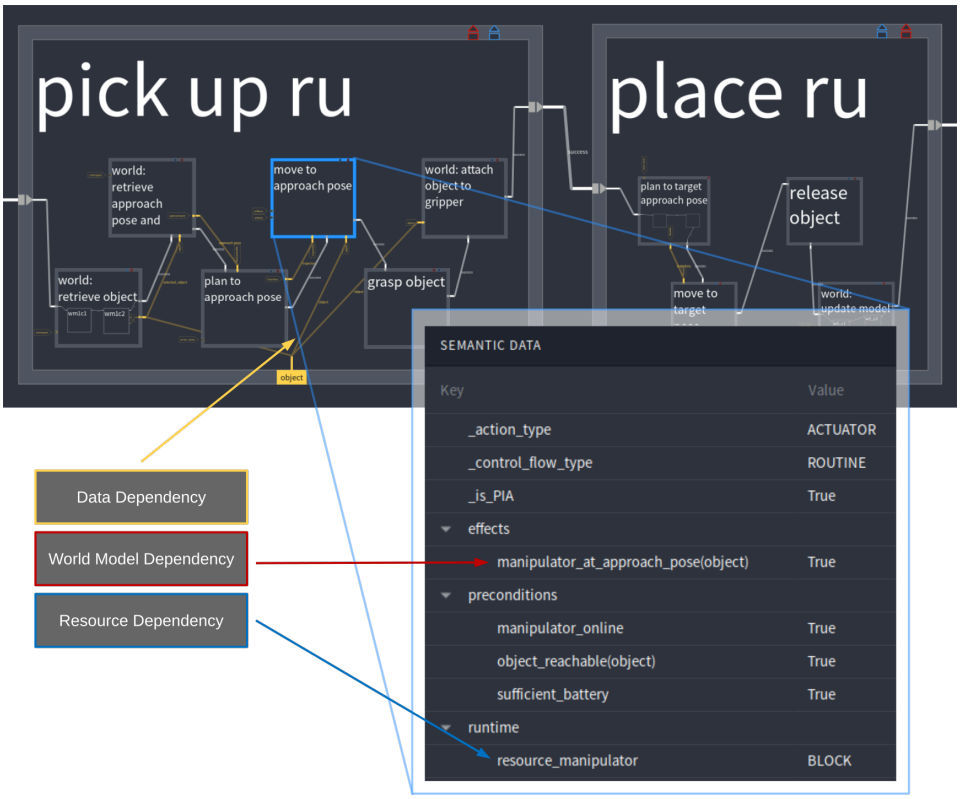


Figure 6.10: *The figure shows a sequence of RATNs in a scientific-load-carrier (slc) pickup task. The RATN with the name* move to approach pose *is selected. All three types of dependencies are shown including how they are implemented. Data flows (yellow arrows) represent the data dependencies, e.g., the* move to approach pose *task node needs a valid trajectory before it can move the arm. The Semantic Data Editor of RAFCON is used to implement world model (red) and resource (blue) dependencies.*

pose RATN requires a trajectory before it can be executed. Data flow dependencies are (as already stated in Sec. 4.5.3) data ports of RATNs that require an external data flow to be connected. This can lead to a very high number of data dependencies if a substantial amount of the behavior (which can feature far more than a thousand data flows, see Table 7.1) is modeled using RATNs. A more closeup view of the data flows of the *move to approach pose* RATN is shown in Fig. 6.9. World model dependencies (red) and resource dependencies (blue) are implemented as strings in the nested Python dictionary of the Semantic Data Editor and stored as key-value pairs in JSON-format on disk. The keyword for defining preconditions is *preconditions* and that for results and side-effects is *effects*. Specifying resources is done via the dictionary keys *CREATE*, *CONSUME*, *BLOCK*, and *RELEASE*. Resources required during the whole runtime of an action (including automatic releasing after action execution) can be listed under the *runtime* keyword. Furthermore, the world dependencies can have parameters, which can refer to data ports of the HPFD. Thus, data specific

semantic effects can be created, assuring the modularity and scalability of the CDTN approach. In the example in Fig. 6.10 the selected RATN possesses the world model effect *manipulator_at_approach_pose*(*object*), whereas *object* refers to the input data port *object* of the selected RATN.

The semantic state type information of RATNs, furthermore, consists of a *control flow type* and *action type*. Both can directly be defined inside the nested dictionary of the Semantic Data Editor. The dictionary keys are $control\_flow\_type$ and $action\_type$ (see Fig. 6.10)

Using ontologies for the RATN-specific semantic data values yields several advantages. Other programs, such as analysis and plotting tools, do not need to have their own glossary of all semantic values used in a CDTN, but can just load the ontology and can filter RAFCON's execution histories straight away. Furthermore, syntactically checking the semantic data values against a defined ontology identifies typing errors quickly. As presented in Chpt. 5, I developed an ontology plugin, which enforces the *action types* and *resource* dependencies to originate only from predefined ontologies. Finally, in the context of multi-robot scenarios, using the same semantics leads to transferable skills, i.e., actions programmed for one robot, which can directly be used for another.

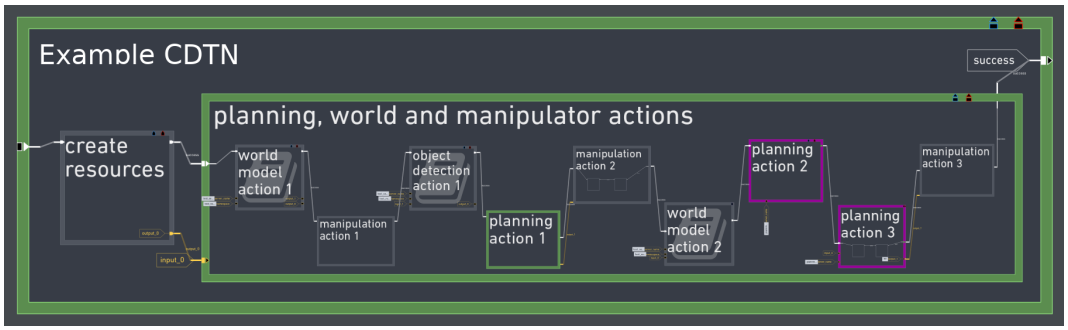


Figure 6.11: *State coloring during executing an example CDTN. The first planning action is currently executed and blocks the single available motion planner resource. The two other planning actions are waiting for their planner resource and are thus colored in purple.*

I replaced the default execution engine with a CDTN execution engine implementing the algorithm described in Sec. 6.3.2. RAFCON's plugin concept enables to use both execution engines interchangeable and even at the same time. Furthermore, I enhanced RAFCON's state machine editor to highlight task nodes that are currently waiting for resources (in purple, see Fig. 6.11). On top of that, the dependency graph created during CDTN execution can be exported to a colored graph for debugging purposes. The graph is built up using the Graphviz library[2] and its Python API. Sec. B of the appendix shows several example dependency graphs created during execution

[2] `https://www.graphviz.org/`

of the CDTN example shown in Fig. 6.9.

Currently, error handling is only supported via conditions, i.e., an action can check for a certain fact to be true and enables different conditional effects based on that. Implicit error handling (as presented in Sec. 4.2) is not supported by CDTN yet, but is part of ongoing and future work.

Finally, I extended the logging mechanism of RAFCON to generate Gantt charts for CDTNs as well in order to analyze the optimized task executions after runtime.

6.5 Application Scenarios

Several CDTN experiments have been performed in the course of this work. They show the applicability of CDTNs in the area of the ROBEX space scenario, an industrial mobile manipulation use case and a domestic service scenario. These experiments are presented in the next chapter, the validation chapter.

6.6 Related Work and Discussion

In this section I will highlight the abilities of various task execution frameworks concerning self-optimization, i.e., the online optimization of the robotic behavior, without the need for a programmer to intervene or perform this optimization explicitly. In literature, this is often referred to as transformational planning (see Beetz (2000)). Transformational planning modifies existing plans – regardless if they have been planned or manually designed – in order to meet different constraints and to optimize various metrics. Transformational planning is model-based, i.e., based on the current state of the system the future actions of the robot are adapted. Opposed to the model-based approach, learning-based approaches also exist (see Sec. 2.1), which often rely on reinforcement learning. Whereas the learning based approaches work fine in simulation, they are not applicable for real systems yet as the number of experiments, which are generating the data to learn with, cannot be scaled up as easily as for simulated systems.

The three main optimization metrics related to transformational planning often referred to in literature are robustness, speed, and resource usage.

Transforming a plan in order to reduce the number or probability of errors is the goal of GORDIUS (by Simmons (1988)), HACKER (by Sussman (1974)) and CHEF (by Hammond (1990)). As in my work about CDTNs, these approaches are able to integrate plan revisions into a partly executed plan, which is necessary to react to exogenous events and to avoid long planning pauses during execution. Apart from that, my work about CDTNs is related to the criticize/revise cycle of XFRM (see Beetz (2000)), the main difference being that my focus is to make the execution of plans more efficient rather than robust. It is important to note that the robustness of the tasks is not hampered by executing them using the CDTN execution algorithm. The projection of action effects and the creation of different versions of possible future world states are very similar to the work by Beetz (2000). On top of that, I provide a powerful resource model, a semantically more distinctive action definition and concepts for dataflow modeling.

Concerning highly parallel execution approaches, the concepts of automatic parallelization in, e.g., compiler construction (see Fox et al. (2014)), dataflow programming (see J. B. Dennis (1980)), software pipelining (see Ruttenberg et al. (1996)), and speculative multi-threading in the context of CPU development (see Yiapanis et al. (2015)) are relevant. What my approach has in common with these methods is the aim to execute operations in parallel and/or in advance as much as possible. To the best of my knowledge, such parallelization and pre-computation approaches have not yet been applied to make the task control of robotic systems more efficient.

Finally, the distribution of executable actions to available resources poses many open scheduling challenges (see Smith (2005)), e.g., scheduling under complex constraints, managing change during execution runtime, and the advantages and disadvantages between self-scheduling systems and central schedulers in the case of multi-agent systems. Related work to integrated planning and scheduling can also be found by Muscettola (1993), who applied temporally flexible constraint networks for short term scheduling of the Hubble Space Telescope.

Chapter seven

Evaluation

The evaluation of my concepts will be performed in two steps. First, several experiments are described that validate my approach presented in the last three theory-oriented chapters. Second, I will discuss weaknesses and strengths of the proposed concepts by comparing them to related work.

7.1 Validation Experiments

In order to show the applicability of my approach in real robotic scenarios and simulation, I describe and discuss two experiments. For both experiments the core concepts of my work are validated which are:

- Autonomous behavior design using HPFD-GL (described in Chpt. 4)
- Whole-lifecycle analysis of the robotic task (described in Chpt. 5)
- Self-Optimization using CDTNs (described in Chpt. 6)

The first experiment is the final demonstration of the ROBEX project, a *Moon-analogue* mission that our LRU rover had to perform fully autonomously.

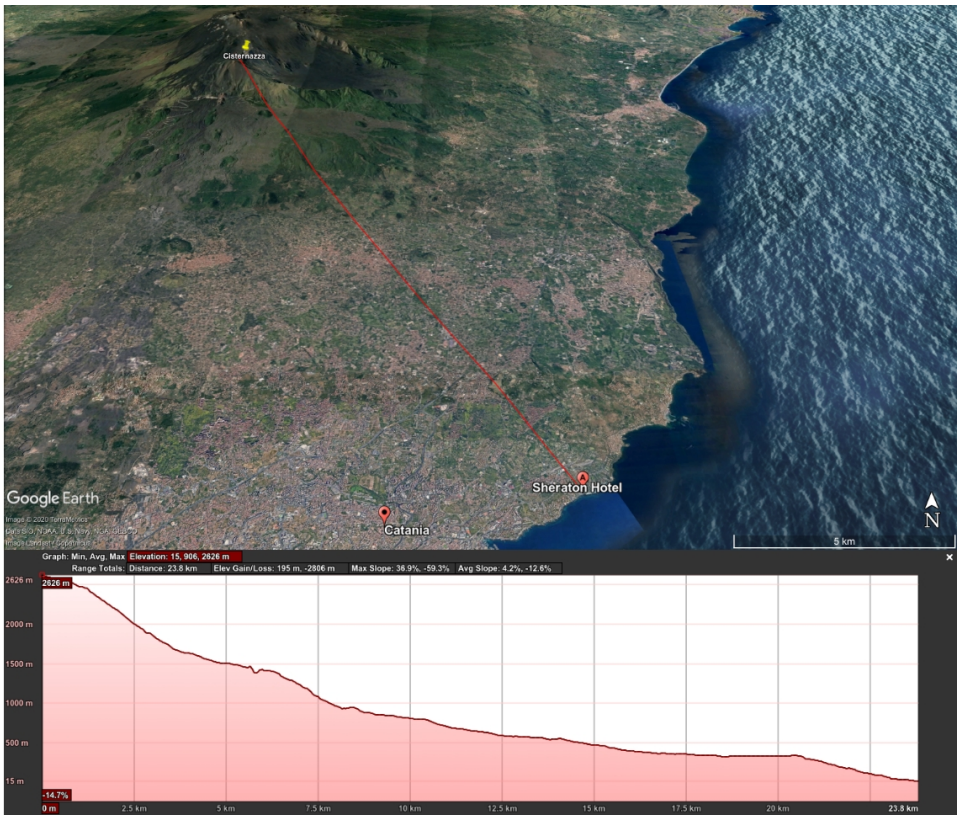


Figure 7.1: *The location of the ROBEX test site on top of Mt. Etna and the location of the mission control center in Catania, in Sicily, Italy. The lower part of the figure shows the inclination from the control center in Catania to the test site, which was located more than 2600m above sea level.*

7.1.1 A Space Scenario: The ROBEX Mission

The ROBEX project was a Helmholtz Association project, which took place from 2013 to 2017. The highlight of the project was a planetary exploration mission on the Moon-analogue site on Mt. Etna, in Sicily, Italy, in 2017. The exact location of the test site was 300 meter west of the Cisternazza crater, a harsh region 2600m above sea level (see Fig. 7.1). The scientific area featured different inclinations, craters, big rocks, sand and stone types of various materials. The mission control center was located in the Sheraton Hotel in Catania, 23.8km beeline to the test site. A radio link connected the mission control center with the robots and the control center of the operator crew on top of the mountain.

Mt. Etna challenged us for many things: Next to various lighting conditions during the day the temperature differences were enormous. Our equipment had to endure less than -0 degree Celsius during night to up to 25 degrees during daytime. If directly exposed to the sun, our components had to suffer even more. Thus, the cooling system of the LRU had to be designed to cover that. Furthermore, the weather

changed often during the three weeks of operation. During rain and heavy wind bursts of up to more than 100km/h we had to keep our equipment inside a container. Apart from that, we executed our experiment regardless of whether the ground was very humid from the rain or the morning dew, or whether it was dry and dusty because of the heat. Finally, a major challenge of the mission was to enable the robot to act autonomously for long time periods (several hours of operations per day), as a broad range of complex mission goals had to be fulfilled.

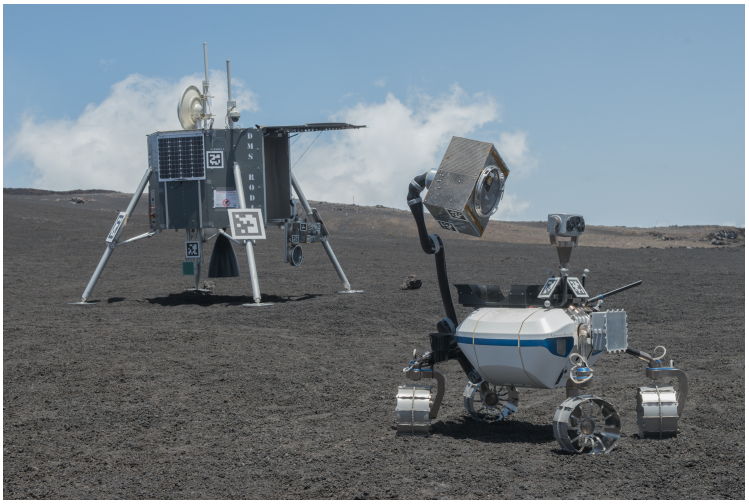


Figure 7.2: *A scene during the ROBEX experiment on Mt. Etna: the lander in the back and the LRU in front, currently manipulating one of the seismometer payload boxes.*

Experiment Description

The goal of the planetary exploration experiment in ROBEX was to analyze the geologic composition of a remote site with seismometers. The task of the rover was to fetch the seismometers from a landing unit and place them on the ground at positions (see Fig. 7.2), which were predefined by our scientists. Furthermore, the robot had to level the ground and had to produce a test impulse to ensure the correct deployment of the instrument. Once the rover deployed the instrument correctly, it could measure remote impulses from real seismic events (that happen frequently on a volcano).

During the time of three weeks (from June 14th 2017 to July 6th 2017), the LRU executed several submissions:

- *Active Seismic Measurement (ASM)*: In order to estimate the geological composition of the target region, the rover had to drive to several predefined spots and had to perform a seismic measurement with one instrument.

- *Seismic Network*: The rover had to deploy a network of four measurement units in order to estimate the source of the seismic events by means similar to triangulation.
- *Sample Return*: In order to analyze the stone and sand materials of the target region the rover had to take a soil sample with a shovel and return it to the lander.

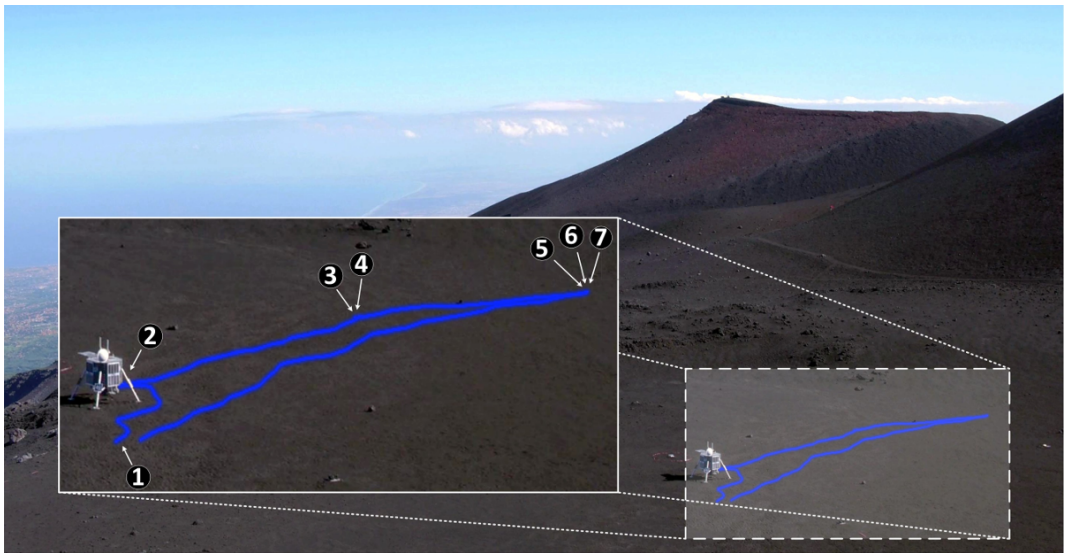


Figure 7.3: *Semi-automated Google Earth export of the trajectory of the LRU on a mission during the ROBEX project on Mt. Etna, Sicily. The labels with numbers show the location of some manually selected events, which where automatically projected into the spatial domain (i.e., the map).*

Figure 7.3 shows an overview image of the *Active Seismic Measurement* (ASM) mission at the test location on Mt. Etna. I selected some key frames (1-7) from the mission to illustrate the execution logging. Figure 7.4 shows real scenes from the mission for the very same key frames. The robot starts in front of the landing unit (1) and drives to the seismic instrument hand-over position. There, the rover docks to the seismic instrument (2) and waits for the landing unit to release the instrument. Once the rover has stored the instrument in its carrier, it drives to the first deploy location and places the seismometer on the ground. To ensure the correct function, the robot levels the ground with the instrument (3). Once the instrument is correctly deployed, the rover waits for an external seismic impulse, which the instrument records (4). The rover repeats the process (6 and 7) at another location and drives back to the landing unit. During the mission, an operator had to change the battery (5).

For the experiment RAFCON was in charge of orchestrating all software modules of the robotic system's functional tier (see Sec. 2.4). We used RAFCON to implement robot skills of various abstraction layers, which included service calls for navigation,

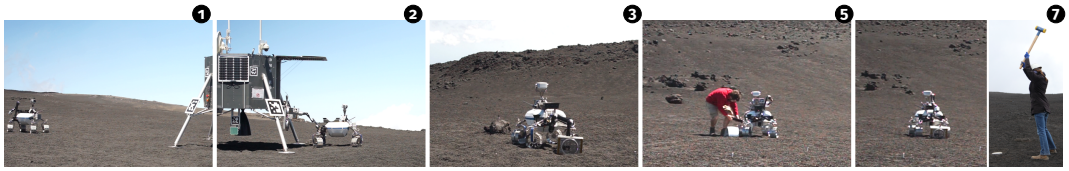


Figure 7.4: *Scenes from the Moon-analogue experiment on Mt. Etna, Sicily in Italy. The labels refer to those of Fig. 7.3, and also Figs. 7.7 and 7.8.*

object detection and localization, manipulation, planning and grasping. One essential requirement RAFCON had to fulfill, was the ability to provide longterm autonomy capabilities to the system. These include robust behavior execution, efficient mission analysis and well-organized mission updates.

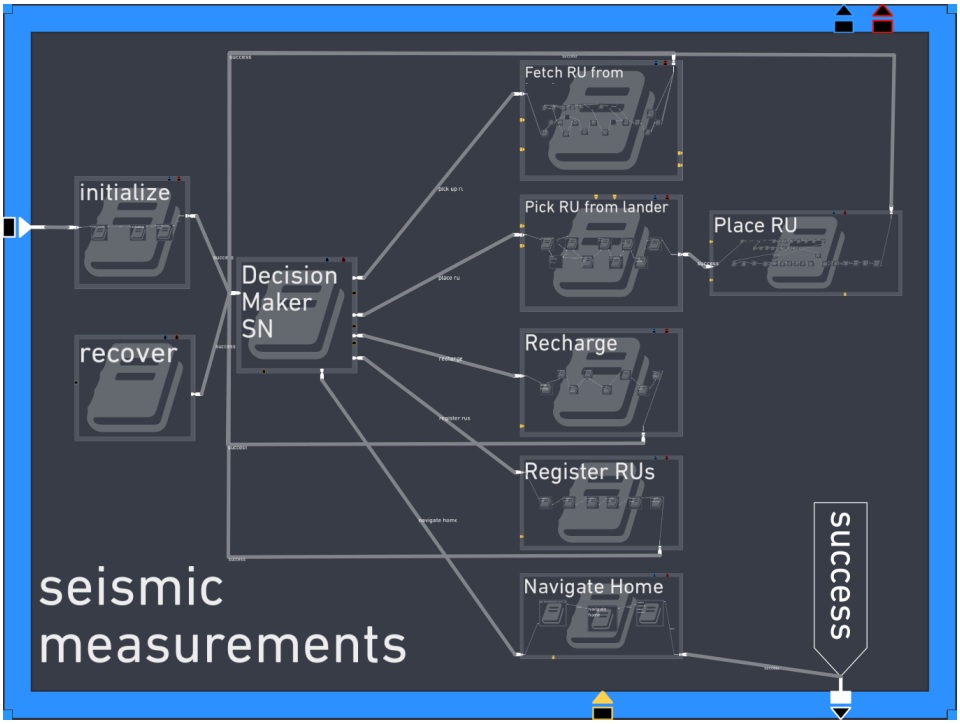


Figure 7.5: *The first two levels of the ASM mission. In the center it shows the decider state. On the right the single sub-tasks of the mission are shown.*

A central decider state (see Fig. 7.5) continuously analyzed the current robot state and the world model and decided upon the next sub-task to execute. The decider consisted of a decision tree, which could easily be monitored and debugged (and was already discussed in Sec. 4.3.4, Fig. 4.19).

The interaction with the world model consisted of the interaction concept presented in Sec. 4.6.3, which is based on the powerful CYPHER query language. We placed various reentry points inside the mission to continue the mission also after mission failure or rover reboots. On the highest level, the “recover” state could be used to

re-initiate the mission by, e.g., loading a backup world model dumped to disk during a previous mission run. Fig. 7.5 also shows the first internal layer of the various sub-tasks in a hierarchical manner. I disabled the data flow (normally visualized in yellow) for the sake of a better overview.

Experiment Analysis

In the following, I showcase how the proposed analysis concepts were used to support the mission on Mt. Etna.

Table 7.1: *Statistics of the state machines for the main ROBEX missions. The columns from left to right show: the state machine name, the total number of states (column 2), transitions (column 3) and data flows (column 4), the maximum hierarchical depth, the percentage of states covered with error handling, the percentage of states covered with direct error handlers, the maximum error handling depth and the average error handling depth (for more information about the error handling metrics see Sec. 5.3.1)*

State machine	State Count	Transition Count	Data Flow Count	Max Depth	States with EH	States with Direct EH	Max EH Depth	Average EH Depth
ASM	1532	1996	1857	8	36.62%	2.48%	2	1.02
Seismic Network	1455	1905	1801	8	34.64%	2.73%	2	1.03
Sample Return	947	1184	1115	7	28.51%	2.66%	2	1.05

Design Time Analysis: To create the autonomous behavior of the LRU we made extensive use of the code syntax and data integrity checks. Thus, many errors could be eliminated before runtime. To make a statement about the robustness of the task implementation, Table 7.1 can be taken into account. It shows that, although the state machines are big, the error handling coverage is still at more than 28 percent of all states for all missions (i.e., in average, each third action could fail and the rover knew how to recover). This and the fact that some states even had nested error recovery procedures made the state machine sufficiently robust. The percentage of states with direct error handling is rather low compared to the total amount of states with error handling. This shows that the majority of the error handling was performed by indirect error handling procedures, i.e., handlers that were attached to container states and that were able to handle a multitude of errors. Examples for such handlers were actions that, in case of a failed manipulation or object detection action, just canceled the current action, re-positioned the rover and then re-performed the

(manipulation or detection) action again. Direct error handling especially was needed for intricate manipulation steps, e.g., altering the stiffness and force-direction parameters for in-contact motions that clamped the end-effector. Another example for direct error handling were critical actions in which simple retrying was not possible, e.g., in situations where the rover had just grasped a remote unit and encountered unforeseen contacts such as un-modeled rocks on the ground.

Table 7.2 shows statistics of the ASM mission concerning the number and usage of RAFCON LibraryStates. In total, the ASM mission features 1330 library instances. The figure shows that many libraries are heavily reused throughout the overall behavior. The table differentiates between different library categories. The "Combined" category refers to higher level libraries (higher level skills and tasks), which include libraries of several other categories.

Table 7.2: *LibraryState statistics of the ASM ROBEX mission: the first column shows the category of the library, the second column total number of libraries in that category and the last column the total number of library instances per library category.*

Library Category	Total Number of Libraries	Library Instances in ASM
Manipulation	128	288
Gripper	3	20
Motion Planning	30	145
Navigation	47	97
Object Detection	25	34
World Model	23	279
Middleware	14	184
Combined	186	283

For this experiment, model checking was unfortunately out of scope because of resource constraints. For simulation we relied on the RoverSimulationToolkit (see Hellerer et al. (2016) and Schuster et al. (2016)), a multi-body simulator written in Modelica.

Runtime Analysis: Next to the benefit of logging outputs with filterable severity levels, we made extensive use of the online execution history view. Especially having access to every executed movement of the manipulator, with its stiffness and damping factors, and interpolator parameterization, helped us to find errors quickly. As explained above, we were also able to execute and monitor the RAFCON behavior even from the 23.8km (beeline) far away mission control center in Catania. If any anomaly occurred, we could pause the execution engine and examine the current context data of the state machine.

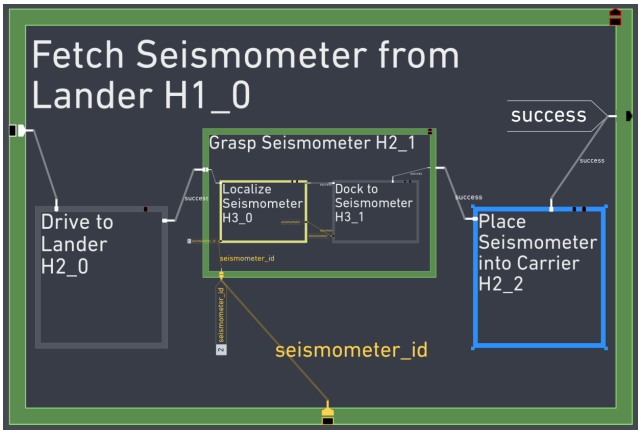


Figure 7.6: *Different colors show the different highlighting modes of RAFCON: Green shows currently executing states, yellow already executed states before their transition to the next state (especially usable during step mode), and blue the currently selected state.*

Furthermore, the graphical viewer of RAFCON supports the mission control team to keep track of the current internal state of the autonomous behavior. Fig. 7.6 shows the execution of a small example state machine inspired by the real "fetch seismometer from lander" action of the ROBEX mission. Green highlights the currently active states, light yellow the last executed one, and blue the currently selected state. The logic flow is shown in gray lines and the data flow in dark yellow lines. Next to the data ports (dark yellow letters, e.g., "seismometer_id") the current data is visualized. For example, the mission control immediately sees that the current value of the "seismometer_id" has the value 2. This saves a lot of time, normally spent searching through the logging output or in the execution history view for the variable of interest.

During the test runs for the final demonstration mission in front of the broad public and the press we often had to update the robot behavior from the control center. The reason for this was either because the robot encountered some error or we, the mission control, decided to change the test setup and mission objectives. To perform the mission updates, we followed the approach outlined in Sec. 5.4.

The **first step** consisted of bringing the robot into a secure state, which we achieved by moving the robot to even terrain and canceling current manipulation movements. In case of software failures, we restarted the respective components. In the case that the robot was encountering an error during manipulation, we manually retracted the manipulator from the objects involved in the task using custom impedance movements.

In the **second step**, we manually performed relevant actions, which the robot either failed to execute or which were too critical to be done fully autonomously in the current state, step by step. This included re-approaching and re-detecting objects,

and triggering manipulation procedures such as approaching an instrument box, opening or closing the gripper, or leveling the ground for instrument deployment.

The **third step** consisted of updating the robotic plan by changing task parameters (such as the instrument to pick up next from the lander) or by changing the behavior, e.g., by adding error detection and handling routines for bad lightening conditions or strong wind bursts.

In the **forth step**, we updated the robot's belief state in a way such that it reflected all changes that we had performed manually and that could be represented using the robot's semantics. This included setting object states (i.e., a flag to disable a defect remote-unit-holder of the lander, or the manipulator's load parameters), adding or removing objects (e.g., newly detected or defect objects) or updating object frames such as the robot tool frame.

In the **final step** we re-initiated the robot and let it resume the mission from predefined recovery entry points. These recovery points included, e.g., the re-scanning of the close-up area of the robot in order to register to another object (such as the lander) or to update the local obstacle map of the rover.

We performed the procedures required for all steps using predefined maintenance and recovery state machines, which were located either on the robot itself or which we created on the fly by the mission control team and executed remotely. Especially the online modification possibilities of RAFCON sped up the mission update process significantly, as the time for stopping the behavior, deploying the new behavior to the robot and starting the robot again could be saved.

Post-Mortem Analysis: I analyzed all the exceptions that occurred during the three weeks of operation. Table 7.3 shows that many errors occurred during the testing and mission days on Mt. Etna. Many of them were caught, which means that the error handling procedures in the task implementation targeted the correct spots. Both, the total number of exceptions and the number of uncaught exceptions yield interesting results.
On the one hand, the object detection was not robust enough. Both the "get first object from list" and the "detect object" actions belonged to the class of object detection routines. Moving the robot slightly and re-detecting the object most of the times helped to continue the mission. Sometimes, we had to manually remove bugs or butterflies (on Mt. Etna there are many of these in June) from the object surface in order to proceed.
On the other hand, the planning of motions was not properly encapsulated in error recovery strategies, as can be seen by the low caught rate of the states "plan joint

Table 7.3: *The ten exceptions that occurred most often during the ROBEX missions during three weeks (between 2017-06-14 and 2017-07-06). The columns from left to right show: States in which the exception occurred; total exception count; relative error severity (see equation 5.10); number of caught exceptions; at how many different locations this state was included.*

State Name	Exception Count	Relative Error Severity	Caught	Different Origins
get first object from list	91	16.73	59	17
plan joint move	62	11.40	6	31
open	41	7.54	29	15
close	39	7.17	36	14
check transform validity	35	6.43	26	10
get submap id	25	4.60	23	3
plan task	18	3.31	7	5
detect object	18	3.31	5	8
move rel cartesian impedance	15	2.76	12	9
load world	13	2.39	0	7

move" and "plan task". This was especially true in the first days of the testing phase, when we realized that both the lander and the robot slightly sank into the sand. For this reason, combined with the fact that the test area featured many slopes, the lander and the robot had different relative poses to each other, varying from those during the tests in our laboratory. This affected the workspace of the robot and thus the reachability of objects. Therefore, we had to adapt some of the calculated poses in order for the robot's manipulator to be able to reach its goal poses again. As we use a statistical planner based on sampling, simple repositioning and re-planning resolved the remaining motion planning issues.
Finally, the "load world" action was never caught, but this is unproblematic as the action only returned with an error if there were syntax errors in the world description or the world representation node was not started yet. These errors (both of human nature) could be avoided by respecting the spelling and syntax rules of our world model description language and following the correct robot system startup procedures.

Next to analyzing exception statistics, I created two different types of runtime-analysis visualizations based on the gathered execution history log date. Fig. 7.7 shows a diagram of the most important resources the rover used during the mission: the Jaco 2

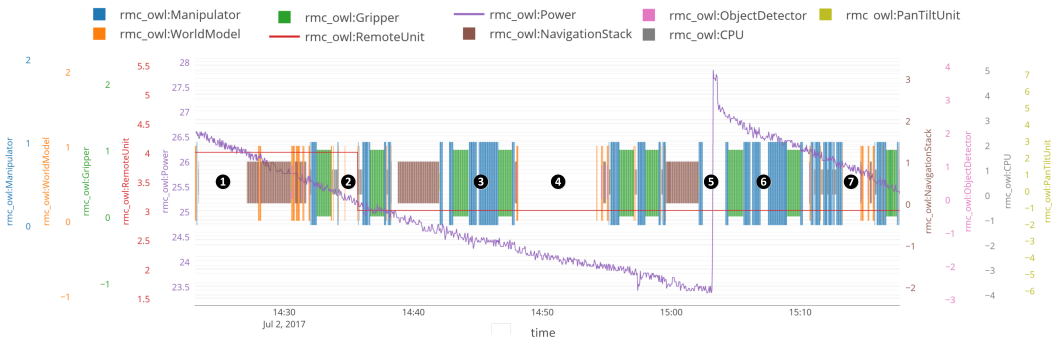


Figure 7.7: *IRCP-SYA-Bars diagram of the the state machine created for the ASM mission of ROBEX. The labels show milestones and interesting events of the mission and are the same labels as those of Figure 7.3.*

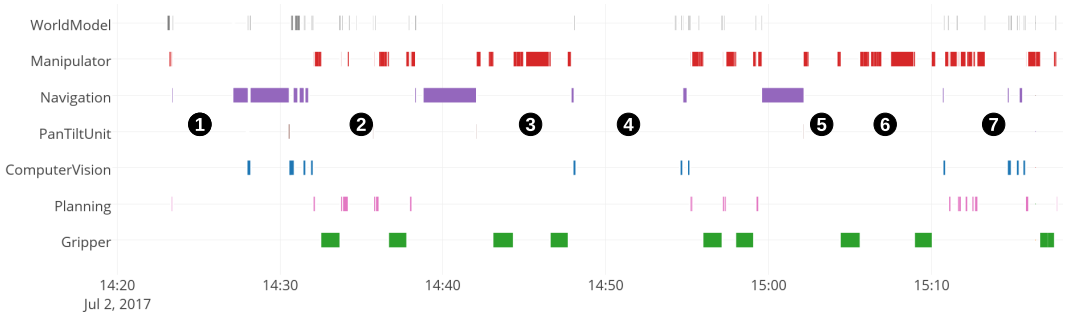


Figure 7.8: *Gantt diagram of the ASM mission. All executed actions are grouped by their action type (shown on the y-axis). The labels refer to those of Figure 7.3.*

manipulator[1], the docking interface (i.e., the gripper) on the manipulator, the navigation stack, the world representation, the seismic instruments (*rmc_owl:RemoteUnit*) and the power of the rover (the object detector and pan-tilt unit actions cannot be seen at this resolution and the CPU usage is not visualized for the sake of clarity). The diagram shows that during the initialization (1), the docking interaction with the landing unit (2), the battery exchange (5), and the seismic measurements (4 and 7), most of the resources of the rover are idle, as the autonomous behavior had to wait for external events (or the finalization of the startup procedure). Additionally, during the battery exchange (5), the power voltage is replenished to 27.8 volts. While the rover places and levels the instrument (3 and 6), the diagram shows heavy usage of the manipulator and the docking interface.

Fig. 7.8 presents a Gantt chart of the execution times of the individual software and hardware components. The plot shows that the rover spent (next to navigation) most of the execution time during manipulation while moving either the manipulator (red) or docking or undocking the seismic instrument with the docking interface (green).

[1] `https://www.kinovarobotics.com/en/products/robotic-arms/kinova-gen2-ultra-lightweight-robot`

Table 7.4: *Metrics for actions grouped by action types for the ASM state machine of the ROBEX mission. The columns from left to right show: Name of the action type class defined in the action type ontology; total time spent for all actions of the respective class; number of executed actions per class; time spent relative to the whole runtime of the state machine; time for the action with the longest runtime; time for the action with the shortest runtime; average time per call; weighted relative execution time (see equation 5.9); all times are given in seconds*

Action Type	Spent Time	Total Call Count	% of Total Time	Max Time	Min Time	Time per Call	WRET
Navigation	1164.57	31	40.98	192.52	0.10	37.57	4.17
Manipulator	595.01	100	20.94	82.34	0.10	5.95	6.86
Gripper	550.49	12	19.37	71.94	0.60	45.87	0.76
Planning	101.82	19	3.58	17.14	1.34	5.36	0.22
ComputerVision	89.96	30	3.17	15.84	0.15	3.00	0.31
WorldModel	43.67	106	1.54	2.69	0.12	0.41	0.53
PanTiltUnit	4.09	6	0.14	2.79	0.09	0.68	2.7e-3

The logged data can be used to create various metrics as shown in Table 7.4, which can be used to identify bottlenecks of the overall task. In this table (as well as in Fig. 7.8)

- *Navigation* refers to navigation actions such as driving to a new map location
- *Manipulator* to manipulator movements
- *Gripper* to grasping actions, e.g., opening and closing the docking interface
- *Planning* to motion planning actions
- *ComputerVision* to computer vision related actions such object detection and localization routines
- *WorldModel* to world model operations such as storing or retrieving semantic information to/from the world model
- *PanTiltUnit* to operations in order to move the pan tilt unit and to decide where it has to look next

It can be seen that manipulator, navigation and gripper actions took most of the time. Furthermore, for navigation and gripper actions, the average time per call is the highest. An explanation for the navigation action runtimes is simply the covered

distance that the rover traveled during the mission. In turns out that the gripper itself is the biggest bottleneck of the mission, as only 12 calls take more than 17.82% of the overall runtime. One reason for this is that a tight form closure between the gripper and the remote unit is needed for the manipulation tasks. At the same time, the gripper has to be very lightweight as the gripper's weight reduces the overall payload of the manipulator. Thus, we inserted a high mechanical transmission, which reduces the speed of grasping. Concerning the computer vision actions, it has to be said that they would have made up an even higher percentage of the overall runtime if we would not purely rely on AprilTags. In a space mission employing such tags definitely makes sense, as they work great for detecting the exact location of objects and other agents, even from a long distance. On Mt. Etna we could perform 360deg scans in less than 16 seconds, which would not be the case if applying more sophisticated object detection algorithms not relying on AprilTags. This especially holds true as LRU does (for power saving reasons) not feature a GPU.

Manual Optimization Based on Analysis Results

In this section, I will summarize the benefits of the analysis results. As can be seen in Table 7.4, the gripper was one of the main bottlenecks in the ROBEX mission from a runtime perspective. We redesigned it in order to increase the execution speed for future missions (such as the ARCHES mission[2]). The new gripper features a new transmission design which enables to reduce the grasping time to 9.1s, approximately 25% of the original one. We reduced the exerted force also by the same factor, which is unproblematic as our tests showed that the reduced force is enough for grasping boxes of up to 2.5kg.

Next to navigation and grasping, the action classes motion planning, manipulation and computer vision are responsible for vast parts of the overall runtime.

For motion planning, the situation is more complex. The planning actions would have made up a much higher percentage of the runtime if we would have truly relied on planned manipulation movements only. As we could select the target environment by ourselves, we could rely on many assumptions (e.g., the non-existence of obstacles at the target object locations) and thus taught many of the manipulator movements in the laboratory. For this reason, the actual planning time for a truly unknown terrain would have been significantly higher.
The duration used by the manipulation actions is already quite optimized, as we tuned

[2] https://www.arches-projekt.de/en/helmholtz-future-topic-project-arches/

the taught trajectories to be short and efficient. Moving the joints faster increases risks such as overheating or collision to other parts of the robot's structure (especially taking wind bursts into account). Thus, the runtimes for manipulation movements are disproportionate low. For real, planned motions the runtimes would vary much more and would be executed slower to compensate for the higher trajectory uncertainties originating from the planner.

We performed the majority of all object detections by using the April Tags that we attached to every manipulated objects (RUs) or to every object in reachability during manipulation (e.g., the lander). As stated above, we did not perform a proper scene reconstruction (because of resource reasons) and modeled the ground as a flat surface, which, of course, is a non-realistic assumption. For the next missions, we want to include a real environment modeler responsible for creating a voxel map of the target area. This voxel map is then included into the path planner, which can then avoid the ground contours during trajectory generation. In order to speed up the computer vision runtimes, we already added a NVIDIA Jetson board[3], as it offers decent performance for very small power footprints.

For future missions (such as ARCHES), we want to reduce the amount of assumptions and thus want to get closer to real space missions. Thus, we will have to use a planner for all motions, which will increase the planning times drastically. Essentially, this is the point where the runtime optimization using CDTNs is paying off, as it basically is able to cancel off the majority of the planning time. In order to show this, I created a simulation which is realistic in terms of the action runtimes.

Optimization Based on CDTNs

In the following, I will present experiments using a simulated LRU rover (see Fig. 7.9). The simulator employs the same software that runs on the real system for belief state modeling (i.e., the *Neo4j* based world model), manipulation planning (i.e., the *rmpl* planner by Lehner and Albu-Schäffer (2018)) and behavior execution (i.e, RAFCON). The robot's kinematics, structure and controllers are simulated using the Gazebo simulator (by Koenig and Howard (2004)) including the *ros_control* package (see Quigley et al. (2009)). The simulator uses the real CAD models of the lander, the RUs and the LRU and is thus realistic in terms of collision simulation and path planning durations. In the simulation, no assumptions are made about the terrain. Thus, all manipulator movements (except the relative, impedance controlled movements) are

[3] see `https://www.nvidia.com/en-us/autonomous-machines/jetson-store/`

planned in order to avoid collisions with obstacles and the robot's own structure.

I examined three different test scenarios (all of which were part of the real ROBEX missions):

1. **Grasping of a single remote unit from at the lander**: This scenario assumes that the rover already navigated to the lander and positioned properly in front of the RU to pick up. The first action is the parsing of the scene, followed by the docking motion, the RU pick-up and the RU placement on the RU carrier on the rover's back. The final action is the movement to bring the manipulator into transport position.

2. **Deployment of a single remote unit**: This scenario assumes that the rover traveled already to the deploy pose in the field. After scene reconstruction and ground analysis it grasps the RU and uses it for ground leveling by shoveling the ground using the RU, thus creating a flat surface for RU positioning. Subsequently, the LRU places the RU on the ground and optimizes the contact between the RU and the ground using impedance controlled sine motions, which press the RU towards the ground. (In the following, both motions, the leveling and the contact optimization, will be referred to as "leveling".) After RU placement, the LRU creates a test impulse to ensure the RU's correct positioning by punching on the ground using the manipulator. Finally, it brings the manipulator back in transport position and carefully moves away from the RU in order not to reposition it again.

3. **Remote unit network deployment**: This scenario equals the ASM mission of the ROBEX scenario. For this the rover fetches four RUs from the lander and deploys them in a Y-shaped network at a nearby target site defined by a scientist.

For the second and third scenario, I simulated different variations for the RU deployment. In the first variation, deployment can fail, which is detected by the test impulse. Upon failure the detection, the robot picks up the RU drives 0.7 m forward and redeploys it. This variation is called "with redeployment". The second variation affects the leveling. For areas that are flat and feature fine sand the leveling motions are not required. As currently modeled, the leveling motions are fixed and do not include planning.

Fig. 7.9 shows different scenes of the RU deployment scenario. At first, the rover drives to a position of interest (POI). After placing the RU, which includes ground leveling and contact optimization, a seismic test impulse is generated by the rover

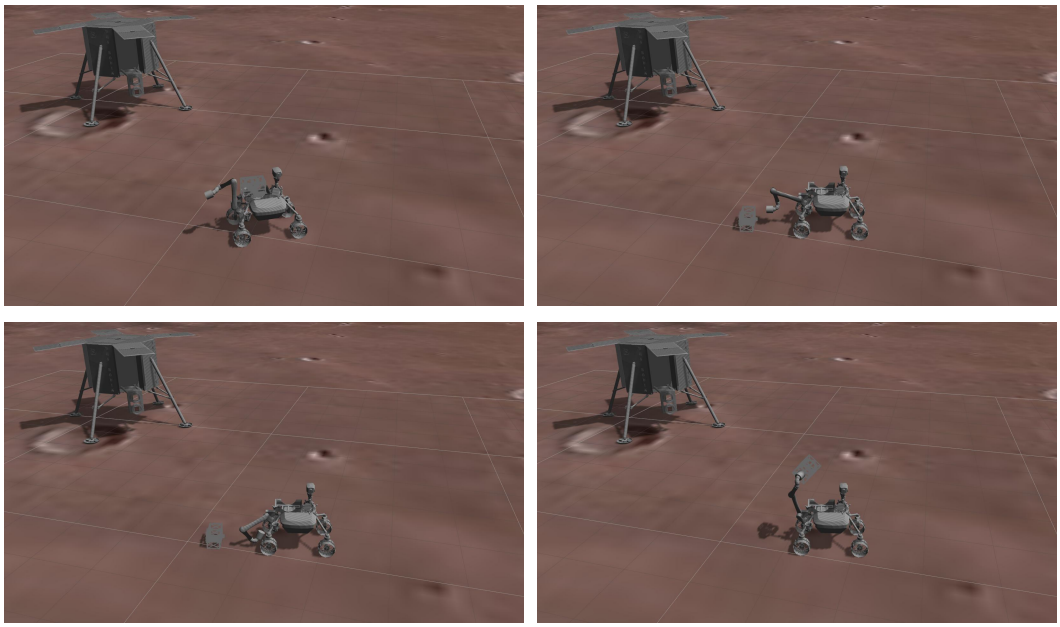


Figure 7.9: *Scenes of the ROBEX Gazebo simulator. From top left to bottom right: LRU driving to POI, LRU after RU deployment, LRU doing a seismic test impulse, LRU picking up the RU*

in order to check the RU's correct deployment. In this case the RU could not be deployed correctly: thus the rover picks it up again to place it at a nearby location.

Table 7.5: *Metrics for actions grouped by action types for the simulated ASM state machine of the ROBEX mission. The columns from left to right are the same as in Table 7.4. The time is given in seconds.*

Action Type	Total Spent Time	Total Call Count	Time per Total Spent Time	Max Time	Min Time	Time per Call	WRET
navigation	1236.24	57	51.89	88.28	0.05	21.69	2.44
planning	1006.64	164	42.25	19.50	1.12	6.14	5.73
manipulation	814.59	342	34.19	12.64	1.05	2.38	9.66
utils	319.23	92	13.40	9.02	0.50	3.47	1.02
gripper	290.40	32	12.19	9.13	9.05	9.07	0.32
vision	96.48	60	4.05	2.04	1.14	1.61	0.20
world	66.39	333	2.79	1.34	0.00	0.20	0.77
utils + init	1.82	130	0.08	0.03	0.01	0.01	0.01

Table 7.5 shows the metrics for the execution of the whole mission in simulation. I simulated the actions in a way that their runtimes resemble the real runtimes on the mountains as close as possible. Only the gripper runtime was reduced to 9.1s, as we

already replaced and optimized it (as described above). On top of that, the majority of all manipulator motions are now planned, in opposite to the ROBEX mission on Mt. Etna, where we used a lot of hard-coded trajectories for object manipulation. The table shows that the weighted relative execution times for planning and manipulation is the highest. Thus, they are a reasonable target for optimization.

Table 7.6: *Runtime statistics for the ROBEX test scenarios with HPFDs and the corresponding CDTNs. "r" means with redeployment, "l" means with leveling motions, "rgt" means with realistic gripper timings (9.1s). Execution times are averages of five runs. MP=Motion Planner.*

# test case	HPFD	CDTN (1 MP)	CDTN (2 MPs)	Improvement (1 MP)
deploy 1 RU	95.49s	75.91s	74.63s	21.85%
deploy 1 RU (r)	210.56s	164.13s	156.92s	25.47%
deploy 1 RU (r, rgt)	248.7s	199.56s	199.34s	19.85%
deploy 1 RU (r, l)	261.67s	218.14s	216.12s	17.41%
grasp RU from lander	72.41s	65.23s	64.34s	11.14%
deploy 4 RUs (l)	1017.1s	957.06s	943.26s	7.26%
deploy 4 RUs (r, l)	1606.62s	1424.7s	1394.36s	13.21%
deploy 4 RUs (r, l, rgt)	1764.51s	1619.9s	1611.06s	8.70%

The runtimes of the different execution modes (i.e., HPFD vs CDTN execution) for each of the test scenarios (including different variations) is shown in Table 7.6. The default HPFD implementation is compared with the CDTN execution using both one and two planners. The last column shows the runtime improvement of the CDTN execution using two planners compared to the sequential execution. The table shows that substantial runtime improvements of more than 25% can be achieved.

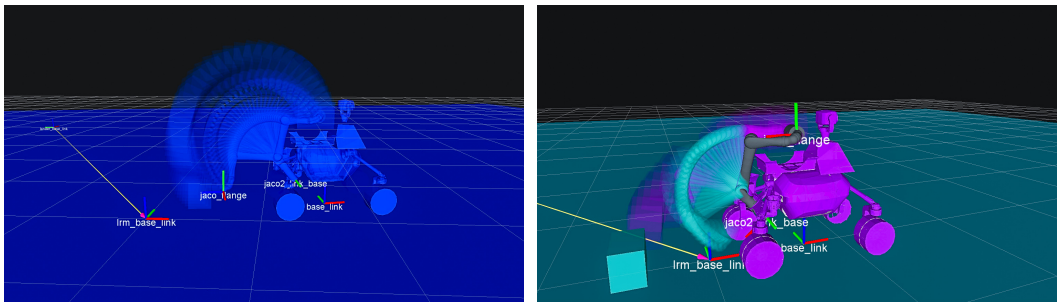


Figure 7.10: *Planner visualization during ROBEX simulator execution; the left side shows a trajectory planned by a single planner during sequential execution; the right side shows the planned trajectories during CDTN execution using two motion planners.*

Fig. 7.10 shows two screenshots visualizing the motion planner during simulator execution, including its geometric world state and the planned, collision free trajec-

tories. On the left side a single planner (as used in the sequential execution using a HPFD) is shown. The right side shows the world representation of two planners, as used during CDTN execution. For the teal motion planner, the RU is currently located at the ground in front of the robot's manipulator. For the purple motion planner the RU is currently docked at the robot's manipulator, which has to be taken into account in order to avoid self-collisions or collisions with the ground. Important robot frames, such as the robot's base frame (called "base_link") and the end-effector (called "jaco_flange") are visualized in order to depict the robot's current position (which may vary substantially from the planner's world state as the planner tries to compute a trajectory for a future action). On top of that, the current robot position is depicted in gray for debugging purposes.

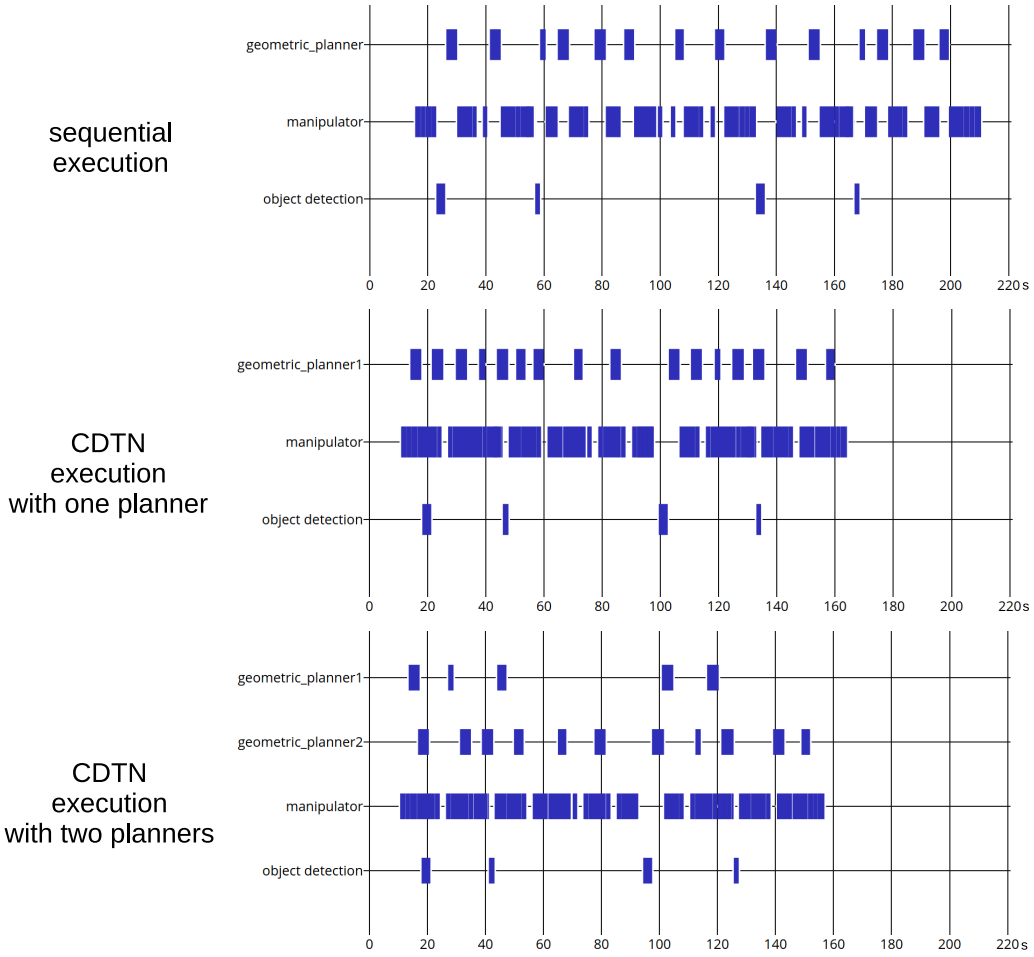


Figure 7.11: *Gantt diagrams of executing the ROBEX simulation test case "deploy 1 RU (r)". The first chart shows a sequential execution, the second chart a CDTN execution using one motion planner and the third chart a CDTN execution using two motion planners.*

In Fig. 7.11, the test scenario "deploy 1 RU (r, rgt)" is analyzed using Gantt charts. The top row shows a Gantt chart representing the sequential execution, the second

chart shows the CDTN execution using one motion planner and the third chart the CDTN execution using two motion planners. It can be seen that manipulation and planning, and the planning actions themselves overlap during execution. However, the overlap of the planning actions themselves is only minimal, which is one reason why the CDTN execution using two planners is often not significantly faster than the execution using only one. By just using one planner most paths can already be pre-computed. Thus, the inclusion of another planner does not optimize the overall execution time in all cases. For analyzing the course of the entire mission the Gantt chart of the "deploy 4 RUs (r, l, rgt)" scenario is shown in the Appendix C.

Table 7.7: *Resource overview of the simulated remote-unit-network deployment mission. Next to the modeled resources the number of resource requirements is shown.*

resource name	number of blocks	resource name	number of blocks
manipulator	133	motion planner 1	19
robot base	29	motion planner 2	17
gripper	12	world model copy 1	19
pan tilt unit	29	world model copy 2	17
world model	81	ground analyzer	8
navigation stack	29	scene parser	12

Table 7.7 shows all resources modeled for the simulated RU-network deployment mission. Next to the resource names, the table shows the total requirements of the respective resource by all actions executed during the mission. Table 7.8 and Table 7.9 show all world model facts created during the mission, including their number of creations (by action effects) and requirements (by action preconditions). Note, that recurrent CDTN behavior (needed for iterating over all the RUs) is able to delete already created world model facts. This is the reason why a world model fact can be created more often. In total, 58 world model facts are used during the mission.
The list of data dependencies would be too big to be shown in this work using a table as there are several hundreds of them (as I modeled a substantial part of the seismic network mission by using RATNs, see Table 7.1).

Finally, there are several dependency graphs for different mission setups given in Appendix B. As the dependency graphs are huge and meant for debugging using a viewer featuring a full-text search their details cannot be identified using the paper format of this thesis. Therefore, some areas are enlarged in order to be able to investigate the action dependencies for some specific nodes.

Table 7.8: *First part of the world model dependency overview of the simulated remote-unit-network deployment mission. Next to the modeled world model fact, the number of fact creations (by an action effect) and the number of fact requirements (by an action precondition) is shown.*

world model fact	# creations	# requirements
init_done	1	2
planner_init_done	2	36
adjusted	8	8
grasped_ru_on_lander	4	4
moved_to_ru_on_back	8	8
object_detection_enabled	8	48
reached_pick_up_point	4	4
back_at_impulse_approach_pose	8	8
moved_to_deploy_pose_above_carrier_at_lander	4	4
remote_unit_on_ground	8	8
grasped_on_carrier	8	8
parsed_scene_after_ru_pickup_from_lander	4	4
impulse_done	8	8
moved_ru_up_from_ground	4	4
reached_deliver_point	4	12
retracted_from_ru	4	4
synced_with_arm_on_lander	4	4
moved_ru_away_from_lander	4	8
released_on_carrier	4	4
reassigned_rodin_after_ru_pick_up_at_lander	4	0
next_iteration	3	0
released_ru_on_carrier_at_lander	4	4
parsed_scene_at_lander	4	24
synced_with_arm	4	4
moved_forward_to_ru_on_lander	4	4
retracted_from_ru_carrier_at_lander	4	8
mission_completed	1	1
robot_at_second_try_deploy_pose	4	0

Table 7.9: *Second part of the world model dependency overview of the simulated remote-unit-network deployment mission.*

world model fact	# creations	# requirements
...		
signal_good	8	4
signal_bad	8	4
moved_base_away_from_lander	8	4
moved_forward_to_ru_on_ground	4	4
ru_released	8	20
lifted_from_carrier	8	8
at_home_for_impulse	8	8
moved_towards_ground	8	8
contact_optimized	8	8
retracted_from_ru_carrier	4	8
at_impulse_approach_pose	8	8
moved_home_after_grasping_from_ground	4	0
moved_home_after_deployment_on_carrier	4	4
analyzed_ground	8	16
ground_leveled	8	8
copied_world1	1	1
copied_world2	1	1
placed_down_on_carrier_at_lander	4	4
grasped_ru_on_ground	4	4
moved_to_deploy_pose_above_carrier	4	4
at_approach_for_grasping_ru_on_lander	4	4
placed_down_on_carrier	4	4
retrieved_ru_id	4	4
at_approach_for_grasping_ru_on_ground	4	4
arm_in_transport_position_after_ru_pick_up_at_lander	4	4
at_home_after_impulse	8	8
parsed_scene_at_deliver_point	4	24
object_detected	8	16
finalized_mission	1	0
continue_mission	4	3

7.1.2 An Industrial Scenario: Logistics Tasks on AIMM

The goal of the second experiment is to show the applicability of my approach in an industrial mobile manipulation scenario. The robot we used for this experiment is the AIMM platform. In terms of complexity, the AIMM robot is similar to LRU. It features a robotic manipulator (LWR iiwa from Kuka[4]), a mobile base, two stereo camera pairs, a pan-tilt unit and two LIDARs. Two main advantages over LRU are that it has a desk to store objects during navigation and it is equipped with four times the computing power of LRU. As for LRU, RAFCON is used to control all actions of AIMM on top of the realtime controllers. A broad set of skills is implemented via HPFDs that are able to access all actuators and sensors.

The experiment was part of the Automatica trade fair 2018. In this context, I show the applicability of RAFCON for industrial use cases, results of the whole lifecycle-analysis, and an optimization based on CDTNs performed on the AIMM simulator.

Experiment Description

At Automatica, we showed our AIMM demonstrator carrying out various logistic tasks autonomously. It had to carry *small load carriers (SLCs)*, small blue boxes filled with different content. Fig. 7.12 shows the experiment layout. The SLCs had to be transported from a work desk to other desks, where different parts where needed. For the Human-Robot-Interaction (HRI) desk AIMM had to deliver parts of a toy robot, that were assembled by a human and a robot in an interactive manner. The other desk was a fully autonomous assembly desk, where item profiles were assembled for which the instructions were generated using only a single image of the final product. For this desk, AIMM had to deliver screws and brackets. A gravity rack with five layers served as storage place, in which more than 50 SLCs could be stored. The SLCs had to be collected at predefined pick-up zones (teal) and brought to predefined delivery zones (red).

Fig. 7.13 shows real scenes from the trade fair. At the top, our robot manipulates a SLC in front of the gravity rack. As can be seen, our demonstrator was part of a fully autonomous factory-of-the-future demonstrator set-up by the *Robotics and Mechatronics Center* of the DLR. The lower two images show AIMM during pick up (left) and placement (right).

[4] https://www.kuka.com/en-de/products/robot-systems/industrial-robots/lbr-iiwa

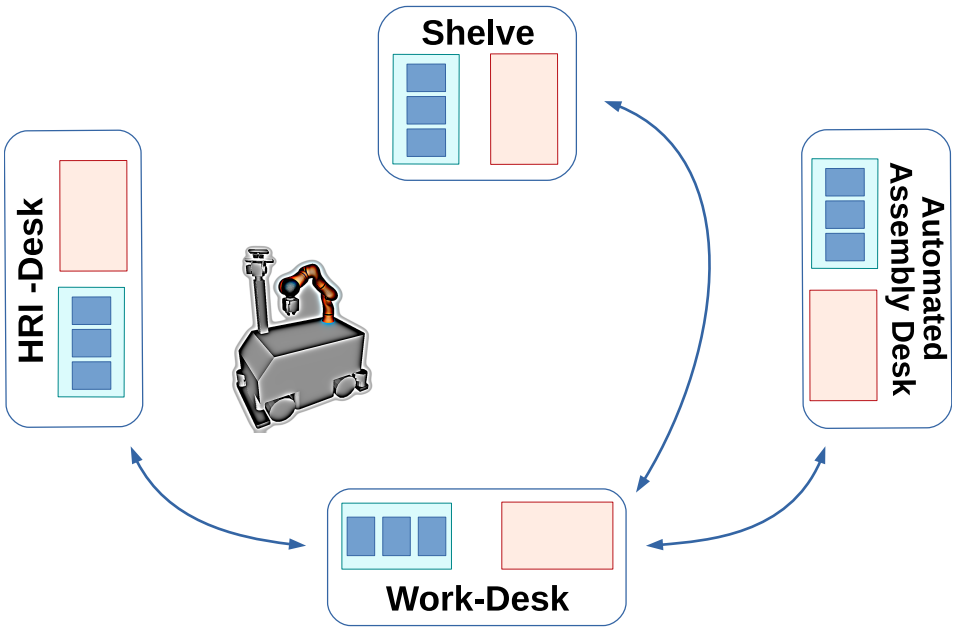


Figure 7.12: *The Automatica trade fair experiment. The AIMM robot has to care about the workflow between different work desks. It has to carry blue boxes with various content from pick-up zones of the desks (teal) to delivery zones (red) of other (or the same) desks.*

Experiment Analysis

We ran our demonstrator five days in sequence for nine hours per day. Thus, a lot of logging data accumulated. In total, we recorded more than 15GB RAFCON execution history logs. Similar to the ROBEX experiment in Sec. 7.1.1, I start with the design time analysis.

The overall behavior of AIMM consisted of four sub-tasks:

- *Desk Supply Chain*: In this sub-task the SLCs were continuously fetched from the pick-up zone of the Work Desk in front and were placed at the delivery zone of the same desk. This was the sub-task AIMM executed most often as the visitors of the trade fair were allowed to place the SLCs in different positions, stack them and fill them with different content. Our main goal was to show that AIMM could handle a wide range of situations autonomously.

- *Shelf Supply Chain*: The goal of this sub-task was to deliver SLCs to the gravity rack and SLCs with different content back to the Work Desk.

- *HRI Supply Chain*: This sub-task cared about delivering parts to the HRI desk. The parts consisted of several building blocks for a toy robot.

Figure 7.13: *Real scenes of the Automatica trade fair 2018, in which our robot AIMM was responsible for various logistic tasks. Top: AIMM during SLC manipulation; Bottom left: AIMM during SLC pickup; Bottom Right: AIMM during SLC placement*

- *Assembly Supply Chain*: In this task AIMM fed the autonomous assembly cell with various parts (screws and brackets).

The high level behavior of each sub-task was straight forward. It consisted of a high level skill responsible for selecting the appropriate scene and parts to deliver. Subsequently, a while loop executed the fetching and delivery of the SLCs. However, for executing the grasping and placing movements for objects we used a special concept: the *dynamic task modification* of a HPFD by the behavior defined in the HPFD itself. The general approach is shown in Fig. 7.14. The dynamic task modification concept works by first querying object properties from the world model. Based on this information it selects the library path and name of the robot skill to be added to a provided template using the *Skill Provider*. The skill to be inserted can, for example, be a grasping (such as in Fig. 7.14) or a placement strategy. After selection, the library is inserted into the template (by executing the "generate state" action) and all higher level parameters are connected to the generated skill using data flows

(e.g., the grasp width in Fig. 7.14). After execution of the skill, it is deleted, as the next time the template is executed another skill will possibly be needed. The skill is surrounded by an error handling procedure, which cares about deleting the skill even in the case that the skill fails. In principle, this approach enabled us to create some form of behavior inheritance, in which the skill parameters configured how the abstract behavior of the template was extended.

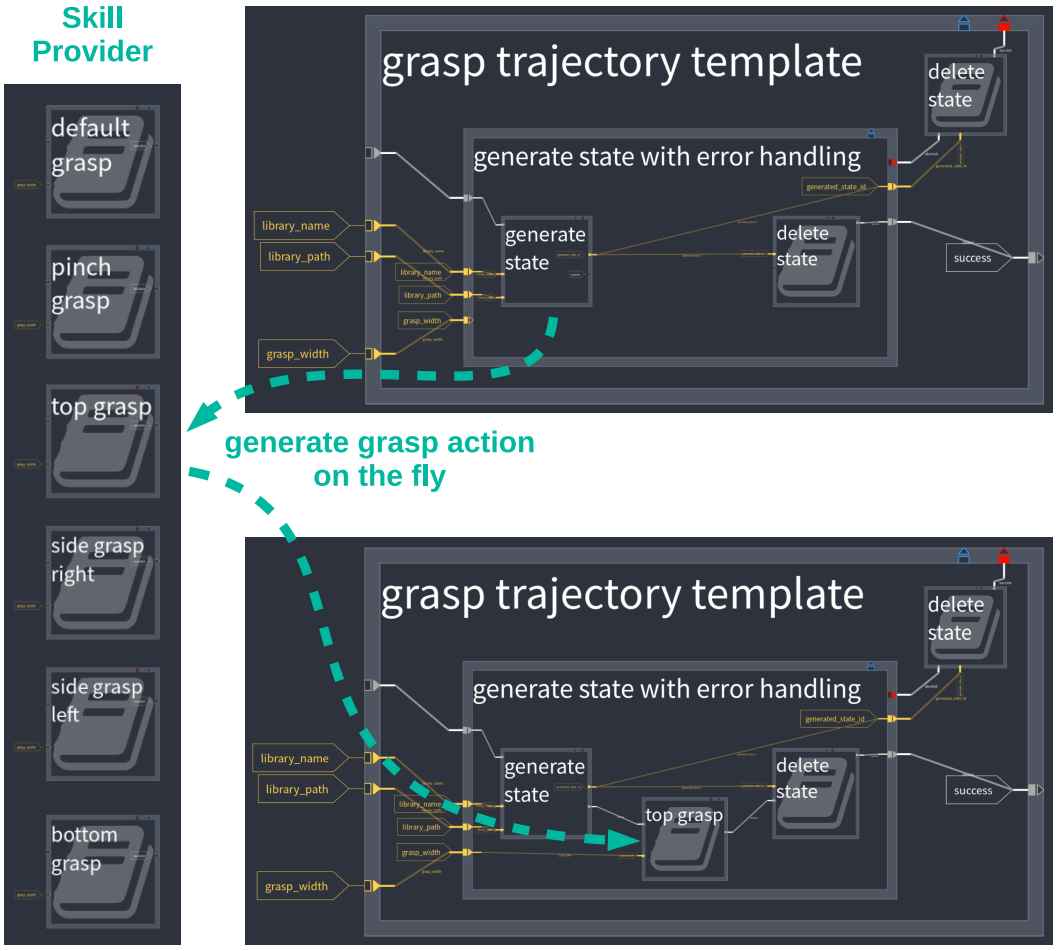


Figure 7.14: *Dynamic, online modification of the grasp trajectory (encoded in HPFDs) during object picking.*

Being able to dynamically modify a task inside RAFCON was a key requirement when designing the task control framework. Both the user and the task itself must be able to perform such modifications in order for the task being flexible also during runtime. This was one of the reasons, why we relied on the high-level programming language Python. Python supports reflective, meta and functional programming paradigms and has thus introspective capabilities similar to those of Lisp (see van Rossum (1997)). These capabilities enable Python programs to construct and execute program fragments on the fly, which is exploited by the dynamic task modification.

Table 7.10: *Statistics of the state machines for the four different tasks during Automatica 2018. The columns are the same as those in table 7.1.*

State machine	State Count	Transition Count	Data Flow Count	Max Depth	States with EH	States with Direct EH	Max EH Depth	Average EH Depth
Desk Supply Chain	2211	2986	3313	14	59.52%	1.9%	3	1.41
Shelf Supply Chain	4166	5651	6259	15	62.22%	1.99%	3	1.42
HRI Supply Chain	4364	5915	6582	15	77.27%	1.97%	4	1.72
Assembly Supply Chain	5488	7411	8460	15	62.54%	2.05%	3	1.43

Table 7.10 shows various metrics concerning AIMM's autonomous behavior. Compared to the behavior of LRU in the ROBEX scenario (see previous section) the state machines of AIMM are more than two times bigger. The states count ranges from 2211 to 5488, the transition count from 2986 to 7411 and the maximum depth goes up to 15. It has to be noted, that the number of states in Table 7.10 refers to the semantic number of states, i.e., all states including the LibraryStates and their children. This means, that if a LibraryState with 60 children is included 10 times in the final state machine, it will add 600 states to the overall number of semantic states. The total number of different LibraryStates equals 149, the total number of different ExecutionStates make up 193. Concerning the error handling depth, the AIMM behaviors were even more robust than the ones in the ROBEX scenario. Up to 77 % percent of all states had error handling routines. The maximum error handling depth goes up to four, i.e., the robot's behavior could handle four errors in various hierarchy levels in a row and could recover from that. The reason for the complexity is that our system could handle boxes with different contents (in terms of color, shape and mass) and in different positions. Even stacked box configurations were possible and could be disassembled. Furthermore, AIMM could navigate around humans in the near proximity and was robust against perturbations in its own world model, e.g., the robot could handle situations in which an operator stole already picked up boxes or added additional ones onto the robot's desktop. The reason for the high robustness of the system was the fact that our demonstrator ran on a trade fair with important visitors from science, politics and industry. As the visitors also were allowed to place, fill and rearrange the boxes by themselves, sophisticated error handling routines had to care for successful task execution. Similar to the ROBEX scenario, the great

majority of error handling was performed by indirect error handlers, which most often just re-triggered actions that failed (most often because a user disturbed the robot). In the ROBEX scenario, relatively more direct error handlers were needed to cover more specific uncertainties imposed by the unstructured environment on Mt. Etna.

As already highlighted in Sec. 5.3.3, design time statistics are important but not sufficient as the task execution is subject to many uncertainties and unforeseeable events, such that the overall runtime cannot be determined beforehand. Moreover the robot's behaviors are subject to change as new actions can be generated and inserted on the fly during execution (by using dynamic task modification). Thus, post-mortem analysis has to be used to find robustness and performance bottlenecks.

During the Automatica 2018 trade fair our system encountered many exceptions while executing the various tasks, especially the *Desk Supply Chain*. These exceptions are shown in Table 7.11. In opposite to the experiments of LRU in the previous section, the *Desk Supply Chain* task was even more robust[5]. Only three times the whole task failed completely. In the first case, the *unblock resource* failed once as there was a race condition in the unblock resources service. In the second case, the *load world* failed once as the yaml-file, which is loaded to generate the initial world model of the robot, contained a typo. Finally, the *look at* action failed once as the module controlling the pan-tilt process failed, a case that our error handling could not cover. It is important to note that these three errors are only those errors, in which the robot knew that it could not handle the error. Of course, there had been a few more errors, of which the robot was not aware of, but which have been manually fixed by the operators. These were especially errors that the robot classified wrongly and tried to handle with a non-working error recovery strategy in an endless loop until the operator helped the robot out, e.g., placing an object in a position to which the robot had an unobstructed view or moving an object into the workspace of the robot's manipulator.

Similar to Table 7.4, Table 7.12 shows runtime statistics of various action types for the *Desk Supply Chain* task of AIMM. In this table

- *Manipulator* refers to manipulator movements
- *Planning* to motion planning actions
- *ComputerVision* to computer vision related actions, such as object detection and localization routines

[5] Care must be taken when interpreting this table, as for ROBEX I also included the exceptions of the two weeks mission preparation time.

Table 7.11: *All exceptions that occurred for the Desk Supply Chain task during Automatica (between 2018-06-18 and 2018-06-22). The columns are the same as for Table 7.3.*

State Name	Exception Count	% of Total Count	Caught	Different Origins
plan task goal	1256	42.18	1256	8
plan task local	784	26.33	784	5
move relative impedance tcp	261	8.76	261	7
move location navigation	199	6.68	199	1
resolve collision	181	6.08	181	2
plan local link move	180	6.04	180	7
select storage	31	1.04	31	2
plan task	29	0.97	29	5
select grasp	23	0.77	23	2
move relative platform	14	0.47	14	4
unblock resource	8	0.27	7	6
move joint path	2	0.07	2	2
detect object	2	0.07	2	2
get first pose stamped from object list	2	0.07	2	1
load world	1	0.03	0	1
add item	1	0.03	1	1
update pose	1	0.03	1	1
get property	1	0.03	1	1
look at	1	0.03	0	1
retrieve transformation	1	0.03	1	1

- *Navigation* to navigation actions, such as driving to a new location on a map
- *World* to world model operations, such as storing or retrieving semantic information to/from the world model
- *Gripper* to grasping actions, e.g., opening and closing a gripper
- *Tools* to calls of interfacing and conversion actions, such as transforming an Euler pose into a quaternion
- *ResourceManagement* to action caring about resource handling (blocking and unblocking)

Table 7.12: *Metrics for actions grouped by action types for the Desk Supply Chain task executed by the AIMM robot. All times are given in seconds.*

Action Type	Spent Time	Total Call Count	% of Total Time	Max Time	Min Time	Time per Call	WRET
Manipulator	898.50	295	34.89	26.43	0.35	3.05	1.80
Planning	821.46	546	31.90	11.60	0.35	1.50	3.04
ComputerVision	504.67	163	19.60	7.72	0.44	3.10	0.56
Navigation	301.75	40	11.72	15.75	3.47	7.54	0.08
WorldModel	147.38	1567	5.72	2.31	0.02	0.09	1.56
Gripper	122.27	75	4.75	2.14	0.40	1.63	0.06
Tools	35.15	2926	1.37	0.37	0.00	0.01	0.70
Resource-management	0.96	120	0.04	0.05	0.00	0.01	8.3e-4

The table shows that manipulator actions, planning actions and vision actions make up most parts of the overall runtime. These three action types, however, are not similar. Manipulator actions perform real physical movements and change the robot's environment, while planning and computer vision actions merely compute results: collision free movements on the one hand and object locations on the other hand. In our case, the *computing* actions take up even more time in total than the actions moving the robotic manipulator or its base. By using CDTNs, large parts of this time can be saved. The next section describes how I performed this for experiments with the AIMM simulator.

Runtime Optimization using the AIMM Simulator

I performed the optimization of AIMM's behavior in a simulator employing the Gazebo framework (see Koenig and Howard (2004)). I use it to demonstrate the efficiency benefit of CDTNs in the constraint object manipulation task of the Automatica trade fair use case.

AIMM is simulated analogue to the real system, i.e., the simulated robot consists of a mobile base, a 7-DOFs LWR with a two finger gripper, a pan-tilt unit, four cameras and two LIDARs. Fig. 7.15 shows the simulated robot during object manipulation.

For simulation I use a setup with three computers. One computer hosts the Gazebo

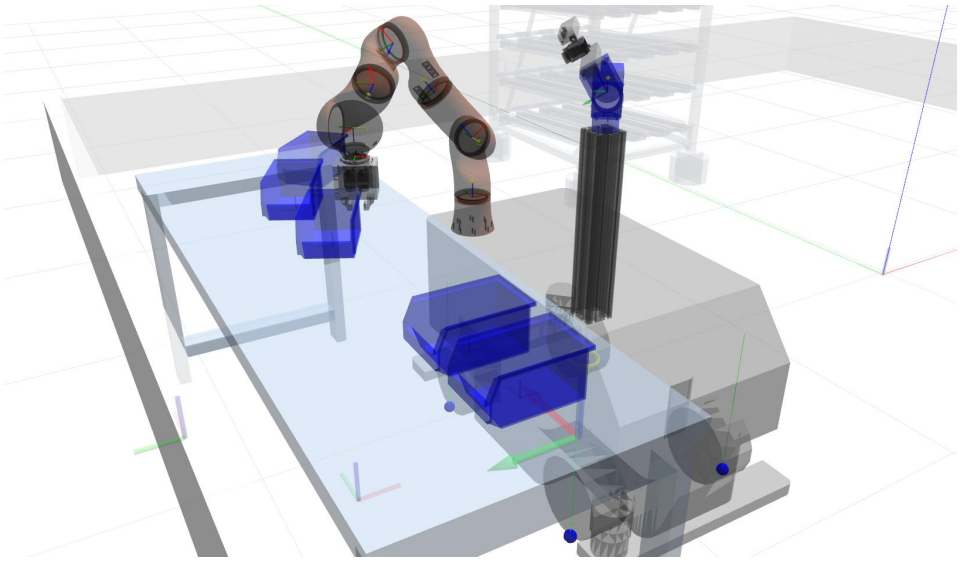


Figure 7.15: *The AIMM platform in the Gazebo simulation environment with transparent objects in order to visualize frames and contacts.*

simulator with the robot controllers and sensor data providers. The second computer runs the ROS navigation stack[6], the path planner process pool with several instances, the world model pool with at least the same number of instances as the path planners, the object detection process pool and several interfacing nodes. For world modeling I use the HPFD world model based on Neo4j. The third computer runs RAFCON as the task control framework, together with RViz and a Gazebo client for visualization and debugging. The *Links and Nodes Manager* (see Florian Schmidt and Robert Burger (2014)) controls the distribution and management of all processes.

The goal of the simulated task is to pick up a varying amount of boxes (SLCs) from a table and bring them to a shelf. Although the results in this work have been gathered in simulation, this same task has been executed over 900 times on the real AIMM system at the Automatica trade fair in 2018.

At the beginning of the experiment, the robot only knows its initial position and the positions of the table and the shelf. For deeper investigation, I focus on the execution of the first part of the task after the robot just reached the table. There, it detects the objects and plans collision free plans to grasp the boxes in order to place them onto the mobile base.

Fig. 7.16 shows Gantt charts of the execution of the behavior modeled as a HPFD (top), and the same behavior converted to a CDTN (bottom). On the y-axis, several resources of interest are shown, i.e., the motion planner(s), the the object detector(s) and the manipulator. The planning actions often took multiple seconds as the

[6] gmapping, see `http://wiki.ros.org/gmapping`

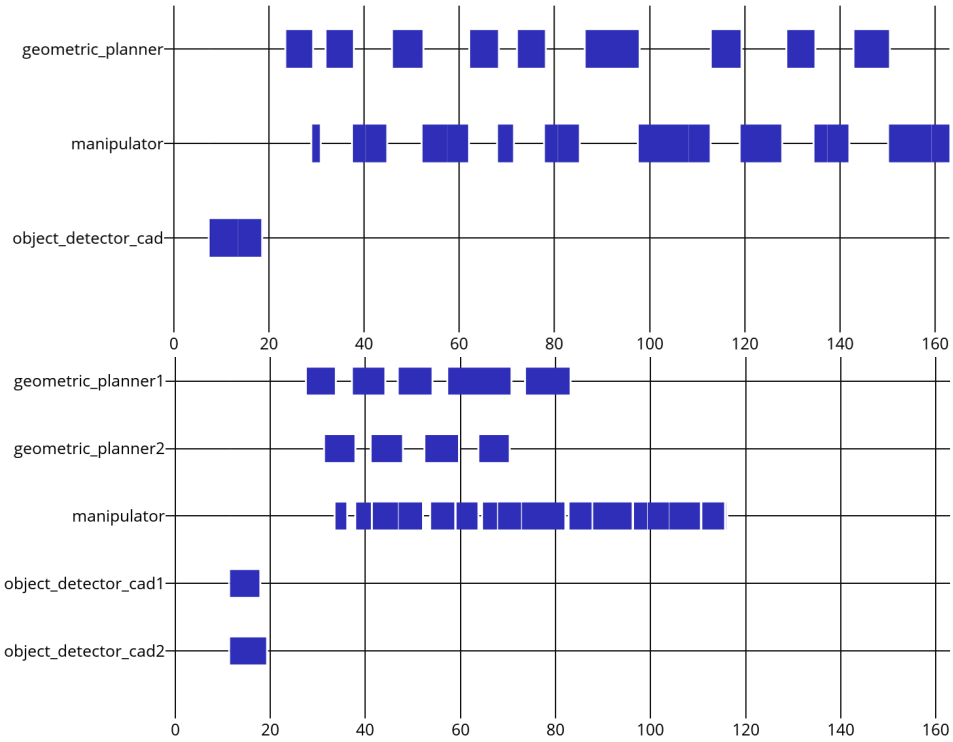


Figure 7.16: *Gantt charts of the execution of the mobile manipulation task (with three boxes to grasp). Top: execution of the HPFD. Bottom: execution of the corresponding CDTN. The x-axis shows the time in seconds and the y-axis the used resources.*

Table 7.13: *Runtime statistics for fetching boxes from the table with HPFDs and the corresponding CDTNs. Execution times are averages of five runs. MP=Motion Planner.*

# boxes	HPFD	CDTN (1 MP)	CDTN (2 MPs)	Improvement (2 MPs)
1	81.3s	73.5s	72.3s	11.1%
2	123.7s	107.3s	97.3s	21.3%
3	164.2s	141.5s	116.5s	29.0%

manipulator movements had to obey Cartesian constraints: The boxes must not be tilted during manipulation as otherwise the content is spilled.

The HPFD execution is purely sequential and there are thus no overlaps between task nodes in the Gantt chart. In the CDTN execution many task nodes are parallelized, such as the object detection calls, the planning tasks themselves and the planning and manipulation tasks. For parallelizing the planning task nodes p, the world model state of p is projected (as described inSec. 6.2.3) and then passed to p.
Table 7.13 shows the execution of different task scenarios by using both HPFDs and CDTNs. Depending on the number of boxes, CDTNs reaches a speedup between

11-29% compared to HPFDs (column marked 'Improvement'). For comparison, the CDTN runs were performed using one or two motion planner (MPs).

7.1.3 A Household Scenario: Table Setting with AIMM

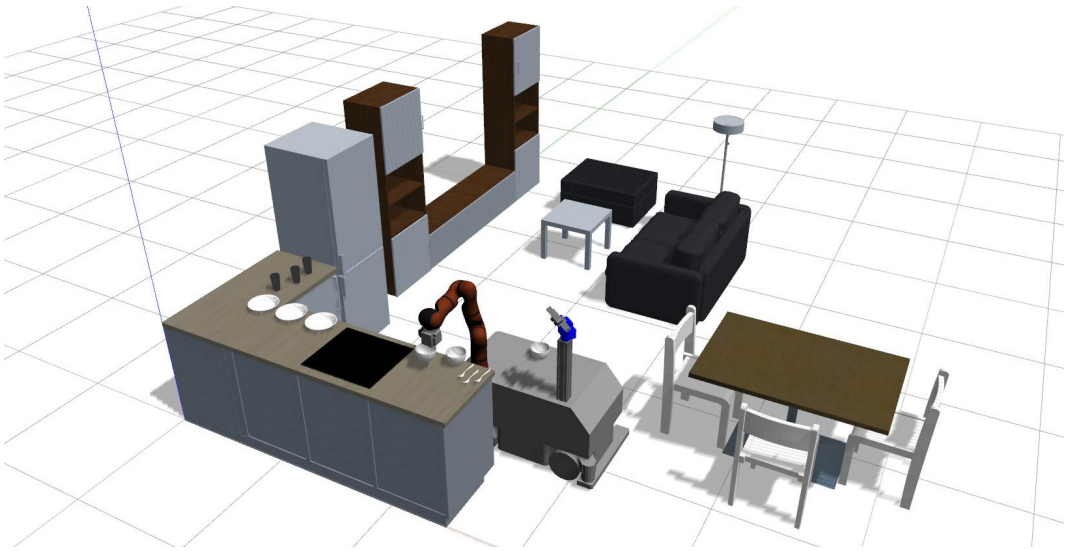


Figure 7.17: *The AIMM platform in the Gazebo simulation environment manipulating various dishes and cutlery.*

In a final validation experiment, the task of AIMM is to set a table in a living room (Fig. 7.17). Various dishes and cutlery have to be manipulated in order to set the table for different amount of persons. Plates, bowls and cups are filled, and the robot thus has to decide if it needs to hold the target objects upright during manipulation (e.g., bowls) or not (e.g., forks).

This experiment only analyzes the performance and applicability of CDTNs. Thus, information about the executed behavior and task analysis is skipped and I directly proceed to the results of applying CDTNs. Table 7.14 shows the sub-task of fetching bowls (B) and forks (F) for two and three persons. It shows that also in this scenario, drastic performance gains of up to 21.7% can be achieved.[7] It furthermore shows that the more un-constraint objects are manipulated (e.g., forks), for which planning times are low anyways, the more the improvement of using CDTNs decreases.

Fig. 7.18 show the Gantt chart of the experiment for fetching the cutlery for three persons. It shows well, that the object detection calls can be parallelized. Further-

[7] In Brunner et al. (2019), I accidentally set the optimization time of the planning parameters to five seconds, which lead to longer planning times and thus to higher improvement rates; I corrected this not-representative parameterization in the course of this work and set it back to one second.

Table 7.14: *Runtime statistics for parts of the table setting scenario of Fig. 7.17 with HPFDs and the corresponding CDTNs. Execution times are averages of five runs. MP=Motion Planner.*

# objects	HPFD	CDTN (2 MPs)	Improvement (2 MPs)
2 B + 2 F	169.2s	132.5s	21.7%
3 B + 3 F	229.4s	203.3s	11.4%

more, the planning actions are executed concurrently (see overlaps between planning actions) and run in parallel with the movement of the manipulator itself. After 143.9 seconds execution time all planning actions are finished. As a consequence the idle time of the manipulator from this time on is drastically reduced.

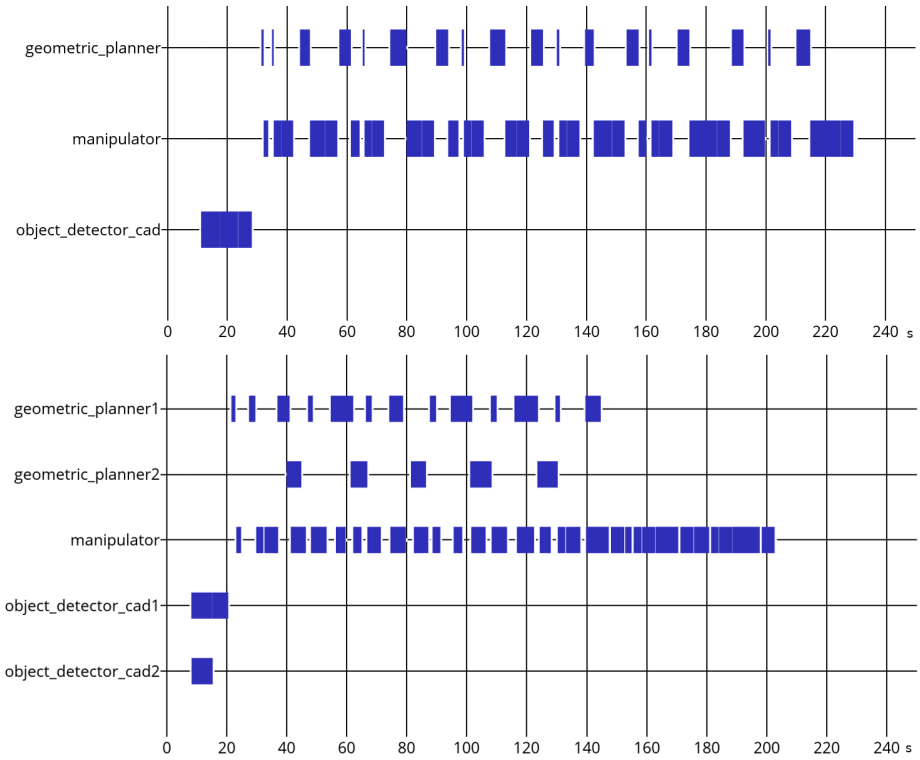


Figure 7.18: *Gantt charts of the execution of the table setting task (with three bowls and three forks). Top: execution of the HPFD. Bottom: execution of the corresponding CDTN. The x-axis shows the time in seconds and the y-axis the different resources used.*

7.2 Discussion

In this section I want to critically analyze the strengths and weaknesses of my approach and compare it to related work most similar to my methodology.

7.2.1 Strengths and Weaknesses of the Proposed Approach

The major strength of my approach is that I tested it thoroughly in various, real and simulated experiments and in several application domains. The possibility to program robotic behavior graphically in 2.5D leads to a high information retrieval rate and to an eminent diagnosability and transparency of the created robotic behavior. The behavior programmed with RAFCON can scale up to complex tasks and can solve them robustly. On top of that, HPFDs are no rigid structures but are able to modify themselves during execution and thus adapt to the needs of the task. The proposed analysis mechanism can generate a vast amount of log data, which can be used to effectively find bottlenecks concerning robustness or speed. Finally, CDTNs, which build on top of plain HPFDs, are able to optimize the behavior's execution speed autonomously.

One drawback of my implementation is that it is not capable of real time execution. Hard time constraints cannot be guaranteed as Python does not allow for deterministic runtime estimation. However, the original motivation behind HPFDs was to be able to control the high level behavior of robots, which in most application fields does not require hard deadlines to be met. Another missing aspect of my approach is the capability to learn new actions on its own, or improve existing ones in terms of extending their functionality. As this is an essential feature in order to reduce the programming overhead and enable the deployment of robots in real environments, this aspect is part of my future work.

When designing a complex system, the designers have to deal with the conflicting concerns of expressiveness, analyzability and implementability (see Eker and Janneck (2012)). When building RAFCON, we tried to find a good compromise between expressive power, analyzability (and hence debugability) and ease of learning and using the proposed graphical DSL. Blackwell (1996) and Curtis et al. (1989) already showed the expressive power of modeling control and data flow graphically. The statements related to programming languages are supported by the research of Myers (1986) and Ford and Tallis (1993), who claim that the human mind is optimized for

vision, and that shapes are easier to process than words. To visualize the benefits of visual programming, I showed many examples for creating programs in HPFD-GL in a clear and concise manner. HPFDs are analyzable both visually and formally. Curtis et al. (1989) showed the analyzability of graphically visualizing control and data flow, and ,furthermore, found out that a flowchart representation was superior to textual pseudo code when the task involved tracing flow of control or data.

Visuals can be easier to reason about for programs that are related to flow control and especially flowcharts help to trace execution flow and localize the area where the bug is located (Brooke and Duncan (1980)). But also on a formal level HPFDs are amendable to several types of analysis including model checking, which can be used for change-impact analysis, verification and bug-finding. The amount of systems, challenges and domains for which robotic experts from various research institutes, universities and companies used RAFCON already shows its usability, popularity and scalability. Its popularity is furthermore shown by the high number of robotic experts in our institute using RAFCON for their robotic systems[8]. Another fact supporting this statement is the high total number of downloads of our software (>25000 downloads)[9].

7.2.2 Limitations of CDTNs

As shown in the validation chapter, CDTNs are able to decrease the execution time of robotic tasks by almost one third. However, like any approach, CDTNs also have their limitations. First and foremost, CDTNs cannot be used in totally sequential tasks as in this case no action can be pre-computed. This holds true for tasks, in which the robot's poses are fully pre-programmed (such as in current wielding tasks of common conveyor belts). The degree of runtime benefits using CDTNs drops with the increasing number of sequential actions per task. As can be seen in Table 7.6, the runtime savings of the CDTN execution drops to less than 10% for the "deploy 4 RUs (l)" and "deploy 4 RUs (r, l, rgt)" case. The main reason for this is that these scenarios include a lot of actions that cannot be parallelized, e.g., the long navigation actions from the lander to the RU deploy poses, the leveling actions that do not include planning, or the long grasping and releasing times (in the case of "deploy 4 RUs (r, l, rgt)" 9.1s per grasp/release).

If a planning system (see Sec. D) is used to decide upon the robot's next action and

[8] https://dlr-rm.github.io/RAFCON/projects
[9] https://pepy.tech/project/rafcon

does only commit to very few future actions at a time, CDTNs cannot be employed as well. Apparently, if there are no actions about which the robot knows that it might have to execute them in future, no pre-calculations can be performed.

Finally, another limitation of CDTNs are behaviors consisting of actions with different runtimes combined with the property that those actions with the high runtimes cannot be pre-computed. E.g., if a robot spends 90% of its time to travel from a to b then optimizing a few manipulation actions does not optimize the whole runtime a lot (see the ROBEX simulation test cases in Table 7.6 with time savings below 10%). Nevertheless, if these 10% of the actions are very time critical (such as acting in a valley with no exposure to the sun, which is the robot's energy supply) using CDTNs will pay off.

7.2.3 Comparison with Closely Related Task Control Frameworks

In the following I will highlight related work that has a similar graphical component as RAFCON or has a similar expressiveness for creating robotic plans. In order to be listed in this differentiation, the robotic plans must have been shown in real world experiments and not purely in simulation.

Concerning high similarity to RAFCON's graphical editor the robot task control frameworks CS::APEX by Buck and Zell (2019), FlexBE by Schillinger et al. (2016), ArmarX Statechart by Wächter et al. (2016), and YARP Behavior Trees by Colledanchise and Natale (2018) have to be listed.
CS::APEX, written in C++, is a powerful tool for creating robotic algorithms especially in the context of computer vision algorithms. It features a VPL to setup processing nodes and to create dafa-flows, which can also be triggered in a synchronous manner. However, it does not support the creation of complex robotic behavior and does not feature dynamic behavior changes during runtime in terms of creating or removing processing nodes on the fly.
FlexBE is a powerful engine for behavior development written in Python, which is based on the state machine framework SMACH and features a graphical editor. It's main drawback compared to RAFCON is that the FlexBE editor does not support Continuous Visual Abstraction (i.e., no zooming) into state hierarchies and the creation of data flows with data type consistency checks. On top of that, FlexBE's execution engine does not offer execution capabilities similar to modern IDEs (such as *Step*, *Step Into*, *Step Out*) and is middle-ware dependent (ROS-only).

The ArmarX Statechart has a very similar state concept than RAFCON and also a

powerful graphical editor. However, it does not support zooming as well, does not offer execution breakpoints and, most notably, does not support concurrencies, which is imperative for implementing more complex behaviors. Additionally, the data flow is bound to state transitions, which overly restricts the possibility of data distribution among the single nodes.
Finally, the YARP Behavior Tree package also offers a powerful graphical editor for robotic behavior. Opposed to RAFCON it is based on Behavior Trees and not on state machines. One drawback is that different semantic actions are represented by the same tree node (for more information about the advantages and disadvantages of behavior tree see Chpt. E). Furthermore, the YARP Behavior Tree package neither supports the modeling of data flows nor execution breakpoints for debugging, only a pausing functionality.

Software frameworks that do not feature a graphical editor but are nevertheless sufficiently relevant for the field because of their expressiveness are CRAM by Beetz et al. (2010), SMACH by Bohren and Cousins (2010), and the Fawkes Behavior Engine by Niemueller et al. (2009).
CRAM is a powerful task control language written in the Lisp language. The DSL is very expressive in terms of formalizing both sequential and concurrent robot behavior. In opposite to RAFCON, CRAM was mainly designed for high expressiveness focusing less on analyzability and implementability. For RAFCON we considered all concerns equally, thus accepting compromises in the field of expressiveness. Like RAFCON, CRAM is able to dynamically change the behavior during runtime. One of its drawbacks is its lack of a both a graphical viewer and a graphical editor to change the robotic behavior. On top of that it is not capable of modeling data flow between actions, which renders the application of CDTNs, as proposed in this work, impossible.
SMACH, as one of the most famous task control packages of ROS, has a broad community of users. It is implemented in Python and features a similar state machine concept to RAFCON. Unfortunately, it does not offer fine-grained execution control and only comes with a graphical viewer, not a graphical editor. Additionally, it does not support the definition of data flows between states but just has a semi-global data pool per HierarchyState.
Finally, the Fawkes Behavior Engine is another feature-rich behavior definition framework written in Lua, which has a small memory footprint and is very efficient. It ships with a graphical viewer for runtime monitoring of the behavior, but does not provide a graphical editor. Its Bibtex-like behavior definition format resembles a DSL, but the framework does not abandon its capability to support general purpose programming. Instead of supporting the explicit modeling of data flow it supports state local and

state machine global data structures, the latter one via a blackboard approach.

Another tool worth mentioning is the SmartMDSD toolchain for designing complex robotic systems using a model-driven development approach. Although targeting at the holistic modeling of whole robotic systems including design choices concerning sensor selection, navigation pipelines, manipulation modules, inter-process communication and behavior creation, it uses similar conceptual approaches as I do in this work. On the one hand, it clearly advocates the separation of data and logic flow as an enabler for modularization, composition and configuration validation, monitoring and failure detection in the large scale. Furthermore, the component model is based on a component-port-connector schema, which is similar to the HPFD model. On top of that, it endorses the importance of a clear Separation of Concerns and separation of roles, which are central aspects of the 2T* architecture and the whole-lifecycle analysis architecture. Compared to the whole-lifecycle analysis architecture, the SmartMDSD toolchain enforces an even stronger separation of roles during the development lifecycle of the robot application. The identified roles include the application domain expert, the robotic expert, the component developer, the system architect, the behavior developer, the quality of service (QoS) engineer, the safety engineer, the system integrator and the end user.
Finally, similar to the differentiation between the descriptive and the operational action model in the context of RATNs, the SmartMDSD toolchain strictly differentiates between the semantic model of a component and its implementation. This leads to a technology agnostic system modeling that is able to reuse proposed components on different systems employing different technologies (such as different middlewares).

Chapter eight

Conclusion

This chapter summarizes my achievements. The strong separation of theory and implementation that has been respected so far is softened in order to be concise. At the end of the chapter some future work concerning the use of HPFDs and CDTNs will be given.

8.1 Summary

This work presents an approach of controlling complex robotic agents in real and simulated domains. In order to differentiate from purely theoretical work focusing on simplified, simulated domains (such as the blocks world domain, see Gupta and Nau (1992)), I focused on the challenges of real world tasks right from the beginning. Using the 2T* architecture definition I highlight the complexity of real systems and clearly separate the responsibilities of various task control modules amongst others procedural task control, belief state modeling, procedural reasoning and semantic task planning.

One of the most essential parts is the procedural task control. In this context, HPFDs and the RAFCON implementation play the central role. Building on HPFD-GL I created a novel graphical editor[1] to implement robotic behavior in an efficient and

[1] in collaboration with my colleagues at the Institute of Robotics and Mechatronics of the German Aerospace Center, see Sec. 4.6

diagnosable manner. It has been shown that a graphical representation yields a promising approach and is superior to purely textual programming in many aspects.

Next to programming the logic flow also the data flow can be explicitly modeled and is the dominant way to distribute data for decision making and action parameterization. The central store for task goals and data describing the robotic environment is the robot's belief state. As a central element the procedural task control (i.e., RAFCON) retrieves data using a powerful query language (CYPHER) and forwards the data as parameterization to various actions. Properly used, it is one of the key concepts to solve the challenge of the explosion of the number of states required to cover all relevant aspects of an autonomous behavior. On top of that, a powerful library and template concept allows for an easy modularization of complex behavior. This also supports the collaboration of whole developer teams onto the same task.

One strong aspect of HPFDs is their event-less design, which is replaced by powerful observer structures. These observers can be implemented elegantly using various types of concurrency patterns.

As HPFDs are based on a state machine formalization, model checking can be performed on the implemented behavior. By using this methodology the absence of dangerous system states can be guaranteed, which leads to increased safety, both for the robot and its environment.

Furthermore, developers of HPFDs can both create the layout of complex behavior and edit the Python code of the atomic actions (i.e., the ExecutionStates) inside RAFCON. Although Python is just one of many high level programming languages it has some essential benefits. It is a reflective meta-programming language (similar to Lisp) that allows for online introspection and dynamic code changes. This is exploited by RAFCON to alter the robotic behavior during runtime by creating or removing states on the fly depending on the robot's belief state. On top of that, during the last few years, Python became one of the two most popular programming languages (see the RedMonk index by O'Grady (2020), the PYPL index by Carbonnelle (2020), and the TIOBE index by TIOBE Software Quality Company (2020)).

Another essential aspect of RAFCON is its powerful execution engine. It can be used to control the behavior execution in a very fine grained manner such as pausing, stopping, resuming, and stepping into, over or out of state hierarchies. Another feature, which is commonly known from IDEs is that the behavior supports breakpoints to some extent. On top of that, the execution explicitly handles state abortion and preemption as first order objects, to which the subsequent execution can react appropriately. There are no limits (except for the computer's main memory and computation power) in the number of concurrently executed states, as every state

has its own execution thread.

During behavior execution, great amounts of precious log data can be accumulated. The log data does not only include the data flow information but also the state's semantics (i.e the action's descriptive model), which is based on ontologies. Thus, RAFCON enables HPFD developers to develop the operational and descriptive action models inside the same tool. The log data and the state semantics enable a holistic view on task programming and analysis. This analysis enables the efficient localization of robustness and performance bottlenecks and is hence a necessary step towards sustainable task development.

One of the central messages of this work is that instead of fixing performance bottlenecks manually, this can be done automatically. The developer's main task is to create a clear and expressive behavior description while not obfuscating the layout in order to achieve a higher performance. Otherwise, the clarity of the task will decrease during its optimization, which will eventually lead to errors (The famous proverb "Early optimization is the source of all evil" holds true in this case, as the behavior is under constant modification, even by itself).

The approach postulated by my thesis is to divide and conquer, i.e., first create correct behavior, annotate it with rich semantics and then have it automatically optimized. This is performed by creating CDTNs based on HPFDs and the bottleneck information. The subsequent optimization based on CDTNs leverages pre-computation and parallelization to achieve performance improvements of almost 30 %. Implementation-wise, CDTNs could easily be integrated into RAFCON as the framework provides a powerful plugin mechanism, which enables CDTNs to exchange even one of the most central elements: the execution engine.

I implemented and analyzed the listed features in various domains (i.e., space, industry and household) on different real and simulated mobile robots. Furthermore, I showed that HPFDs scale up to complex behaviors of more than 5000 states and more than 7000 transitions. These experiments show the scalability of HPFD-GL and the RAFCON implementation. Both the Continuous Visual Abstraction and the modular approach of HPFDs lead to increased controllability and analyzability of behavior, which can be highly complex and concurrent. The use of RAFCON in various domains stresses its generality and applicability.

Next to the complex real world experiments the positive response of the robotic community towards RAFCON validates my approach. With more than 25000 downloads[2] it is one of the fastest growing open source task control frameworks in the community.

[2] see https://pepy.tech/project/rafcon

8.2 Future Work

At the time of writing I see three major possibilities for improvement. First, more work could be put into the graphical programming interface of HPFDs. Possibilities to add semantic colors or logos to various states inside the graphical editor could help building up the mental model of the state machine. Another subject for research would be to analyze whether real 3D information is even superior to 2.5D as implemented in RAFCON. The ability to have more semantic layers of an action, which could be inspected by changing the viewpoint inside a 3D scene, is an interesting idea. On top of that, alignment features (with the help of grids and formatting tools for alignment) could improve the effectiveness of information retrieval even further. Enforcing special, semantic arrangements of actions in order to visualize concurrencies, sequences, monitors or decision structures could increase the expressiveness of the HPFDs, especially if a team of developers collaborate on the same behavior. Certainly, meaningful user studies have to be carried out to foster the scientific significance of such approaches, especially in terms of intuitiveness. Classical user studies in dedicated user study facilities are hard to perform as it takes several hours to days to master all concepts of RAFCON, even for skilled programmers. Thus, community based approaches via online surveys seem to be the more promising approach.

The second area for future work is the improvement of the semantic logging architecture in order to generate even more powerful logs. This could be used to learn classifiers for special events or errors autonomously. These generated modules could then be inserted into the behavior in order to monitor various conditions and to make predictions towards the outcome of various actions or sub-tasks.

Finally, a huge challenge of future AI research is to try to break up the hierarchical layered system architecture approach of, e.g., the three or two layered architecture (and even the 2T* architecture). A promising approach would be to perform semantic planning on every abstraction layer and not only on the highest level above the procedural behaviors. One challenge is to integrate the planners into the various architecture levels, which might not be achieved easily as the required domain knowledge of the respective level has to be modeled sufficiently well in symbols that a planner can understand. Another challenge is to create planners, that are capable of planning not only sequential plans (such as Fast Downward Helmert (2006)) but also reactive or concurrent plans. Current state of the art planners, such as contingency planners (see Gaschler et al. (2013)) or the Madagascar planner (see Rintanen (2014)), go in the right direction but are not capable enough yet and can either not handle reactive patterns or concurrencies or both.

Appendix A

Appendix 1 - List of Equations

List of Equations

Appendix B

Appendix 2 - CDTN Dependency Graph Snapshots

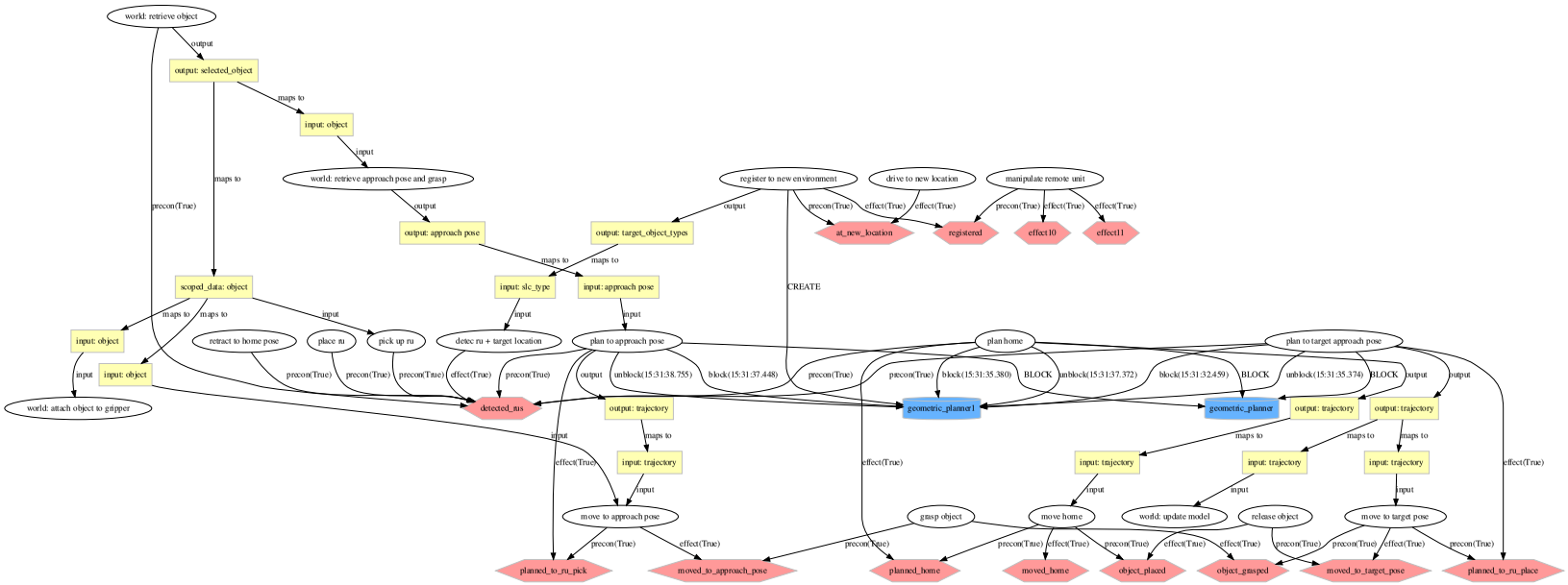


Figure B.1: *A dependency graph snapshot taken during execution of the ROBEX example state machine of Sec. 6.4. Data dependencies are visualized in yellow, world model dependencies in red and resource dependencies in blue. The blocking and unblocking of resources are tagged with timestamps. The labels clarify how two entities relate to each other, e.g.,* input *means that a state has the data dependency as input,* precon(x) *means that a state requires a precondition on the world model fact* x*. For information about the implementation see Sec. 6.4.*

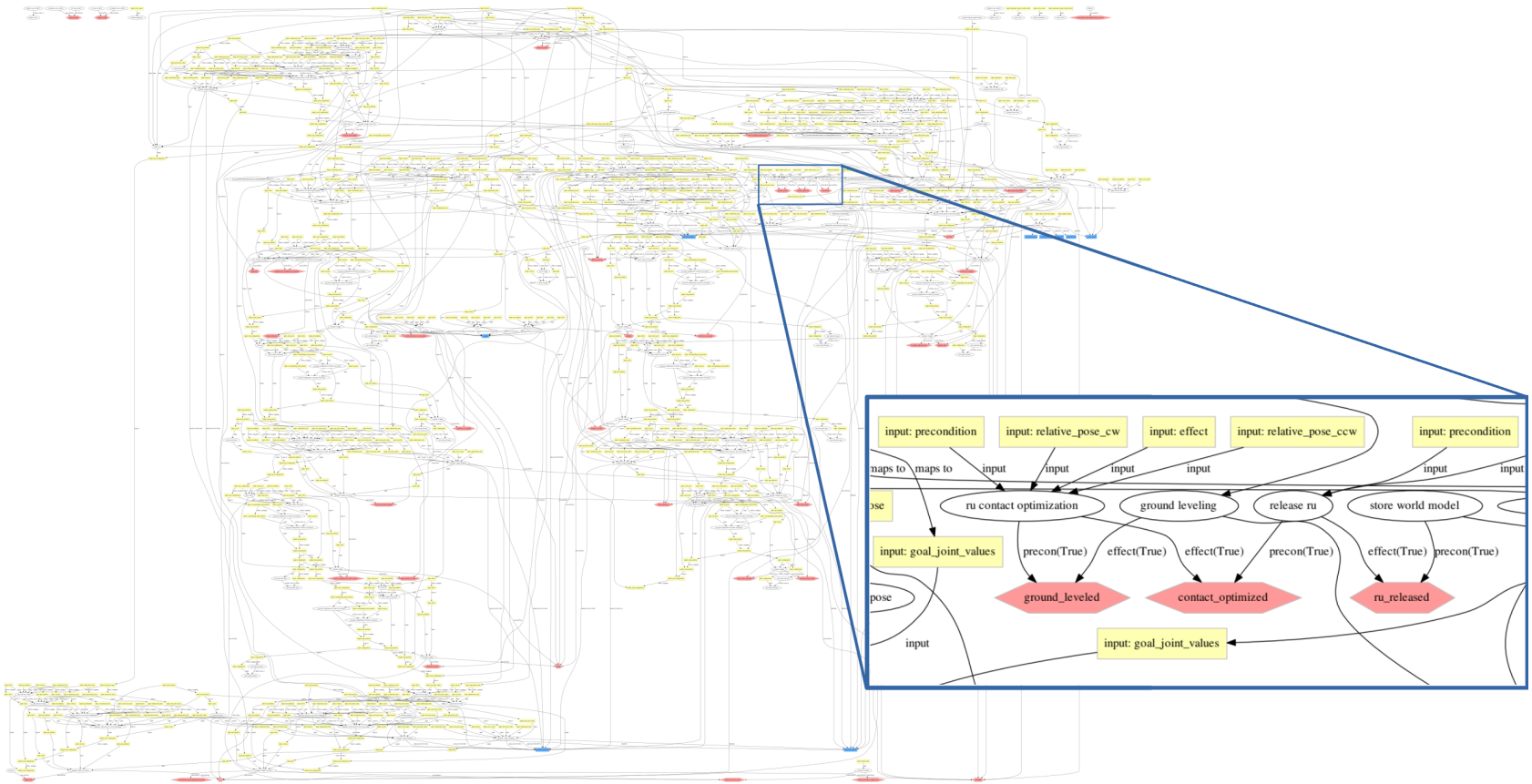


Figure B.2: *Example CDTN dependency graph of the RU deployment task of the LRU Gazebo simulation (presented in Sec. 7.1.1). The semantics of the nodes and colors are the same as for Fig. B.1. The manual zoom shows the three world model dependencies "ground_leveled", "contact_optimized" and "ru_released" including their creating and depending actions.*

Figure B.3: *Example CDTN dependency graph of the simulated ROBEX ASM mission presented in Sec. 7.1.1. The semantics of the nodes and colors are the same as for Fig. B.1. The manual zoom shows the resource "ground analyzer", on which the "analyzed ground" action depends and which was blocked by that action already two times. The timestamp of the blocking can be seen next to blocking-call.*

Appendix C

Appendix 2 - Gantt Charts

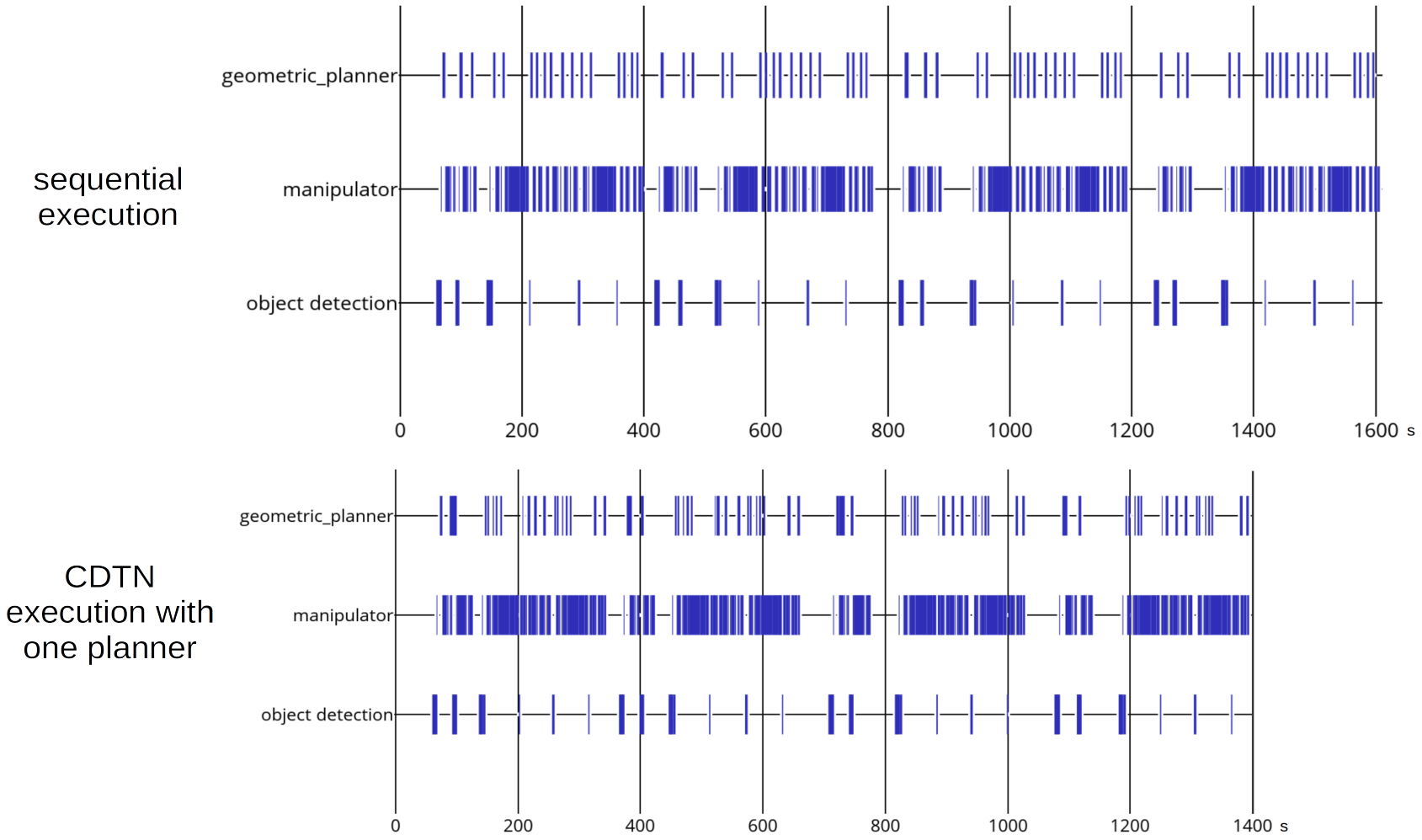


Figure C.1: *Gantt charts for the "deploy 4 RUs (r, l)" scenario, as described in Sec. 7.1.1. The top shows the sequential mission execution, the lower one the execution using CDTNs with one motion planner.*

Appendix D

Appendix 3 - Semantic Planning

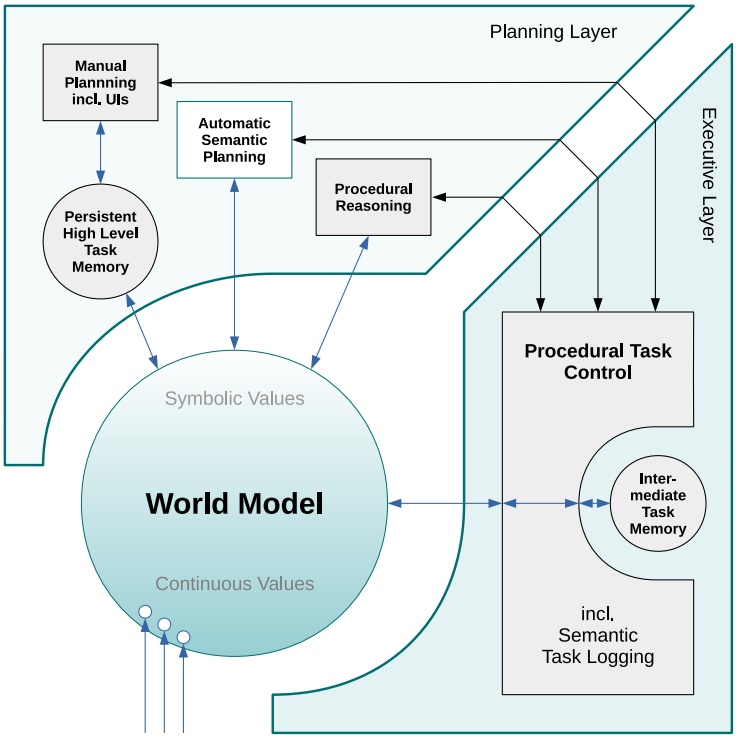


Figure D.1: *The software layers of the decision tier of the 2T* architecture with the automated planning module highlighted in white.*

Although the focus of this work is the executive layer, I invested a significant amount of work in integrating it with the other layers of the proposed 2T* architecture. Therefore, one goal of this work is to integrate automated semantic planners (see Fig. D.1) in order to generate HPFDs automatically.

D.1 Related Work

There exist a vast amount of related work in the field of semantic planning, often also called automated planning (see Nau et al. (2004)) or task planning (see Cambon et al. (2009)).

The basic idea behind semantic planning is straight forward: given a domain specification (which includes all possible actions a agent can perform), a start configuration and a goal configuration, a planner tries to find a plan that leads an agent from the start to the goal configuration. For example, in a maze environment, the plan the agent receives will guide him from his starting position to the exit of the maze.

The planners in literature differ on the one hand by their objectives. Most of the planners simply try to find a sequential plan, which an agent can execute in order to reach a goal (see Helmert (2006), Hoffmann (2001) or Nau et al. (2003)). Next to these classical planners there exist also contingency planners (e.g., the KVP planner by Gaschler et al. (2013)), which generate plans with contingencies, i.e., conditions. These contingencies allow an agent to react to uncertainties without re-planning, which can induce a significant runtime deterioration. Finally, there exist planners which generate plans consisting of a partial order (in opposite to the total order of sequential plans). One famous example is the Madagascar planner by Rintanen (2014). The resulting plans allow the agent to parallelize the plans' actions in order to gain execution speedups.

Furthermore, planners differ by their mathematical approach to search through the problem domain. The most famous approach is heuristic state space search, which is used by Fast Downward by Helmert (2006), Shop2 by Nau et al. (2003) or HSP2 by Bonet and Geffner (2001). This family of planners employ and improve classical search strategies such as Breadth First Search (BFS), A* or other search algorithm based on heuristics, which are the key aspect of such planners. The second type of planners employ SAT solvers. Famous examples are SATPlan by Kautz and Selman (1996) or Madagascar by Rintanen (2014). The third type of planning approaches is called symbolic planners. These planners try to group states into sets and try to process the sets directly. They often rely on binary decision diagrams (BDDs) (see Bryant (1986)), which can efficiently represent vast problem spaces. Classical examples are SymBA* by Torralba et al. (2016), Gamer by Kissmann and Edelkamp (2011) and SYMPLE by Speck et al. (2018).

D.2 Semantic Planning Using RATNs

Writing a planner is not in the scope of this work. However, interfacing planning systems capable of finding plans in changing environments under complex constraints is an elementary part of my architecture.

Reactive plans can be programmed effectively using HPFDs. Yet, if the task variability is high, pre-programmed procedures reach their limits as they are either too inefficient, if they are programmed in a too general manner, or require too much manual work to get optimized versions of specific task configurations.

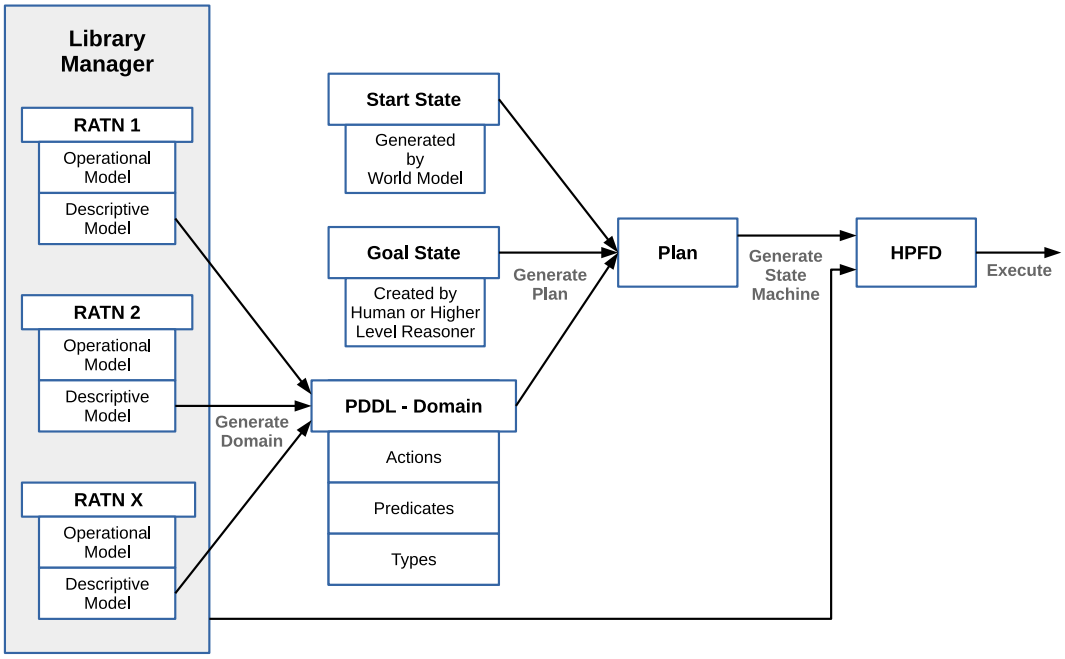


Figure D.2: *Pipeline for generating HPFDs using semantic planning.*

The pipeline to generate HPFDs using a semantic planner is shown in Fig. D.2. The library manager (or skill manager) manages all the skills of the robot as RATNs, i.e., HPFD states and their descriptive action model including all preconditions and effects. The type system has to be predefined manually, which includes the definition of the inheritance structure of all objects in the PDDL domain (i.e., object taxonomy). Based on this information, a PDDL domain is generated, consisting of a type system, predicates and actions. The predicates and actions can be directly inferred by analyzing the RATNs including their preconditions and effects. Thus, their PDDL representation can be generated automatically. By using this domain, the start state (as given by the world model) and the goal state, a plan is generated. This plan facilitates the robot to reach the goal state. Finally, a HPFD is generated based on the

plan which can be executed subsequently.

D.3 Implementation

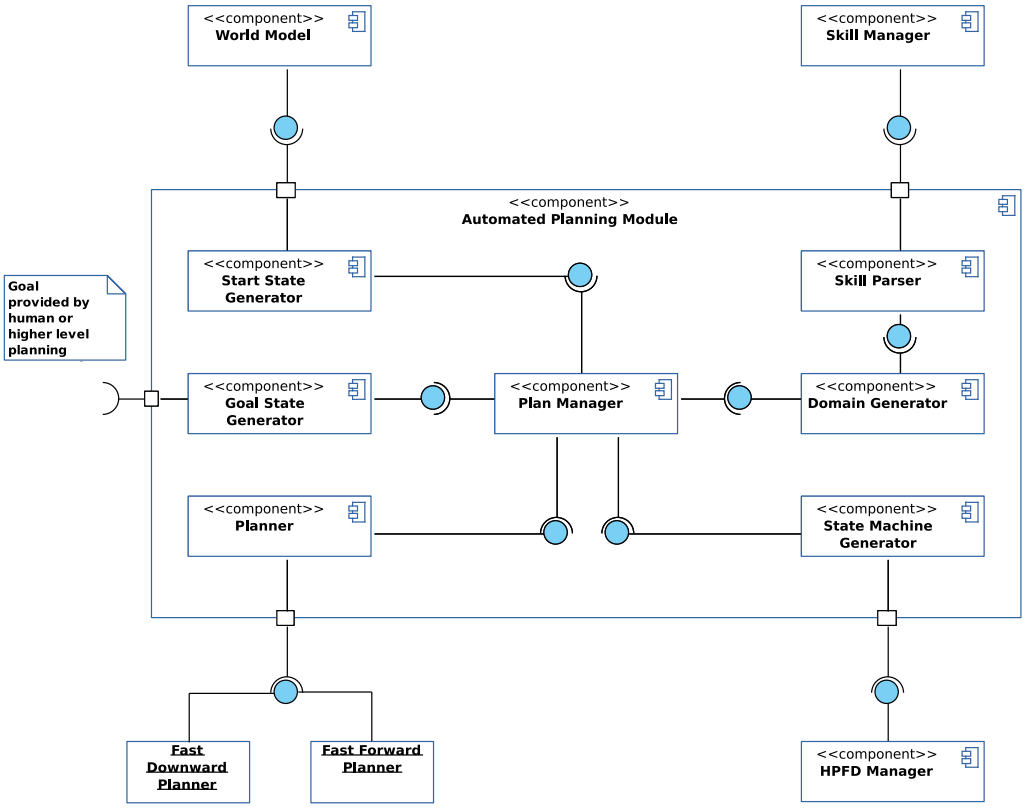


Figure D.3: *Component diagram showing RAFCON's semantic planning integration.*

The implementation was performed by using the RAFCON plugin interface[1]. We currently do not directly generate the PDDL description based on the RATN dependencies but reserve a dedicated entry in the semantic editor for the PDDL action definition. Of course, this is redundant and will be treated in a future release. The plugin adds an editor for defining the PDDL representation of a RATN. The editor supports the direct manipulation of the action descriptions and supports the user by automatically inferring all necessary types, predicates and PDDL requirements.

All components of the plugin (apart from the PDDL editor) are shown in Fig. D.3. The *Skill Manager* and *Skill Parser* are responsible for managing all RATNs in the robot's action pool, i.e., the set of actions which are accessible by the planner and can

[1]The plugin can be found at https://github.com/DLR-RM/rafcon-task-planner-plugin

be used in order to find a suitable plan. The *Domain Generator* takes all PDDL action descriptions and generates all predicates needed by the actions. Furthermore, it takes the manually defined type system and compiles everything into the PDDL domain. The world model information is used as input for the *Start State Generator*. A human can also also write the initial PDDL world state manually, which is currently the only officially supported method in the current plugin release. The *Planner* component enables the access to various planning systems. We have currently integrated the *Fast Downward Planner* and the *Fast Forward Planner*, which have different advantages and support different heuristics. The Planner component is written in a modular way, allowing other developers to easily integrate other PDDL planners. The *Plan Manager* uses the functionality of all components in order to generate a plan that satisfies the goal constraints and finally generates the HPFD using the *State Machine Generator*.

The presented task planning plugin benefits from the RAFCON framework in several ways. First of all, it can use and enhance RAFCON's library manager, which can be used to organize the skills of the target robot(s) in a consistent and clear manner. Furthermore, it can use RAFCON's monitoring and execution capabilities (i.e., pausing, resuming, stepping, break points), which is useful for runtime introspection and debugging. Finally, RAFCON can be used to store the resulting plans for future use in dedicated state machines, which in turn can also be included in more complex behaviors.

D.4 Experiments

In the experiments by Sürig (2018), we automatically generated a plan for the ROBEX scenario sketched in Fig. 4.12. Listing D.1 shows an excerpt of the PDDL domain used for planning. The excerpt shows all necessary types, predicates and actions (however only one action implementation for the sake of conciseness).

Listing D.1: *PDDL domain for the seismic network mission of the ROBEX scenario (see Fig. 4.12)*

```
(define (domain robex_domain)
 (:requirements :adl :strips :typing :equality)
 (:types
      Physobj Location - Object
      Dockable Storage - Physobj
      Vehicle Stationary - Storage
```

```
      RemoteUnit - Dockable
      Rover - Vehicle
      Lander - Stationary
 )
 (:predicates
      (at ?obj - Physobj ?loc - Location)
      (in ?obj - Physobj ?store - Storage)
      (is-grasping ?veh - Vehicle)
      (is-grasped ?dock - Dockable ?veh - Vehicle)
      (is-full ?store - Storage)
      (located ?veh - Vehicle ?store - Storage)
      (is-leveled ?loc - Location)
      (contact-optimized ?dock - Dockable)
      (deployed ?dock - Dockable ?loc - Location)
 )
 (:action navigate ...)
 (:action grasp-from-storage
    :parameters(?veh - Vehicle ?store - Storage
                ?dock - Dockable ?loc - Location)
    :precondition(and (not (is-grasping ?veh))
                      (in ?dock ?store)
                      (at ?veh ?loc)
                      (at ?store ?loc)
                      (located ?veh ?store))
    :effect (and (not (in ?dock ?store))
                 (not (is-full ?store))
                 (is-grasping ?veh)
                 (is-grasped ?dock ?veh)
                 (not(contact-optimized ?dock))))
 (:action load-into ...)
 (:action place-at ...)
 (:action locate ...)
 (:action level ...)
 (:action optimize-contact ...)
 (:action deploy ...)
)
```

Fig. D.4 shows a generated state machine for the ROBEX domain. For presentation reasons the goal was only to deploy one seismometer instead of four, as otherwise there would be too many states in the state machine and the labels could not be read in paper format. The initial state and goal for the presented plan are shown in listing D.2.

Listing D.2: *Initial and goal state for the seismic network mission of the ROBEX scenario (see Fig. 4.12)*

```
(define (problem ROBEX_Seismograph_grid)
    (:domain robex_domain)
    (:objects
        l1 l2 l3 l4 l5 - Location
        u1 u2 u3 u4 - RemoteUnit
        lru - Rover
        lan - Lander
    )
    (:init
        (at lru l5)
        (at lan l5)
        (in u1 lan)
        (in u2 lan)
        (in u3 lan)
        (in u4 lan)
    )
    (:goal
        (and
        (deployed u1 l1)
        (at lru l5)
        )
    )
)
```

For the planner integration into the RAFCON framework we performed different experiments (see Sürig (2018)). Next to analyzing different advantages and disadvantages of various planning systems, we investigated in important planning metrics such as planning time, plan length and mission time. In this section I just provide a proof of concept experiment in order to show the applicability of the RAFCON planning plugin. Although the plugin was not used yet in a real robotic application (and thus also not in the validation experiments) the planning plugin is going to be integrated in future use cases especially in tasks containing high action variability.

D.5 Drawbacks

Especially in the face of changing environments and high task variability an autonomous system can benefit from the application of semantic planners. However,

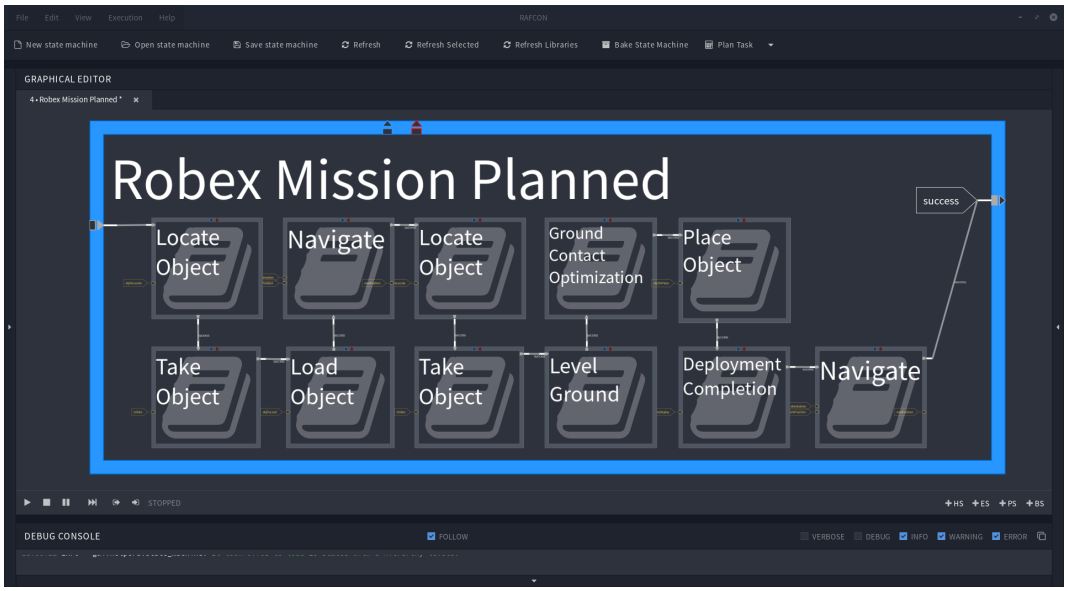


Figure D.4: *Automatically generated HPFD using a semantic planner.*

there are several limitations that restrict their use.

- Using a planner always leads to uncertainties concerning the planning success, the plan structure and the planning time.
- Non-optimal plans are often inferior to manual designed plans as they do a lot of unnecessary steps, which can lead to significant performance overhead (see Sürig (2018)).
- The total mission execution time is increased by planning delays.
- Many planners only support sequences and only some support contingencies or concurrencies. To the best of my knowledge there does not exist any planner able to plan partially ordered plans with contingencies. This would be required in order to generate full-featured HPFDs.
- Planners cannot be used if the problem space is too big (state explosion problem). Thus, people often rely on manually designed procedures implementing well-established, heuristic approaches (see Schuster et al. (2017), Beetz et al. (2010), Jentzsch et al. (2013)).

Finally, in most of the real scenarios, we relied on directly using behavior defined in HPFD-GL as often (especially on the higher level) the optimal course of action can easily be defined manually.

Appendix E

Appendix 4 - Comparing HPFD-GL with Behavior Trees

The following gives an overview of how Behavior Trees can be programmed using HPFD-GL. Behavior trees are not state-based such as finite state machines, but rely on periodically triggering all nodes of the tree (e.g., in a 10Hz cycle) until all nodes have returned success or an unhandled failure occurs (see Marzinotto et al. (2014)).

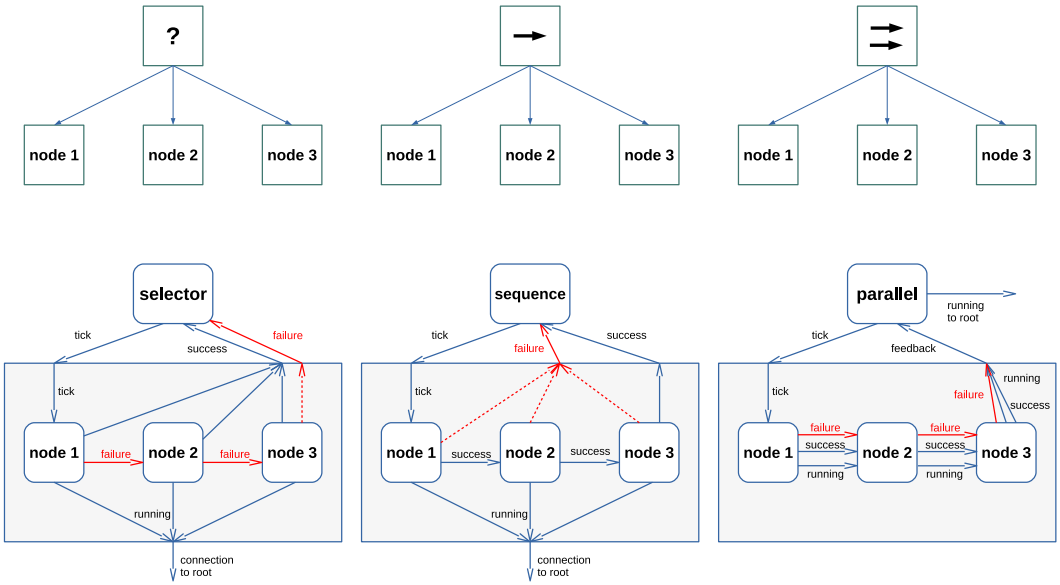


Figure E.1: *Top: the graph grammar for Behavior Trees; Bottom: the functional equivalences of the Behavior Tree building blocks created with HPFD-GL.*

Fig. E.1 shows the graph grammar of the major Behavior Tree building blocks (as defined by Marzinotto et al. (2014)). These are: *selectors*, *sequences* and *concurrencies*. Selectors are control blocks that trigger their child nodes subsequently until one of them returns *Success*. It returns *Failure* if all children return Failure. If the child returns *Running*, the selector simply forwards the signal to the root. The selector can be efficiently used to program conditions, fallback behavior and error recovery. Sequences are similar to selectors, however they only return Success if all child states have returned Success. If one of the children terminates because of a Failure, the sequence returns a Failure. As the name suggests, sequences are used to program sequences of actions. Concurrencies trigger all their children at the same time. This happens until either more than S children succeed or more than F children fail. S and F are both parameters of the parallel node.

For the sake of brevity, I skipped the *root state*, *conditions*, *actions* and *decorators* in Fig. E.1. The root state is trivial, as it just triggers the ticks for the Behavior Tree. Actions (also having the default state values Success, Failure or Running) can simply be mapped to HPFD ExecutionStates. The conditions and decorators can be built using the other node types: actions can be used to model conditions (as actions can have several outcomes and are thus a super-set of conditions) and the combination of a condition and a sequence can be used to model decorators (that basically just wraps a condition around their only child action).

The HPFD-GL representation of the behavior-tree control nodes executes the same behavior (see Fig. E.1). The main logic of the single control flow strategies resides in the structure of the transitions and implementation of the control nodes, which are implemented using ExecutionStates and Python. While the selector and sequence ExecutionState performs simple forwarding of the state execution values Success, Failure and Running, the Parallel state is more sophisticated: On its first execution when traversing the Behavior Tree it directly executes all its children. On the second execution (when the control flow comes back originating at “node3”) it has to check the state values of all children. If more than S children have succeed, then it returns Success, if more than F child state have failed it returns Failure. Otherwise, it just reports Running to the root state.

For the sake of completeness I have to mention the node extension by Marzinotto et al. (2014) as well: the * and $\tilde{}$ extension. The stateful * extension of the selector and sequence node as proposed by Marzinotto et al. (2014) can be reproduced using RAFCON’s scoped and local variables straight forward. The aim of the *Decorator* $\tilde{}$ extension is to synchronize several agents, which can also be achieved by using a Sequence with a synchronization action as first child, only allowing the execution of the second child when the synchronization was successful.

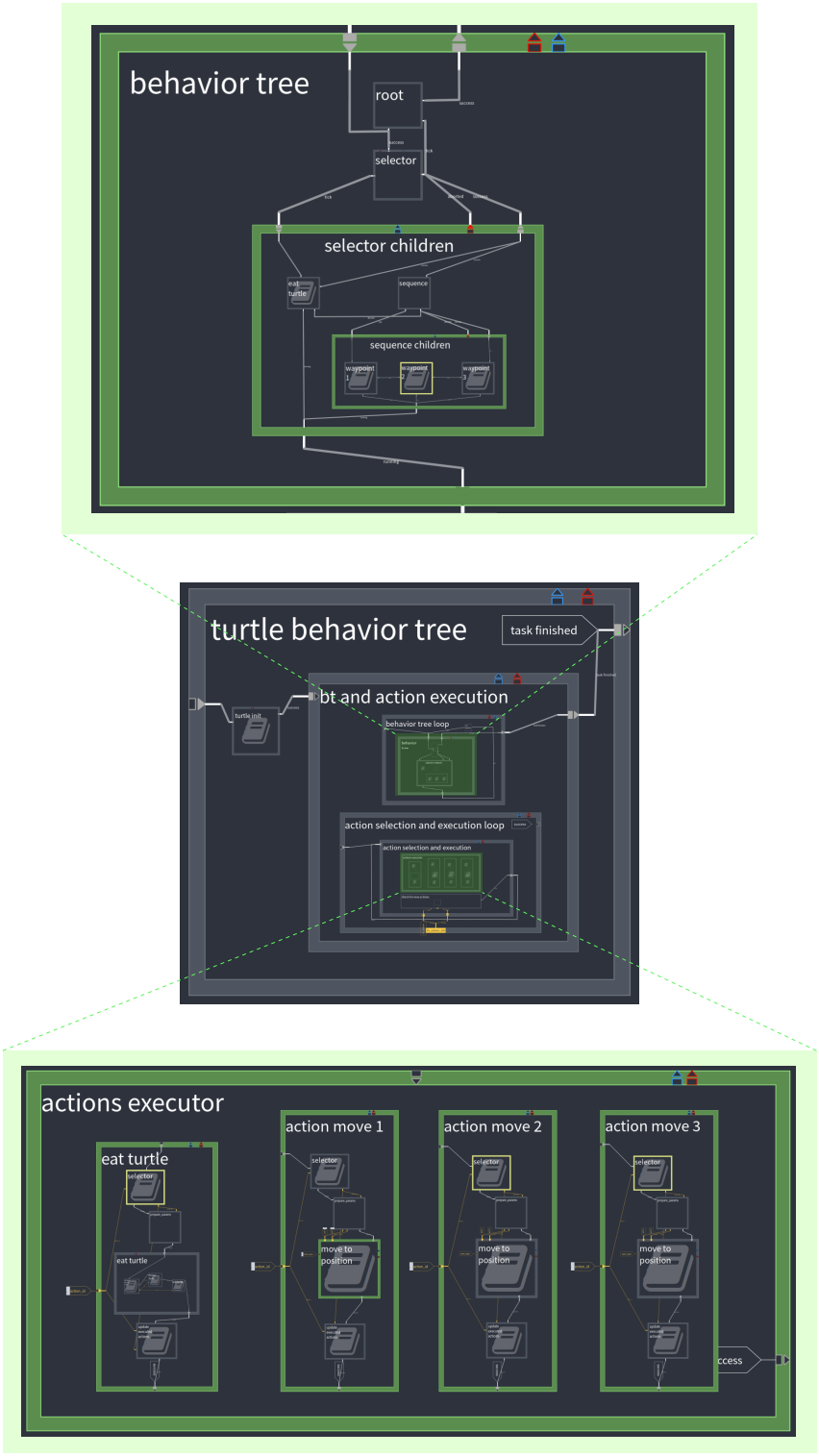


Figure E.2: *An example Behavior Tree built in RAFCON to control the reactive behavior of a turtle in ROS's turtlesim simulator.*

Fig. E.2 shows an example Behavior Tree programmed in RAFCON. The top part shows the actual Behavior Tree. The tree only retrieves the current state of the actions and selects which action to execute. The lower part visualizes a *concurrent action selector*, which performs the appropriate actions based on the behavior-tree's selection. The concurrent action selector listens to the selection and preempts the current actions if a new action has to be triggered.

The Behavior Tree executes a simple behavior for a turtle of ROS's well known turtle simulation (see Quigley et al. (2009)). The agent under control is a turtle that is hungry and eats other turtles (of different colors and shape) when they are close to the turtle. If no other turtles are there, the agent moves to different spots on the map. The example Behavior Tree just consists of the root state, a selector and a sequence. The selector takes care, that the turtle eats other appearing turtles. If no turtle is nearby the sequence is executed, which commands the turtle to move to three different waypoints. One run of the behavior is visualized in Fig. E.3. The top left shows the initial state of the map with the main agent in the middle. The second image in the top row shows another turtle while the main turtle is currently heading to its second waypoint. The agent cancels its movements and moves toward the new turtle. In the third image the main turtle has just eaten the other turtle. The fourth image shows the main turtle on its way to its third waypoint at the moment when another turtles appears. In the fifth image the agent has just eaten the second turtle and heads to a third waypoint (image six).

In the RAFCON implementation the visual tree structure of the behavior-trees is preserved. HPFD-GL do not need additional constructs to represent the control flow of Behavior Trees. The modeling of the Behavior Trees using HPFD-GL in RAFCON has several advantages and disadvantages.

Advantages:

In opposite to Behavior Trees, HPFDs use the visual representation not only for modeling the composition of nodes and parent-child relationships. Also, control and data flow can be modeled by the HPFD-GL representation directly instead of hiding them in the BT control building blocks.

HPFD-GL can directly be used to create the behavior-tree control nodes. This includes hierarchical abstraction, the control flow modeling via transitions and the use of outcomes for the different return values *Success*, *Running* and *Failure*. Further, the execution engine can be used to control a Behavior Tree in a fine grained manner, by pausing, resuming, stepping and using breakpoints. The colorization of the states based on the state's execution state increases the analyzability during debugging.

By using HPFD-GL, new or custom control structures could be added easily, which

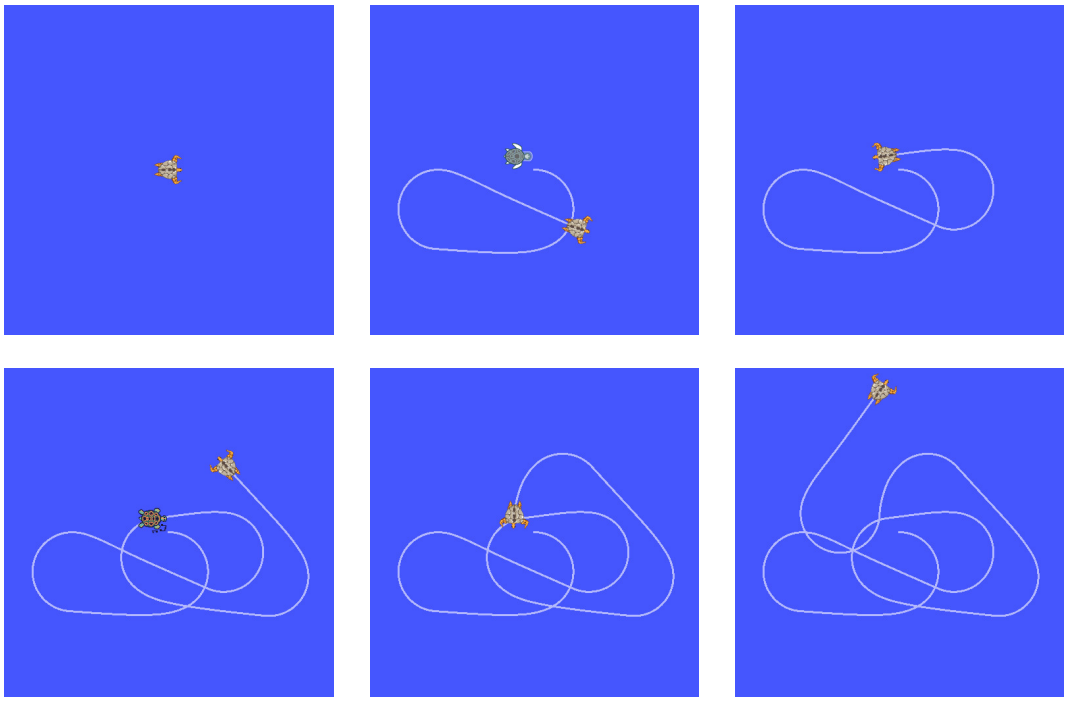


Figure E.3: *Example of a RAFCON Behavior Tree execution using ROS's turtlesim simulator (see Quigley et al. (2009)). The single images are screenshots of the map in which the main agent (the turtle with the red hands and feet) traverses the map and eats other turtles as it is hungry. The image order is: top left to right, then bottom left to right*

are partly self-explaining as the control flow is modeled explicitly. Behavior-trees hide the control flow of the children using abstract symbols such as the question mark or the arrow. New node types would need new symbols, which enhances the background knowledge a developer needs the have when using them.

Finally, RAFCON can be used to not only model Behavior Trees but also default HPFD constructs, which have different advantages, especially concerning the modeling of data flow.

Disadvantages:

In HPFD-GL, hierarchies have to be created visually by setting up a scope for the children. For Behavior Trees, single connections between two nodes define a parent-child relationship.

HPFD-GL programs need more transitions (approximately three times the number, as can be seen in the graph grammar of Fig. E.1 or the Behavior Tree example of Fig. E.2). Thus, the representation of behavior-trees is more concise and needs less modeling.

The disadvantages could be solved by enhancing tool support in RAFCON. This would

include to automatically insert or delete transitions if another child state is added. This is done in a similar manner already when dealing with ConcurrencyStates (see Sec. 4.6). Furthermore, when inserting a new Selector state, a new HierarchyState could automatically be created, in which the Selector's children are going to be placed.

In summary, Behavior Trees can be modeled using HPFD-GL. The existing execution semantics are fully sufficient, no additional state types are needed. This renders HPFD-GL at least as powerful as Behavior Trees concerning the control flow. Concerning data flow, classical Behavior Trees do not offer any features. The data modeling needs to be done by the traditional way of using the Behavior Tree's host language. More principal disadvantages of Behavior Trees are the iterative (high frequency) condition checks of the whole tree, which can be very expensive (see Colledanchise and Ögren (2014)). Furthermore, most of the time the conditions are checked in vain, as the system or task state did not change but the checks are performed in a fixed frequency. Finally, all actions need to offer a "tick-interface", which allows for a step-wise execution of their functionality.

Acronyms

2T* *Enhanced Two-Tiered Architecture* 34, 36–39, 41, 42, 45, 46, 96, 101, 124, 221, 224, 233

AI *Artificial Intelligence* 29

ARCHES *Autonomous Robotic Networks to Help Modern Societies* 1, 3, 8, 45, 195, 196

AIMM *Advanced Industrial Mobile Manipulator* 21, 23, 87, 120, 124, 148, 153, 154, 204–206, 208, 211, 212, 214

ARM *Advanced RISC Machine* 49

ASM *Active Seismic Measurement* 186, 187, 189, 193, 194, 197, 198, 230

CDTN *Concurrent Dataflow Task Network* 17, 112, 161–165, 167, 169, 172–178, 180–183, 196, 199–201, 204, 211–213, 218, 221

CI *Continuous Integration* 52, 148

CPU *Central Processing Units* 36, 49, 164, 182

DLR *German Aerospace Center* 6, 123, 125, 148, 204

DOF *Degrees of Freedom* 6, 27, 48, 211

DSL *Domain Specific Language* 4, 23, 66, 76, 114, 219

EKF *Extended Kalman Filter* 49, 50

eSATA *external Serial AT Attachment* 49

ESI *Earth Similarity Index* 2

FPGA *Field Programmable Gate Arrays* 36, 49, 50, 63

FST *Finite State Transducers* 71, 75

GNSS *Global Navigation Satellite System* 50

GPU *Graphics Processing Units* 21, 36, 195

HAL *Hardware Abstraction Layer* 37

HPFD *Hierarchical, Parallel, Finite State Machine with Data Flows* viii, 12, 13, 17, 19, 23, 67–79, 84, 85, 96, 99, 101, 102, 107–113, 115, 117–122, 124, 132, 134, 135, 139, 140, 145, 149–152, 160–164, 167, 172–175, 177, 199, 204, 207, 212–217, 220–224, 233, 235, 237, 240, 242, 244, 245

HPFD-GL *the HPFD Graphical Language* 13, 76–79, 81, 83–85, 87–96, 99, 100, 113, 115, 118, 183, 217, 221, 223, 240–242, 244–246

IAI *Institute for Artificial Intelligence* 7

IDE *Integrated Development Environment* 64–66, 140

IMU *Inertial Measurement Unit* 49, 50

IRCP-SYA-Bars *Inverted Resource Capability Profile with Shifted-Y-Axis using Bar plots* 142, 193

LIDAR *Light Detection and Ranging* 36, 204, 211

LRU *Lightweight Rover Unit* 4–7, 9, 10, 14, 19–21, 23, 27, 45–50, 88, 90, 97, 104–106, 123, 124, 132, 148, 150, 171, 183–186, 188, 195–198, 204, 208, 209, 229

NASA the *National Aeronautics and Space Administration* 1, 157

NEEMs *Narrative-Enabled Episodic Memories* 152

PCDU *Power Control and Distribution Unit* 36, 49

PCIe *PCI Express* 49

PDDL *Planning Domain Definition Language* 17, 107, 109, 161, 169, 235–237

PIA *Potentially Irreversible Action* 166, 170, 175, 177

RAFCON *RMC Advanced Flow Control* vi, 4, 13, 14, 17–21, 69, 72, 76, 108, 113–120, 122–124

RATN *Resource-Aware Task Node* 17, 23, 108, 109, 111–113, 118, 122, 130, 132, 141, 143, 147, 148, 150, 152, 161–165, 167, 168, 172, 173, 175, 178–180, 201, 220, 235, 236

RCP *Resource Capability Profile* 141

RMC *Robotics and Mechatronic Center* 6

ROBEX *Robotic Exploration of Extreme Environments* iii, v–ix, 3–6, 8, 19, 20, 23, 45, 88, 89, 96, 100, 104, 120, 121, 124, 131, 132, 141, 142, 146, 147, 154, 181, 183–186, 188–190, 192–195, 197–200, 205, 208, 209, 228, 230, 237–239

ROS *Robot Operating System* 50, 51, 113, 122, 156, 212, 218, 219, 243–245

RRT *Rapidly-Exploring Random Tree* 50

RU *Remote Unit* 88–90, 94–96, 100, 196–198, 200, 217, 229

SLAM *Simultaneous Localization and Mapping* 50

TCP *Tool Center Point* 38, 42, 88, 98

UAV *Unmanned Aerial Vehicles* 125

USB *Universal Serial Bus* 49

VP *Visual Programming* 63–66

VPL *Visual Programming Languages* 63–66, 119, 218

Bibliography

A. Albu-Schäffer, C. Ott, and G. Hirzinger. A Unified Passivity-based Control Framework for Position, Torque and Impedance Control of Flexible Joint Robots. *The International Journal of Robotics Research*, 26(1):23–39, 2007. doi: 10.1177/0278364907073776.

S. Alexandrova, Z. Tatlock, and M. Cakmak. RoboFlow: A Flow-Based Visual Programming Language for Mobile Manipulation Tasks. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 5537–5544. IEEE, 2015.

T. Allard. ONR's Warfighter Performance Department: Making Investments in Cognitive Science and AI. In *The Mystery of Artificial Intelligence*, 2014.

R. H. Andersen, T. Solund, and J. Hallam. Definition and Initial Case-Based Evaluation of Hardware-Independent Robot Skills for Industrial Robotic Co-Workers. In *41st International Symposium on Robotics (ISR)*, pages 1–7. VDE, 2014.

J. R. Anderson. ACT: A Simple Theory of Complex Cognition. *American Psychologist*, 51(4):355, 1996.

C. Archibald and E. Petriu. A Model for Skills-Oriented Robot Programming (SKORP). In *Optical Engineering and Photonics in Aerospace Sensing*, pages 392–402. International Society for Optics and Photonics, 1993.

C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT press, 2008.

J.-C. Baillie. Urbi: Towards a Universal Robotic Low-Level Programming Language. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 820–825. IEEE, 2005.

J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, and J. Weiser. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *Computer Aided Verification (CAV)*, volume 8044 of *LNCS*, pages 863–868. Springer, 2013.

J. Barreiro, M. Boyce, M. Do, J. Frank, M. Iatauro, T. Kichkaylo, P. Morris, J. Ong, E. Remolina, T. Smith, et al. EUROPA: A Platform for AI Planning, Scheduling, Constraint Programming, and Optimization. In *22nd International Conference on Automated Planning and Scheduling (ICAPS)*, 2012.

M. Beetz. *Concurrent Reactive Plans: Anticipating and Forestalling Execution Failures*. Springer-Verlag, Berlin, Heidelberg, 2000. ISBN 3-540-67241-9.

M. Beetz, L. Mösenlechner, and M. Tenorth. CRAM – A Cognitive Robot Abstract Machine for Everyday Manipulation in Human Environments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1012–1017, Taipei, Taiwan, October 18-22 2010. doi: 10.1109/IROS.2010.5650146.

M. Beetz, U. Klank, I. Kresse, A. Maldonado, L. Mösenlechner, D. Pangercic, T. Ruhr, and M. Tenorth. Robotic Roommates Making Pancakes. In *IEEE International Conference on Humanoid Robots (Humanoids)*, pages 529–536. IEEE, 2011.

M. Beetz, M. Tenorth, and J. Winkler. Open-EASE. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1983–1990. IEEE, 2015.

M. Beetz, D. Beßler, A. Haidu, M. Pomarlan, A. K. Bozcuoğlu, and G. Bartels. Know Rob 2.0—A 2nd Generation Knowledge Processing Framework for Cognition-Enabled Robotic Agents. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 512–519. IEEE, 2018.

R. Bitter, T. Mohiuddin, and M. Nawrocki. *LabView: Advanced Programming Techniques, Second Edition*. CRC Press, 2007. ISBN 9781420004915.

A. F. Blackwell. Metacognitive Theories of Visual Programming: What do we think we are doing? In *IEEE Symposium on Visual Languages*, pages 240–246, 1996. doi: 10.1109/VL.1996.545293.

S. Blumenthal, H. Bruyninckx, W. Nowak, and E. Prassler. A Scene Graph Based Shared 3D World Model for Robotic Applications. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 453–460. IEEE, 2013.

J. Bohren and S. Cousins. The SMACH High-Level Executive [ROS news]. *IEEE Robotics & Automation Magazine (RAM)*, 4(17):18–20, 2010.

J. Bohren, R. B. Rusu, E. G. Jones, E. Marder-Eppstein, C. Pantofaru, M. Wise, L. Mösenlechner, W. Meeussen, and S. Holzer. Towards Autonomous Robotic Butlers: Lessons Learned with the PR2. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 5568–5575, 2011.

B. Bonet and H. Geffner. Heuristic Search Planner 2.0. *AI Magazine*, 22(3):77, 2001. doi: 10.1609/aimag.v22i3.1576.

T. Booth and S. Stumpf. End-User Experiences of Visual and Textual Programming Environments for Arduino. In *International Symposium on End User Development*, pages 25–39. Springer, 2013.

J. Bresson, C. Agon, and G. Assayag. Visual Lisp/CLOS Programming in Openmusic. *Higher-Order and Symbolic Computation*, 22(1):81–111, 2009.

J. Brooke and K. Duncan. Experimental Studies of Flowchart Use at Different Stages of Program Debugging. *Ergonomics*, 23(11):1057–1091, 1980.

R. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal on Robotics and Automation*, 2(1):14–23, 1986.

J. C. Browne, S. I. Hyder, J. Dongarra, K. Moore, and P. Newton. Visual Programming and Debugging for Parallel Computing. *IEEE Parallel & Distributed Technology: Systems & Applications*, 3(1):75–83, 1995.

S. G. Brunner, F. Steinmetz, R. Belder, and A. Dömel. RAFCON: a Graphical Tool for Task Programming and Mission Control. In *20th RoboCup International Symposium*, 2016a.

S. G. Brunner, F. Steinmetz, R. Belder, and A. Dömel. RAFCON: A Graphical Tool for Engineering Complex, Robotic Tasks. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016b.

S. G. Brunner, P. Lehner, M. J. Schuster, S. Riedel, R. Belder, A. Wedler, D. Leidner, M. Beetz, and F. Stulp. Design, Execution, and Postmortem Analysis of Prolonged Autonomous Robot Operations. *IEEE Robotics and Automation Letters (RA-L)*, 3(2): 1056–1063, 2018.

S. G. Brunner, A. Dömel, P. Lehner, M. Beetz, and F. Stulp. Autonomous Parallelization of Resource-Aware Robotic Task Nodes. *IEEE Robotics and Automation Letters (RA-L)*, 2019.

R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986. doi: 10.1109/TC.1986.1676819.

S. Buck and A. Zell. CS::APEX: A Framework for Algorithm Prototyping and Experimentation with Robotic Systems. *Journal of Intelligent & Robotic Systems*, 94(2): 371–387, 2019. ISSN 1573-0409. doi: 10.1007/s10846-018-0831-7.

D. S. Burnett. The Genesis Solar Wind Sample Return Mission: Past, Present, and Future. *Meteoritics & Planetary Science*, 48(12):2351–2370, 2013. doi: 10.1111/maps.12241.

M. M. Burnett. Visual Programming. In *Wiley Encyclopedia of Electrical and Electronics Engineering*. American Cancer Society, 1999. ISBN 9780471346081. doi: 10.1002/047134608X.W1707.

M. M. Burnett, M. J. Baker, C. Bohus, P. Carlson, S. Yang, and P. Van Zee. Scaling up Visual Programming Languages. *Computer*, 28(3):45–54, 1995. ISSN 0018-9162. doi: 10.1109/2.366157.

M. Büttner. Robot Flow Control - Development of a Modular, Graphical User Interface. Master Thesis, Université Toulouse-III-Paul-Sabatier, 2015.

S. Cambon, R. Alami, and F. Gravot. A Hybrid Approach to Intricate Motion, Manipulation and Task Planning. *The International Journal of Robotics Research*, 28(1):104–126, 2009. doi: 10.1177/0278364908097884.

P. Carbonnelle. PYPL Index. Internet, 2020. URL `http://pypl.github.io/PYPL.html`. Accessed: 2020-04-25.

M. Cashmore, M. Fox, D. Long, D. Magazzeni, B. Ridder, A. Carrera, N. Palomeras, N. Hurtos, and M. Carreras. ROSPlan: Planning in the Robot Operating System. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2015.

S. Chien, R. Knight, A. Stechert, R. Sherwood, and G. Rabideau. Integrated Planning and Execution for Autonomous Spacecraft. In *IEEE Aerospace Conference*, volume 1, pages 263–271, 1999. doi: 10.1109/AERO.1999.794242.

M. M. Cohen. First Mars Habitat Architecture. In *AIAA SPACE Conference and Exposition*, page 4517, 2015.

M. Colledanchise and L. Natale. Improving the Parallel Execution of Behavior Trees. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7103–7110, 2018. doi: 10.1109/IROS.2018.8593504.

M. Colledanchise and P. Ögren. How Behavior Trees Modularize Robustness and Safety in Hybrid Systems. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1482–1488. IEEE, 2014.

R. Cooper and T. Shallice. Soar and the Case for Unified Theories of Cognition. *Cognition*, 55(2):115–149, 1995. ISSN 0010-0277. doi: https://doi.org/10.1016/0010-0277(94)00644-Z.

J. Craig. *Introduction To Robotics: Mechanics And Control (Third Edition)*. Pearson Education, 2009. ISBN 9788131718360.

B. Curtis, S. B. Sheppard, E. Kruesi-Bailey, J. Bailey, and D. A. Boehm-Davis. Experimental Evaluation of Software Documentation Formats. *Journal of Systems and Software*, 9(2):167–207, 1989.

S. Dennis, L. Alex, L. Matthias, and S. Christian. The SmartMDSD Toolchain: An Integrated MDSD Workflow and Integrated Development Environment (IDE) for Robotics Software. *JOSER - Journal of Software Engineering for Robotics*, 2016.

J. Diaz-Herrera and R. Flude. Pascal/HSD: A Graphical Programming System. In *IEEE Proceedings COMSAC*, pages 723–728, 1980.

R. Diestel. *Graph Theory (Fourth Edition)*, volume 173 of *Graduate Texts in Mathematics*. Springer-Verlag Berlin Heidelberg, 2012. ISBN 978-3-642-14278-9.

A. Dömel, S. Kriegel, M. Kassecker, M. Brucker, T. Bodenmüller, and M. Suppa. Toward Fully Autonomous Mobile Manipulation for Industrial Environments. *International Journal of Advanced Robotic Systems*, 14(4), 2017.

D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks. Software Architecture Themes in JPL's Mission Data System. In *IEEE Aerospace Conference*, volume 7, pages 259–268, 2000. doi: 10.1109/AERO.2000.879293.

R. T. Effinger, B. C. Williams, and A. G. Hofmann. Dynamic Execution of Temporally and Spatially Flexible Reactive Programs. In *Workshops at the Twenty-Fourth AAAI Conference on Artificial Intelligence*, page 121, 2010.

J. Eker and J. W. Janneck. Dataflow Programming in CAL-Balancing Expressiveness, Analyzability, and Implementability. In *Conference Record of the Forty Sixth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, pages 1120–1124. IEEE, 2012.

J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull. Graphviz – Open Source Graph Drawing Tools. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing: 9th International Symposium*, pages 483–484. Springer, 2002. ISBN 978-3-540-45848-7. doi: 10.1007/3-540-45848-4_57.

H. Elmqvist, F. Gaucher, S. E. Matsson, and F. Dupont. State Machines in Modelica. In *9th International MODELICA Conference*, pages 37–46. Linköping University Electronic Press, 2012.

J. S. Falcon and M. Trimborn. Graphical Programming for Field Programmable Gate Arrays: Applications in Control and Mechatronics. In *American Control Conference*, pages 7–9. IEEE, 2006.

E. Fernandez, L. S. Crespo, A. Mahtani, and A. Martinez. *Learning ROS for Robotics Programming*. Packt Publishing Ltd, 2015.

R. E. Fikes and N. J. Nilsson. Strips: A new Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3):189–208, 1971. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(71)90010-5.

R. J. Firby. The RAP Language Manual. *Animate Agent Project Working Note AAP-6, University of Chicago*, 1995.

Florian Schmidt and Robert Burger. How we Deal with Software Complexity in Robotics: 'links and nodes' and the 'robotkernel'. Technical report, DLR, 2014.

B. Foing. Towards a Moon Village: Vision and Opportunities. In *EGU General Assembly Conference Abstracts*, volume 18, 2016.

G. Folino, A. A. Shah, and N. Kransnogor. On the Storage, Management and Analysis of (Multi) Similarity for Large Scale Protein Structure Datasets in the Grid. In *22nd IEEE International Symposium on Computer-Based Medical Systems*, pages 1–8, 2009. doi: 10.1109/CBMS.2009.5255328.

L. Ford and D. Tallis. Interacting Visual Abstractions of Programs. In *IEEE Symposium on Visual Languages*, pages 93–97, 1993. doi: 10.1109/VL.1993.269584.

M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010. ISBN 0321712943, 9780321712943.

G. C. Fox, R. D. Williams, and G. C. Messina. *Parallel Computing Works!* Elsevier, 2014.

N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An Evolving Query Language for Property Graphs. In *International Conference on Management of Data*, SIGMOD '18, pages 1433–1445. ACM, 2018. ISBN 978-1-4503-4703-7. doi: 10.1145/3183713.3190657.

J. Frank and A. Jónsson. Constraint-Based Attribute and Interval Planning. *Constraints*, 8(4):339–364, 2003. ISSN 1572-9354. doi: 10.1023/A:1025842019552.

A. Gaschler, R. P. Petrick, T. Kröger, A. Knoll, and O. Khatib. Robot Task Planning with Contingencies for Run-Time Sensing. In *IEEE International Conference on Robotics and Automation (ICRA), Workshop on Combining Task and Motion Planning*, volume 308, 2013.

E. Gat, R. P. Bonnasso, R. Murphy, et al. On Three-Layer Architectures. *Artificial intelligence and mobile robots*, 195:210, 1998.

D. F. Glas, T. Kanda, and H. Ishiguro. Human-Robot Interaction Design Using Interaction Composer - Eight Years of Lessons Learned. In *11th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pages 303–310. IEEE, 2016.

G. Gousios and D. Spinellis. Java Performance Evaluation Using External Instrumentation. In *Panhellenic Conference on Informatics*, pages 173–177, 2008. doi: 10.1109/PCI.2008.14.

T. R. Green, M. Petre, and R. Bellamy. Comprehensibility of Visual and Textual Programs: A Test of Superlativism Against the 'Match-Mismatch' Conjecture. In *Empirical Studies of Programmers: Fourth Workshop*, volume 121146. Ablex Publishing, Norwood, NJ, 1991.

J. P. Grotzinger et al. Mars Science Laboratory Mission and Science Investigation. *Space Science Reviews*, 170(1):5–56, 2012. ISSN 1572-9672.

N. Gupta and D. S. Nau. On the Complexity of Blocks-World Planning. *Artificial Intelligence*, 56(2-3):223–254, 1992.

S. Haddadin, B. Belder, and A. Albu-Schaeffer. Reactive Motion Generation for Robots in Dynamic Environments. In *IFAC, World Congress*, 2011.

Hai Nguyen, M. Ciocarlie, Kaijen Hsiao, and C. Kemp. ROS Commander (ROSCo): Behavior Creation for Home Robots. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 467–474, 2013.

K. J. Hammond. Explaining and Repairing Plans that Fail. *Artificial Intelligence*, 45 (1-2):173–228, 1990.

D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

M. Harrison and M. Prentiss. *Learning the Pandas Library: Python Tools for Data Munging, Analysis, and Visual*. CreateSpace Independent Publishing Platform, 1st edition, 2016. ISBN 153359824X, 9781533598240.

S. Hart, P. Dinh, J. D. Yamokoski, B. Wightman, and N. Radford. Robot Task Commander: A Framework and IDE for Robot Application Development. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1547–1554. IEEE, 2014.

G. Heald, J. McKean, and R. Pizzo. Low Frequency Radio Astronomy and the LOFAR Observatory. *Springer International Publishing, doi*, 10:978–981, 2018.

M. Hellerer, M. J. Schuster, and R. Lichtenheldt. Software-in-the-Loop Simulation of a Planetary Rover. In *The International Symposium on Artificial Intelligence, Robotics and Automation in Space*, 2016.

M. Helmert. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

Helmholtz Center Future Topic Projects. The Arches Project. Internet, 2019. URL `https://www.arches-projekt.de/helmholtz-zukunftsprojekt-arches/`. Accessed: 2020-04-25.

F. E. Hernandez and F. R. Ortega. Eberos GML2D: A Graphical Domain-Specific Language for Modeling 2D Video Games. In *10th Workshop on Domain-Specific Modeling*, pages 1–1. Citeseer, 2010.

J. Hoffmann. FF: The Fast-Forward Planning System. *AI Magazine*, 22(3):57, 2001.

T. Hofmann, T. Niemueller, J. Claßen, and G. Lakemeyer. Continual Planning in Golog. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

J. Huang and M. Cakmak. Supporting Mental Model Accuracy in Trigger-Action Programming. In *ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 215–225. ACM, 2015.

W. L. Hürsch and C. V. Lopes. Separation of Concerns. Technical report, Northeastern University, 1995.

V. Indrawati, A. Prayitno, and T. Kusuma Ardi. Waypoint Navigation of AR.Drone Quadrotor Using Fuzzy Logic Controller. *TELKOMNIKA Indonesian Journal of Electrical Engineering*, 13(3):381–391, 2015.

F. Ingrand and M. Ghallab. Deliberation for Autonomous Robots: A Survey. *Artificial Intelligence*, 247:10–44, 2017.

F. Ingrand, S. Lacroix, S. Lemai-Chenevier, and F. Py. Decisional Autonomy of Planetary Rovers. *Journal of Field Robotics*, 24(7):559–580, 2007. doi: 10.1002/rob.20206.

F. F. Ingrand, R. Chatila, R. Alami, and F. Robert. PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots. In *IEEE International Conference on Robotics and Automation*, volume 1, pages 43–49. IEEE, 1996.

J. B. Dennis. Data Flow Supercomputers. *Computer*, 13(11):48–56, 1980. ISSN 0018-9162. doi: 10.1109/MC.1980.1653418.

S. Jentzsch, S. Riedel, S. Denz, and S. G. Brunner. TUMsBendingUnits from TU Munich: RoboCup 2012 Logistics League Champion. In *RoboCup 2012: Robot Soccer World Cup XVI*, volume 7500 of *Lecture Notes in Computer Science*, pages 48–58. Springer Berlin Heidelberg, 2013.

G. L. S. Jr. and R. P. Gabriel. The Evolution of Lisp. In *History of Programming Languages Conference (HOPL-II)*, pages 231–270, 1993. doi: 10.1145/154766.155373.

H. Kautz and B. Selman. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1194–1201, 1996.

G. Kazhoyan and M. Beetz. Specializing Underdetermined Action Descriptions Through Plan Projection. *arXiv preprint arXiv:1812.08224*, 2018.

G. Kazhoyan, A. Niedzwiecki, and M. Beetz. Towards Plan Transformations for Real-World Pick and Place Tasks. *arXiv preprint arXiv:1812.08226*, 2018.

J. F. Kelly. *Lego Mindstorms NXT-G Programming Guide*. Apress, 2010.

A. King. The Future of Agriculture. *Nature*, 544(7651):S21–S23, 2017. doi: https://doi.org/10.1038/544S21a10.1038/544S21a.

D. Kirchner, S. Niemczyk, and K. Geihs. RoSHA: A Multi-Robot Self-Healing Architecture. In *Robot Soccer World Cup*, pages 304–315. Springer, 2013.

P. Kissmann and S. Edelkamp. Gamer, a General Game Playing Agent. *KI - Künstliche Intelligenz*, 25(1):49–52, 2011. ISSN 1610-1987. doi: 10.1007/s13218-010-0078-3.

N. Koenig and A. Howard. Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2149–2154, 2004.

K. Konolige. COLBERT: A Language for Reactive Control in Sapphira. In G. Brewka, C. Habel, and B. Nebel, editors, *KI-97: Advances in Artificial Intelligence*, pages 31–52. Springer Berlin Heidelberg, 1997. ISBN 978-3-540-69582-0.

I. Kruijff-Korbayová, G. Athanasopoulos, A. Beck, P. Cosi, H. Cuayáhuitl, T. Dekens, V. Enescu, A. Hiolle, B. Kiefer, H. Sahli, et al. An Event-Based Conversational System for the Nao Robot. In *Proceedings of the Paralinguistic Information and its Integration in Spoken Dialogue Systems Workshop*, pages 125–132. Springer, 2011.

V. Kulkarni and S. Reddy. Separation of Concerns in Model-Driven Development. *IEEE software*, 20(5):64–69, 2003.

J. Laird, A. Newell, and P. Rosenbloom. SOAR: An Architecture for General Intelligence. *Artificial Intelligence*, 33:1–77, 1987. doi: 10.1016/0004-3702(87)90050-6.

J. E. Laird. *The Soar Cognitive Architecture*. MIT press, 2012.

J. E. Laird, K. R. Kinkade, S. Mohan, and J. Z. Xu. Cognitive Robotics Using the Soar Cognitive Architecture. *Cognitive Robotics AAAI Technical Report*, 6:46–54, 2012.

L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

G. Landis, T. Kerslake, D. Scheiman, and P. Jenkins. Mars Solar Power. In *2nd International Energy Conversion Engineering Conference*, page 5555, 2004.

J. H. Larkin and H. A. Simon. Why a Diagram is (Sometimes) Worth Ten Thousand Words. *Cognitive Science*, 11(1):65–100, 1987. ISSN 0364-0213. doi: https://doi.org/10.1016/S0364-0213(87)80026-5.

P. Lehner and A. Albu-Schäffer. The Repetition Roadmap for Repetitive Constrained Motion Planning. *IEEE Robotics and Automation Letters (RA-L)*, 3(4):3884–3891, 2018.

P. Lehner, S. G. Brunner, A. Dömel, H. Gmeiner, S. Riedel, B. Vodermayer, and A. Wedler. Mobile Manipulation for Planetary Exploration. In *Aerospace Conference*. IEEE, 2018.

H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *The Journal of Logic Programming*, 31(1-3):59–83, 1997.

J. Li, A. Xu, and G. Dudek. Graphical State Space Programming: A Visual Programming Paradigm for Robot Task Specification. In *IEEE International Conference on Robotics and Automation*, pages 4846–4853. IEEE, 2011.

R. Li, R. E. Arvidson, K. Di, M. Golombek, J. Guinn, A. Johnson, M. Maimone, L. H. Matthies, M. Malin, T. Parker, et al. Opportunity Rover Localization and

Topographic Mapping at the Landing Site of Meridiani Planum, Mars. *Journal of Geophysical Research: Planets*, 112(E2), 2007.

M. Loetzsch, M. Risler, and M. Jungel. XABSL - A Pragmatic Approach to Behavior Engineering. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5124–5129, 2006.

M. Margitus, G. Tauer, and M. Sudit. RDF Versus Attributed Graphs: The War for the Best Graph Representation. In *18th International Conference on Information Fusion (Fusion)*, pages 200–206, 2015.

R. Martin, J. Rabaey, A. Chandrakasan, and B. Nikolic. *Agile Software Development: Principles, Patterns, and Practices*. Alan Apt series. Pearson Education, 2003. ISBN 9780135974445.

A. Marzinotto, M. Colledanchise, C. Smith, and P. Ögren. Towards a Unified Behavior Trees Framework for Robot Control. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 5420–5427. IEEE, 2014.

Mathworks. Matlab Stateflow. Internet, 2019. URL `https://www.mathworks.com/help/stateflow/`. Accessed: 2020-04-25.

MATLAB. *version 9.4 (R2018a)*. The MathWorks Inc., 2018.

D. McDermott. A Reactive Plan Language. Technical report, Research Report YALEU/DCS/RR-864, Yale University, 1991.

D. Mcdermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL - The Planning Domain Definition Language. Technical Report TR-98-003, Yale Center for Computational Vision and Control, 1998.

C. McGann, F. Py, K. Rajan, H. Thomas, R. Henthorn, and R. McEwen. A Deliberative Architecture for AUV Control. In *IEEE International Conference on Robotics and Automation*, pages 1049–1054, 2008. doi: 10.1109/ROBOT.2008.4543343.

A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, and J. Timmis. Automatic Property Checking of Robotic Applications. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3869–3876, 2017. doi: 10.1109/IROS.2017.8206238.

A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, J. Timmis, and J. Woodcock. RoboChart: Modelling and Verification of the Functional Behaviour of Robotic Applications. *Software & Systems Modeling*, 18(5):3097–3149, 2019. ISSN 1619-1374. doi: 10.1007/s10270-018-00710-z.

L. Mösenlechner. *The Cognitive Robot Abstract Machine: A Framework for Cognitive Robotics*. PhD thesis, TU Munich, 2016.

S. L. Murchie, D. T. Britt, and C. M. Pieters. The Value of Phobos Sample Return. *Planetary and Space Science*, 102:176–182, 2014. ISSN 0032-0633. doi: https://doi.org/10.1016/j.pss.2014.04.014. Phobos.

N. Muscettola. HSTS: Integrating Planning and Scheduling. Technical report, DTIC Document, 1993.

B. A. Myers. Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '86, page 59–66. Association for Computing Machinery, 1986. ISBN 0897911806. doi: 10.1145/22627.22349.

NASA. Open MCT. Internet, 2017. URL `https://nasa.github.io/openmct/`. Accessed: 2020-04-25.

D. Nau, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.

D. Nau, M. Ghallab, and P. Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., 2004. ISBN 1558608567.

D. S. Nau, M. Ghallab, and P. Traverso. Blended Planning and Acting: Preliminary Approach, Research Challenges. In *AAAI Conference on Artificial Intelligence*, pages 4047–4051, 2015.

R. Navarro-Prieto and J. J. Cañas. Are Visual Programming Languages Better? The Role of Imagery in Program Comprehension. *International Journal of Human-Computer Studies*, 54(6):799–829, 2001.

N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100. ACM, 2007. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250746.

A. Newell. SOAR as a Unified Theory of Cognition: Issues and Explanations. *Behavioral and Brain Sciences*, 15(3):464–492, 1992.

T. Niemueller, A. Ferrein, and G. Lakemeyer. A Lua-based Behavior Engine for Controlling the Humanoid Robot Nao. In *Proceedings of RoboCup Symposium 2009*, 2009.

V. Niewada. Development of an Autonomous Mobile Robot. Master Thesis, Strasbourg University, 2015.

N. Nilsson. Teleo-Reactive Programs for Agent Control. *Journal of artificial intelligence research*, 1:139–158, 1993.

N. J. Nilsson. Shakey the Robot. Technical report, SRI International, Menlo Park, CA, 1984.

N. F. Noy, M. Crubézy, R. W. Fergerson, H. Knublauch, S. W. Tu, J. Vendetti, M. A. Musen, et al. Protege-2000: An Open-Source Ontology-Development and Knowledge-Acquisition Environment. In *AMIA Annual Symposium Proceedings*, volume 953, page 953, 2003.

S. O'Grady. RedMonk Index. Internet, 2020. URL `https://redmonk.com/sogrady/2019/03/20/language-rankings-1-19/`. Accessed: 2020-04-22.

OpenAI. OpenAI Five. Internet, 2019. URL `https://openai.com/blog/openai-five/`. Accessed: 2020-04-22.

S. Opfer, S. Jakob, and K. Geihs. Reasoning for Autonomous Agents in Dynamic Domains. In *ICAART*, pages 340–351, 2017.

S. Opfer, S. Jakob, A. Jahl, and K. Geihs. ALICA 2.0 - Domain-Independent Teamwork. In C. Benzmüller and H. Stuckenschmidt, editors, *KI 2019: Advances in Artificial Intelligence*, pages 264–272. Springer International Publishing, 2019. ISBN 978-3-030-30179-8.

S. Parent. A Possible Future of Software Development, 2007. URL `https://stlab.cc/legacy/figures/Boostcon_possible_future.pdf`.

M. R. Pedersen, L. Nalpantidis, R. S. Andersen, C. Schou, S. Bøgh, V. Krüger, and O. Madsen. Robot Skills for Manufacturing: From Concept to Industrial Deployment. *Robotics and Computer-Integrated Manufacturing*, 37:282–291, 2016. ISSN 0736-5845. doi: https://doi.org/10.1016/j.rcim.2015.04.002.

Plotly. Plotly. Internet, 2017. URL `https://plot.ly/`. Accessed: 2020-04-25.

U. Pohlmann, S. Dziwok, J. Suck, B. Wolf, C. C. Loh, and M. Tichy. A Modelica Library for Real-Time Coordination Modeling. In *9th International MODELICA Conference*, pages 365–374. Linköping University Electronic Press, 2012.

E. Pot, J. Monceaux, R. Gelin, and B. Maisonnier. Choregraphe: A Graphical Tool for Humanoid Robot Programming. In *18th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, pages 46–51. IEEE, 2009.

J.-Y. Puigbo, A. Pumarola, and R. A. Téllez. Controlling a General Purpose Service Robot By Means of a Cognitive Architecture. In *AIC@AI*IA*, pages 45–55, 2013.

Python Software Foundation. The Python Profilers. Internet, 2017. URL `https://docs.python.org/2/library/profile.html`. Accessed: 2020-04-25.

M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: An Open-Source Robot Operating System. In *IEEE International Conference on Robotics and Automation (ICRA), Workshop on Open Source Software*, volume 3, page 5. Kobe, 2009.

M. Radestock and S. Eisenbach. Coordination in Evolving Systems. In O. Spaniol, C. Linnhoff-Popien, and B. Meyer, editors, *Trends in Distributed Systems CORBA and Beyond*, pages 162–176. Springer Berlin Heidelberg, 1996. ISBN 978-3-540-70716-5.

J. W. Reeves. What is Software Design. *C++ Journal*, 2(2):14–12, 1992.

M. Resnick, J. Maloney, A. Monroy Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: Programming for All. *Communications of the ACM*, 52(11):60–67, 2009.

S. Riedel. LogView - Interactive Visualization of Robotic Log Data. Internet, 2018. URL `https://sebastianriedel.github.io/projects/logview`. Accessed: 2020-04-25.

J. Rintanen. Madagascar: Scalable Planning with SAT. *Proceedings of the 8th International Planning Competition (IPC-2014)*, 21, 2014.

S. Russell and P. Norvig. Artificial Intelligence: A Modern Approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25, 1995.

J. Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein. Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler. In *ACM SIGPLAN Notices*, volume 31, pages 1–11. ACM, 1996.

R. Sakagami. Cooperative Operation Framework for Heterogeneous Robotic Teams in Planetary Exploration. Master's thesis, The University of Tokyo, 2020.

P. Schillinger, S. Kohlbrecher, and O. von Stryk. Human-Robot Collaborative High-Level Control with an Application to Rescue Robotics. In *IEEE International Conference on Robotics and Automation (ICRA)*, Stockholm, Sweden, May 2016.

M. J. Schuster. *Collaborative Localization and Mapping for Autonomous Planetary Exploration : Distributed Stereo Vision-Based 6D SLAM in GNSS-Denied Environments*. PhD thesis, Universität Bremen, 2019.

M. J. Schuster, C. Brand, S. G. Brunner, P. Lehner, J. Reill, S. Riedel, T. Bodenmüller, K. Bussmann, S. Büttner, A. Dömel, W. Friedl, I. Grixa, M. Hellerer, H. Hirschmüller, M. Kassecker, Z.-C. Marton, C. Nissler, F. Ruess, M. Suppa, and A. Wedler. The LRU Rover for Autonomous Planetary Exploration and its Success in the SpaceBotCamp Challenge. In *IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, 2016.

M. J. Schuster, S. G. Brunner, K. Bussmann, S. Büttner, A. Dömel, M. Hellerer, H. Lehner, P. Lehner, O. Porges, J. Reill, S. Riedel, M. Vayugundla, B. Vodermayer, T. Bodenmüller, C. Brand, W. Friedl, I. Grixa, H. Hirschmüller, M. Kaßecker, Z.-C. Márton, C. Nissler, F. Ruess, M. Suppa, and A. Wedler. Towards Autonomous Planetary Exploration: The Lightweight Rover Unit (LRU), its Success in the SpaceBotCamp Challenge, and Beyond. *Journal of Intelligent & Robotic Systems (JINT)*, 2017.

M. J. Schuster, M. G. Müller, S. G. Brunner, and H. Lehner et al. Towards Heterogeneous Robotic Teams for Collaborative Scientific Sampling in Lunar and Planetary Environments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Workshop on Informed Scientific Sampling in Large-scale Outdoor Environments*, 2019.

M. J. Schuster, M. G. Müller, S. G. Brunner, H. Lehner, P. Lehner, R. Sakagami, A. Dömel, L. Meyer, B. Vodermayer, R. Giubilato, M. Vayugundla, J. Reill, F. Steidle, I. von Bargen, K. Bussmann, R. Belder, P. Lutz, W. Stürzl, M. Smíšek, M. Maier, S. Stoneman, A. F. Prince, B. Rebele, M. Durner, E. Staudinger, S. Zhang, R. Pöhlmann, E. Bischoff, C. Braun, S. Schröder, E. Dietz, S. Frohmann, A. Börner, H.-W. Hübers, B. Foing, R. Triebel, A. O. Albu-Schäffer, and A. Wedler. The ARCHES Space-Analogue Demonstration Mission: Towards Heterogeneous Teams of Autonomous Robots for Collaborative Scientific Sampling in Planetary Exploration. *submitted to IEEE Robotics and Automation Letters (RA-L)*, 2020a.

M. J. Schuster, B. Rebele, M. G. Müller, S. G. Brunner, A. Dömel, and B. Vodermayer et al. The ARCHES Moon-Analogue Demonstration Mission: Towards Teams of Autonomous Robots for Collaborative Scientific Sampling in Lunar Environments. In *submitted to European Lunar Symposium*, 2020b.

L. Shapiro. *The Routledge Handbook of Embodied Cognition*. Routledge Handbooks in Philosophy. Routledge, 2014. ISBN 0415623618,9780415623612. doi: 10.4324/9781315775845.

S. B. Sheppard, E. Kruesi, and J. W. Bailey. An Empirical Evaluation of Software

Documentation Formats. In *Conference on Human Factors in Computing Systems*, pages 121–124. ACM, 1982.

N. C. Shu. *Visual Programming Languages: A Perspective and a Dimensional Analysis*, pages 11–34. Springer US, 1986. ISBN 978-1-4613-1805-7. doi: 10.1007/978-1-4613-1805-7_2.

B. Siciliano and O. Khatib. *Springer Handbook of Robotics*. Springer-Verlag, 2007. ISBN 354023957X.

R. Simmons and D. Apfelbaum. A Task Description Language for Robot Control. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 3, pages 1931–1937. IEEE, 1998.

R. Simmons, D. Goldberg, A. Goode, M. Montemerlo, N. Roy, A. C. Schultz, M. Abramson, I. Horswill, D. Kortenkamp, and B. Maxwell. Grace: An Autonomous Robot for the AAAI Robot Challenge. Technical report, Carnegie Mellon University Pittsburgh PA, 2003.

R. G. Simmons. A Theory of Debugging Plans and Interpretations. In *AAAI Conference on Artificial Intelligence*, pages 94–99, 1988.

A. J. Simons. On the Compositional Properties of UML Statechart Diagrams. In *Rigorous Object-Oriented Methods*, 2000.

S. F. Smith. Is Scheduling a Solved Problem? In *Multidisciplinary Scheduling: Theory and Applications*, pages 3–17. Springer, 2005.

D. Speck, F. Geißer, and R. Mattmüller. SYMPLE: Symbolic Planning Based on EVMDDs. *IPC-9 Planner Abstracts*, pages 82–85, 2018.

S. S. Srinivasa, D. Berenson, M. Cakmak, A. Collet, M. R. Dogar, A. D. Dragan, R. A. Knepper, T. Niemueller, K. Strabala, M. V. Weghe, and J. Ziegler. HERB 2.0: Lessons Learned from Developing a Mobile Manipulator for the Home. *Proceedings of the IEEE*, 100:2410–2428, 2012.

A. Steck and C. Schlegel. SmartTCL: An Execution Language for Conditional Reactive Task Execution in a Three Layer Architecture for Service Robots. In *International Workshop on Dznamic Languages for Rbotic and Sensors Systems (DYROS/SIMPAR)*, pages 274–277, 2010.

F. Steinmetz and R. Weitschat. Skill Parametrization Approaches and Skill Architecture for Human-Robot Interaction. In *IEEE International Conference on Automation*

Science and Engineering (CASE), pages 280–285, Fort Worth, Texas, USA, Aug 2016. doi: 10.1109/COASE.2016.7743419.

F. Steinmetz, A. Wollschläger, and R. Weitschat. RAZER – A Human-Robot Interface for Visual Task-Level Programming and Intuitive Skill Parametrization. *IEEE Robotics and Automation Letters (RA-L)*, 3(3):1362–1369, 2018.

Strategic Missions and Advanced Concepts Office. Solar Power Technologies for Future Planetary Science Missions. Technical Report JPL D-101316, Jet Propulsion Laboratory, 2017.

C. Sürig. Analyse und Evaluierung semantischer Logikplaner, mit anschließender Integration des Vergleichsergebnisses in RAFCON. Bachelor Thesis, Hochschule München, 2018.

G. J. Sussman. The Virtuous Nature of Bugs. In *Proceedings of the 1st Summer Conference on Artificial Intelligence and Simulation of Behaviour*, pages 224–237. IOS Press, 1974.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. A Bradford book. MIT Press, 1998. ISBN 9780262193986.

S. Tanimoto and E. Glinert. Pict: An Interactive Graphical Programming Environment. *Computer*, 17(11):7–25, 1984. ISSN 0018-9162. doi: 10.1109/MC.1984.1658997.

S. L. Tanimoto and E. P. Glinert. Designing Iconic Programming Systems: Representation and Learnability. In *IEEE Proceedings Workshop on Visual Languages*, pages 54–60, 1986.

M. Tenorth and M. Beetz. KnowRob - Knowledge Processing for Autonomous Personal Robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4261–4266. IEEE, 2009.

U. Thomas, G. Hirzinger, B. Rumpe, C. Schulze, and A. Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 461–466, 2013.

TIOBE Software Quality Company. TIOBE Index. Internet, 2020. URL `https://www.tiobe.com/tiobe-index/`. Accessed: 2020-04-25.

A. Torralba, C. L. López, and D. Borrajo. Abstraction Heuristics for Symbolic Bidirectional Search. In *IJCAI*, pages 3272–3278, 2016.

G. van Rossum. Comparing Python to Other Languages. Internet, 1997. URL `https://www.python.org/doc/essays/comparisons/`.

D. Vernon. *Artificial Cognitive Systems: A Primer*. The MIT Press. MIT Press, 2014. ISBN 9780262028387.

R. Verret and S. Thompson. Custom FPGA-Based Tests with COTS Hardware and Graphical Programming. In *IEEE AUTOTESTCON*, pages 1–5, 2010. doi: 10.1109/AUTEST.2010.5613619.

M. Vilzmann. Mission Planning and Verification for Autonomous Unmanned Aerial Vehicles. Master's thesis, TU Clausthal, 2016. URL `http://elib.dlr.de/115188/`.

O. Vinyals, I. Babuschkin, J. Chung, M. Mathieu, M. Jaderberg, W. Czarnecki, A. Dudzik, A. Huang, P. Georgiev, R. Powell, T. Ewalds, D. Horgan, M. Kroiss, I. Danihelka, J. Agapiou, J. Oh, V. Dalibard, D. Choi, L. Sifre, Y. Sulsky, S. Vezhnevets, J. Molloy, T. Cai, D. Budden, T. Paine, C. Gulcehre, Z. Wang, T. Pfaff, T. Pohlen, Y. Wu, D. Yogatama, J. Cohen, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, C. Apps, K. Kavukcuoglu, D. Hassabis, and D. Silver. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. *DeepMind Blog*, 2019. Accessed: 2020-04-22.

V. Vlachoudis. FLAIR: A Powerful but User Friendly Graphical Interface for FLUKA. In *International Conference on Mathematics, Computational Methods & Reactor Physics (M&C 2009)*, volume 1, page 3, New York, USA, 2009.

R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The CLARAty Architecture for Robotic Autonomy. In *IEEE Aerospace Conference*, volume 1, pages 1–121. IEEE, 2001.

M. Wächter, S. Ottenhaus, M. Kröhnert, N. Vahrenkamp, and T. Asfour. The ArmarX Statechart Concept: Graphical Programing of Robot Behavior. *Frontiers in Robotics and AI*, 3:33, 2016. ISSN 2296-9144. doi: 10.3389/frobt.2016.00033.

R. Watanabe, M. Minami, and H. Narioka. Digital Map Display Zooming Method, Digital Map Display Zooming Device, and Storage Medium for Storing Digital Map Display Zooming Program, 2006. US Patent 7,075,558.

A. Wedler, B. Rebele, J. Reill, M. Suppa, H. Hirschmüller, C. Brand, M. Schuster, B. Vodermayer, H. Gmeiner, A. Maier, B. Willberg, K. Bussmann, F. Wappler, and M. Hellerer. LRU – Lightweight Rover Unit. In *13th Symposium on Advanced Space Technologies in Robotics and Automation (ASTRA)*, May 2015. URL `http://elib.dlr.de/102155/`.

A. Wedler, M. Vayugundla, H. Lehner, P. Lehner, M. J. Schuster, S. G. Brunner, W. Stürzl, A. Dömel, H. Gmeiner, B. Vodermayer, R. Rebele, I. Grixa, K. Bussmann,

J. Reill, B. Willberg, A. Maier, P. Meusel, F. Steidle, M. Smisek, M. Hellerer, M. Knapmeyer, F. Sohl, A. Heffels, L. Witte, C. Lange, R. Rosta, N. Toth, S. Völk, A. Kimpe, P. Kyr, and M. Wilde. First Results of the ROBEX Analog Mission Campaign: Robotic Deployment of Seismic Networks for Future Lunar Missions. In *68th International Astronautical Congress (IAC)*, 2017.

A. Wedler, M. Wilde, J. Reill, M. J. Schuster, M. Vayugundla, S. G. Brunner, K. Bussmann, A. Dömel, M. Drauschke, H. Gmeiner, H. Lehner, P. Lehner, M. G. Müller, W. Stürzl, R. Triebel, B. Vodermayer, A. Börner, R. Krenn, A. Dammann, U.-C. Fiebig, E. Staudinger, F. Wenzhöfer, S. Flögel, S. Sommer, T. Asfour, M. Flad, S. Hohmann, M. Brandauer, and A. O. Albu-Schäffer. From Single Autonomous Robots Toward Cooperative Robotic Interactions for Future Planetary Exploration Missions. In *International Aeronautical Congress (IAC)*, volume 42. International Aeronautical Federation (IAF), 2018.

A. Wedler, J. Reill, M. J. Schuster, M. Vayugundla, and S. G. Brunner et al. Insights into the Robotic Exploration Activities in the Research Section of the German Aerospace Center (DLR). In *Symposium on Advanced Space Technologies in Robotics and Automation (ASTRA)*, 2019a.

A. Wedler, M. Wilde, J. Reill, M. J. Schuster, M. Vayugundla, S. G. Brunner, K. Bussmann, A. Dömel, H. Gmeiner, H. Lehner, P. Lehner, M. G. Müller, W. Stürzl, B. Vodermayer, M. Smisek, B. Rebele, and A. O. Albu-Schäffer. Analogue Research from ROBEX Etna Campaign and Prospects for ARCHES Project: Advanced Robotics for Next Lunar Missions. In *European Planetary Science Congress (EPSC)*, 2019b.

A. Wedler, M. G. Müller, M. J. Schuster, S. G. Brunner, and P. Lehner et al. First Results from the Multi-Robot, Multi-Partner, Multi-Mission, Planetary Exploration Analogue Campaign on Mt. Etna in Summer 2020. In *submitted to 71st International Astronautical Congress*, Dubai, United Arab Emirates, 2020.

R. Welch, D. Limonadi, and R. Manning. Systems Engineering the Curiosity Rover: A Retrospective. In *8th International Conference on System of Systems Engineering*, pages 70–75, 2013. doi: 10.1109/SYSoSE.2013.6575245.

What! Studio. Python Profiling. Internet, 2017. URL `https://github.com/what-studio/profiling`. Accessed: 2020-04-25.

B. C. Williams, M. D. Ingham, S. H. Chung, and P. H. Elliott. Model-Based Programming of Intelligent Embedded Systems and Robotic Space Explorers. *Proceedings of the IEEE*, 91(1):212–237, 2003.

H. Williams, J. Bridges, R. Ambrosi, M.-C. Perkinson, J. Reed, L. Peacocke, N. Bannister, S. Howe, R. O'Brien, and A. Klein. Mars Reconnaissance Lander: Vehicle and Mission Design. *Planetary and Space Science*, 59(13):1621–1631, 2011. ISSN 0032-0633. doi: https://doi.org/10.1016/j.pss.2011.07.011. Exploring Phobos.

W. Wulf and M. Shaw. Global Variable Considered Harmful. *ACM Sigplan Notices*, 8 (2):28–34, 1973.

S. E. Xavier. *Theory of Automata, Formal Languages and Computation*. New Age International, 2005.

J. Ye. *Building Applications with Spring 5 and Vue.js 2: Build a Modern, Full-Stack Web Application Using Spring Boot and Vuex*. Packt Publishing, 2018. ISBN 9781788831253.

P. Yiapanis, G. Brown, and M. Luján. Compiler-Driven Software Speculation for Thread-Level Parallelism. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 38(2):1–45, 2015. ISSN 0164-0925. doi: 10.1145/2821505.

P. Zech, E. Renaudo, S. Haller, X. Zhang, and J. H. Piater. Action Representations in Robotics: A Taxonomy and Systematic Classification. *CoRR*, abs/1809.04317, 2018.

B. Zhang, J. Liu, and H. Chen. AMCL Based Map Fusion for Multi-Robot SLAM with Heterogenous Sensors. In *IEEE International Conference on Information and Automation (ICIA)*, pages 822–827. IEEE, 2013.

K. Zhang. *Visual Languages and Applications*. Springer Science & Business Media, 2010.