Utrecht University

Universiteit Utrecht

A Planning Module for a ROS-Based Ubiquitous Robot Control System

Technical Artificial Intelligence

MSc Thesis

Author:
ing. Pieterjan P.J.G. van Gastel
ICA-3898857

Supervisors:
ir. Rob R.J.M. Janssen
dr. Mehdi M.M. Dastani

Artificial Intelligence division,
Intelligent Systems group
Princetonplein 5, De Uithof, Utrecht,
The Netherlands

March 27th, 2014

Abstract

As the life expectancy of people is increasing and health care gets more expensive, a desire for assistive technologies in health care is growing. Such assistive technologies are researched and developed at the robotics lab of Eindhoven University Technology (TU/e) in the form of *service robots*. Research efforts are made to combine these robots with intelligent environments. An example of this is the research project described in this thesis. The idea is to have several service robots in an environment, such as a health care institution, and to have them controlled by a central control system. Therefore, a ubiquitous robotic framework for multi-robot planning and control is proposed in this thesis. This framework integrates with existing design efforts of the open-source robotics community. The main focus of this research project is on the planning module of the proposed system.

For such a multi-robot control system, a planning module is needed to handle the planning and scheduling of multiple tasks for the robots in an intelligent way. Therefore, the field of automated planning for robotic systems is analyzed first in this thesis. This is followed by a literature study on subjects ranging from automated planning, to Semantic Web and robotic technologies, such that all fields of ubiquitous robotics are covered. Based on the analysis and the literature study requirements are formed for the system. These requirements are then used to create the design of the framework. Furthermore, a prototype of the framework is implemented and experiments are performed with this prototype. The results of these experiments are evaluated. The thesis also includes a comparison between the proposed system and other similar planning systems as part of the discussion after the evaluation.

The design of the framework includes technologies such as *Robot Operating System* (ROS), *Semantic Web Ontology languages* (OWL and OWL-S), *RoboEarth Cloud Engine* (Rapyuta), and *MIndiGolog*, as they are innovative and provide genericness. MIndiGolog is a high-level agent programming language for multi-agent systems, and is based on *situation calculus*, a logic formalism designed for representing and reasoning about dynamic domains. MIndiGolog can reason over world states and transitions between states. In the proposed system, robots are also required to have a robot-type description available, whichs describe the specific robot capabilities of each connected robot, so that the planner can select the right robot for specific tasks. The planner searches for plans to accomplish user-given tasks.

The programming languages *Python* and *Prolog* are used to implement the prototype of the framework. *PySWIP* is used to query Prolog rules in Python. The prototype includes an executive layer, a planning layer and an ontology layer. This thesis focuses on the planning layer.

Several qualitative tests and an experiment with real robots are performed with the implemented prototype. The tests are performed to validate the functionality of the planning module of the system. The experiment describes a use case in which robots need to help with a *cocktail party*. In this use case, the robots need to take orders from guests, bring drinks to the guests, find empty drinks, and clean up empty glasses. The performed experiment however only includes the subtask of detecting empty drinks, due to limited time and resources. The robots need to navigate to the location where they expect empty drinks to be found and then perceive the empty drink.

The results of both the tests and the experiment are successful, as the tests performed as expected and the robots navigated to the nearest location where they expect empty drinks to be found in the experiment. There were however some performance issues with the communication interface of the system, including latency and bandwidth issues. Based on the results, the proposed framework is evaluated as a feasible approach to a ubiquitous robotic system for multi-robot planning and control.

Keywords: *Ubiquitous robot networks, situation calculus, Semantic Web, RoboEarth, MIndiGolog, Rapyuta*

Acknowledgments

First of all, I would like to thank *Sanne*, my girlfriend, for her patience and support. Furthermore, I want to thank my friends and family for their support. Especially my sister, *Marieke*, for checking the spelling and grammar of (parts of) my thesis.

I would also like to thank both my supervisors for their guidance and support: *Rob Janssen*, my supervisor at Eindhoven University of Technology, and *Mehdi Dastani*, my supervisor of Utrecht University.

Furthermore, I would like to thank the students I worked with during this project: *Erik Geerts*, *Jean-Pierre Denissen* and *Ziyang Li*. The project could not be a success without them. I also want to thank all other students at the robotics lab, especially those who gave me advice during the weekly RoboEarth meetings. Particularly *Sjoerd van den Dries* and *Tim de Jager* were very helpful, as they both have a computer science background.

I would like to thank *René van de Molengraft* and *Maarten Steinbuch* for allowing me to work on my research project at the robotics lab of Eindhoven University of Technology (TU/e), and for providing me with a supervisor who guided me during the project. I also want to thank *Jan Broersen* for helping me contact the people of the TU/e robotics lab, and I want to thank *Henry Prakken* for helping me find a supervisor at Utrecht University.

Finally, I would also like to thank *ETH Zürich IDSC* for their support in setting up the RoboEarth Cloud Engine framework. With special thanks to *Dominique Hunziker* and *Mohanarajah Gajamohan* who helped me understand their framework better.

As a final note: this work is supported in part by the EU FP7 Project *RoboEarth* under grant agreement number 248942.

Contents

Chapter 1

Introduction

"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it." - Mark Weiser [81]

A trend is visible in the evolution of computer technology. This trend is basically a shift in the ratio of users per computer. The first computers were very large *mainframe computers*, shared by many people. This first era evolved to the personal computing era, in which one computer is used by one person. This, in turn, is followed by an era in which each person uses various networked computers simultaneously, the era of *"ubiquitous computing"*, a term coined by Mark Weiser around 1988 [82]. The same is applicable to robotics technology. In the future, every natural human environment is also an intelligent robotic environment. It is the era of *"ubiquitous robotics"*, in which people forget they are actually using robots.

One could think of many examples of a robotic environment with multiple mobile robots, some of which are already common today, and some of which will be common in the future. For example, a *logistics environment* such as a warehouse in which everything is handled by robots, or the harbor of Rotterdam where autonomous robotic cranes and computer controlled chariots handle container loading and stacking. Or in *manufacturing*, such as a car factory in which many heterogeneous robots need to work together to build cars. Furthermore, in the future, many robots will ride around to take care of patients in *health care institutions*, where human-like mobile manipulators are deployed to deliver medicine and to monitor patients. In all these examples, cooperative robotic planning is needed to create an automated environment.

Even though several frameworks and middleware for ubiquitous robots are proposed in the past (e.g. the SURF framework [26], the PEIS Kernel [9], and UPnP Robot Middleware [1]), none of them show (to my knowledge) an experiment in which a task planner instructs multiple robots to work together to reach a user-given goal. Also, none of them use ROS to provide genericness of the robots in the form of web services. Chapter 8 provides a full comparison of these systems.

As the future brings more robots in everyday environments, a good task planner, which is able to handle planning and control of multiple robots, is desired to increase the functionality of these robots. Even though several robotic planning systems are proposed in the past, there is no task planner yet powerful enough to handle a dynamic environment with multiple mobile robots.

1.1 Problem Description

The coming era of ubiquitous robotics has inspired researchers to investigate ways to make robots dissolve in human environments. For this, robots need to have the proper set of capabilities to operate in these environments. The researchers of *Eindhoven University of Technology* (TU/e) have a vision of creating *autonomous service robots*. These robots are meant to help people at home or at health care institutions. They want to extend this research into the field of ubiquitous robotics. Their view on ubiquitous robotics is to have an intelligent environment, in which the user can request services, and that the environment is using robots to provide the services. The service robots of the TU/e use the *Robot Operating System* (ROS) [18], which has the potential to make the robots more generic and thus easier to be found by other systems in an intelligent environment. Every sensor and actuator on the robots needs to become a web service, every robot becomes a collection of web services. These web services should be easily accessible by a web-based task planner. The web services form the knowledge domain of the planner. This task planner searches for web services that fulfill the user-given task, and then selects the plan with the lowest cost. The planner knows the relations of the web service collections when planning.

This planner is part of a larger system. This system can then command a robot, multiple robots, or any other web service to fulfill the given task(s). The other parts of this system include the control layer and the knowledge (ontology) layer. The architecture of this system is further explained in the next chapter.

My focus is to develop a web-based entity (or software agent) that uses service robots represented as collections of web services to create and execute plans for completing tasks provided by an user. The goal of the project is to investigate how such a web-based ubiquitous intelligent software agent for robotic planning and control can be developed and realized. This agent receives goal tasks from users, and a planning component to find the web services which fulfill the tasks requested by the user. These web services can be any web service on the Semantic Web, and are not restricted to web services representing (parts of) robots. Furthermore, the target platform is ROS, as it provides genericness, it is open-source, and it builds upon the technologies currently used at the TU/e.

The purpose of the web-based agent is to use service robots to obtain user-requested information or achieve user-given goals. Because of this, the proposed system fits well in the concept of ubiquitous robotics.

1.2 Thesis Overview

This section gives an overview of the thesis. It shortly mentions the contents of each chapter.

The next chapter is an introductory chapter to show the larger system of which the proposed planning module described in this thesis is part of. Chapter 3 is about the analysis of the project problem domain. It discusses robotic planning in literature and in practice, and defines the requirements of the proposed planning system. Chapter 4 provides a literary study which discusses the preliminaries leading to the design of the system and other topics that were researched before design choices could be made. Chapter 5 presents the design of the proposed planning module. Chapter 6 discusses the implementation of the proposed planning module. It gives a description of the implemented prototype and describes all implementation choices. The implemented prototype of the system is evaluated in chapter 7. This chapter describes the experiments that are performed with the system and discusses the results of these experiments. Chapter 8 discusses work related to the project, and issues and opportunities for potential future research. It also highlights the differences between the proposed system and other frameworks for robotic planning. Finally, chapter 9 gives the conclusion of this thesis.

The reader of this thesis is assumed to have basic knowledge of Prolog, and if not; please refer to the *"Learn Prolog Now!"* website [7]. This website provides a good introduction to Prolog.

Chapter 2

System Architecture

The focus of the research project discussed in this thesis is on the *planning component* of a larger robotic control system. This chapter is therefore an introductory chapter, which explains the design of the system architecture of the whole robotic control system before getting to the planning component. It also presents the requirement of the planning module. This chapter is provided as one of the first chapters of this thesis, so that all following chapters make more sense to the reader.

Other students have also worked on this robotic control system (further referred to as the system), these are: *Rob Janssen* [35], *Erik Geerts* [21] and *Jean-Pierre Denissen* [13]. Erik Geerts's focus is on the executive (or control) layer of the system, whereas Jean-Pierre Denissen's work is on the ontology (or knowledge) layer of the system.

Rob Janssen describes the complete system architecture in his paper [35]. The general concept of this proposed system consists of a centralized, cloud-based computing environment, that has fast access to global information bases containing relevant information for the scheduling and execution of robot tasks. This information consists, for example, of localization and navigation maps, object models (which can be used for perception and navigation), and reusable *task descriptions*. Task descriptions describe how robot tasks are performed.

Each robot needs to have a "robot type-description" available, to compose multi-robot plans for a wide variety of robots. Such a "robot type-description" allows the planning module to allocate robot tasks based on the capabilities of connected robots.

A partial realization of such a system is found in the RoboEarth Project, which is discussed in section 4.3.2. As mentioned in that same section, the RoboEarth project also provides in a cloud-based communication framework, called Rapyuta. When a task planner and a high-level executive (responsible for the grounding, communication and parametrization of atomic tasks) are added to the (RoboEarth) knowledge bases, the result is a system architecture for multi-robot planning and control. This architecture is further described in the next section. This is followed by a section on the focus of this thesis and the requirements of the planning module.

2.1 System Overview

Figure 2.1 shows the overview of the proposed multi-robot planning system. As seen in this figure, the planning module (highlighted in the center of the server) is only a part of the whole system.

This system allows for many robots to connect simultaneously. The functionalities of the sensors and actuators of a robot are presented to the planning module as modular blocks, after the robot connects to the system. The presentation of these modular blocks to the system is similar to the way web services expose their functionality on the Semantic Web, so that they can be found by web service discovery and composition tools (such as in [8], [64] and [80]). Furthermore, the planning module plans robot tasks to accomplish user-given requests. The planning module makes use of knowledge bases, which hold a variety of information, when planning these tasks. A detailed description of the planning module's functionality is provided in chapter 6.

The proposed system is innovative for the technologies incorporated in its design. The chosen technologies for the proposed system are: OWL-S [51] for web service ontologies and the knowledge layer, Prolog [7] for higher-level task planning and the deliberative layer, ROS Python [18] for robotic control and the executive layer, and the RoboEarth Cloud Engine (Rapyuta) [31] for communication between (hardware and software) agents. OWL-S is required to represent robot actuators and sensors as web services, so that it provides

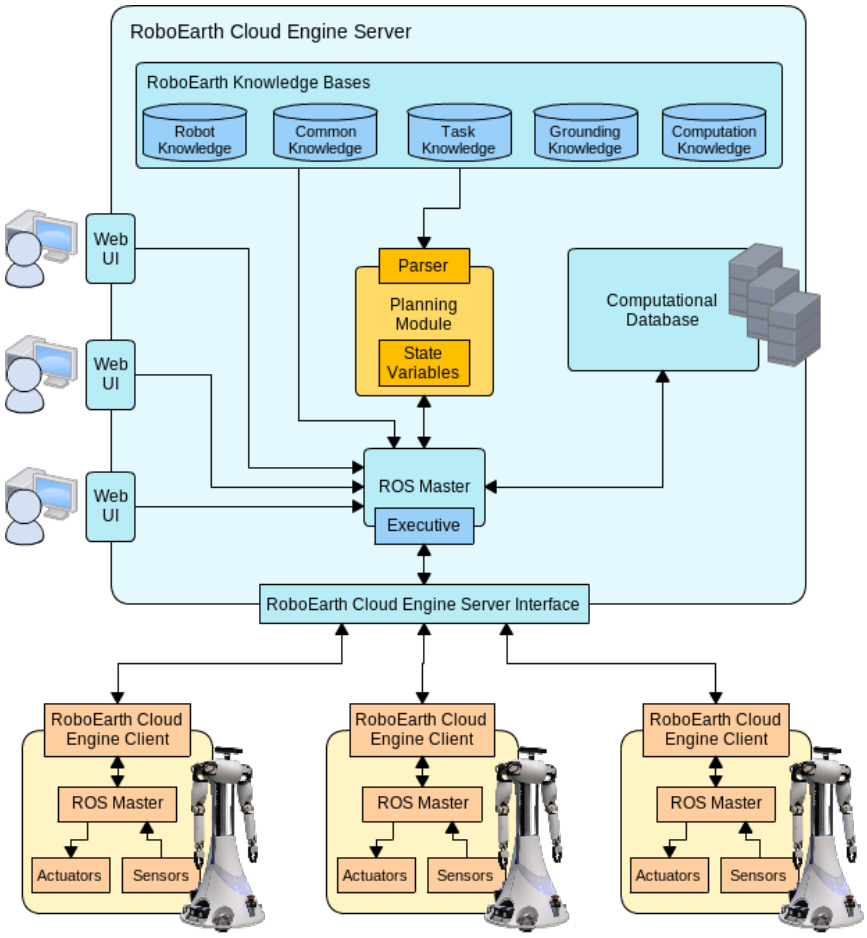

Figure 2.1: The Cloud-Based Task-Scheduling System Architecture.

genericness and can be expanded upon with other Semantic Web technologies.

The combination of OWL-S with Prolog and ROS Python is new. Such a planning system architecture is not designed or developed before. However, the choice of these technologies is not purely based on their innovative nature, but also because they provide a generic solution. For example, ROS is chosen as base of the system, because it is a framework used by many robots today. Rapyuta is based on ROS and is therefore easy to integrate with other ROS systems and an excellent choice for communication between different ROS robots. Even though ROS also allows to communicate between robots, Rapyuta is more modular as it allows every robot to have its own ROS Master. The design choices specifically for the planning module are discussed in chapter 5. The technologies mentioned here are further investigated in chapter 4. The next section describes a synopsis of the contributions of the planning module.

2.2 Focus of Thesis

As mentioned in the previous section, this thesis focuses on the *planning module* of the robotic planning and control system presented in the previous section. This section describes this focus in more detail, and presents some preliminary design choices. The planning component needs to function with the other components of the system. It needs to handle the information from both the knowledge bases and the executive. It receives knowledge about the world from the OWL-S knowledge base, which is parsed as a Prolog file such that the planning module can read it. Furthermore, the planning module receives user-given tasks from the executive and the executive queries the planning module for new plans. A feedback loop is required between the planner and the executive, which makes the planner an online planner. Online planning is essential for planning in a dynamic environment, as the world in which the robots act is too complex to model completely.

Next to handling data from the knowledge bases and executive, the planning module must also include the following contributions:

1. The inclusion of *OWL-S control construct* handlers, so that the planner reasons over the knowledge bases. OWL-S is a standard for describing web services. Robot components are described as OWL-S services, so that the system can easily be extended with more Semantic Web technologies.

2. The inclusion of *temporal planning*, so that the planner reasons over time and durations of actions. This allows the planner to find the cheapest plan out of multiple plans.

3. The inclusion of *multi-robot control*, so that the planner handles multiple robots.

4. The inclusion of *robot capability matching*, so that the planner selects only robots with the right functionalities for certain tasks.

5. The inclusion of *online planning*, so that the executive can query actions at every time and the planner uses current data of the world state.

These contributions are described in more detail in the design of the planning module (chapter 5). Before the design is discussed, the analysis of the problem is described in the next chapter and chapter 4 describes a literary study, providing background information on the technologies and theories relevant to the proposed system. First, the requirements of the planning module are presented in the next section.

2.3 Requirements of the Planning Module

The requirements of the planning module are partially based on the problem domain analysis presented in the next chapter, and on the following four primary goals of the system:

1. The system serves as a centralized task controller for a wide variety of robot platforms and computational algorithms.

2. The coordination of robots is dictated by the central controller only.

3. The computational algorithms are deployed in a distributed and highly optimized computing environment, instead of running locally on the robots.

4. The system needs to have fast and direct access to a knowledge base that is capable of storing and retrieving task-related knowledge securely and efficiently.

Goal number three is required because it is desirable that the involved robot platforms remain lightweight by reducing onboard computing requirements. Also, the task-related knowledge mentioned in goal number four is required in vast amounts for the execution of tasks in human domains. Examples of such task-related knowledge are binary models for perception and logic-based representations for reasoning.

The requirements described in this section give direction to the research project and provide in a base for the design of the system. Based on the previously mentioned goals of the system, and the architecture presented previously in this chapter, the requirements for the planning module are:

- The planning module must find the most optimal plan (with the given knowledge), if a plan exists.

- The planning module must handle time and duration of actions (temporal planning), so that it selects the cheapest plan.

- The planning module must handle concurrency of actions, so that it makes plans for multiple robots.

- The planning module must handle all data that is parsed or provided by the knowledge base, so that the planner has a planning domain to plan for.

- The planning module must provide plans which can be executed by the control unit, so that the plans are actually executable.

- The planning module should use web service composition to find available robots, so that it finds available robots that are connected to the web.

Temporal planning is required because the duration of an action can be of significant importance of a plan. Temporal planning and planning with concurrency is further discussed in section 4.1.5. The most optimal plan for a planning problem can be explained as finding the plan with the best suitable robot for every task and the plan with the shortest expected time of execution (i.e. the cheapest plan). The best suitable robot is defined as the robot that is nearest to the location where the task needs to be performed and the robot with the matching capabilities for the task.

Appendix A provides more detail for the requirements of the system. It describes seven functional requirements and twelve non-functional requirements.

Chapter 3

Problem Domain Analysis

Most robotic planning in literature and practice is about motion planning, path planning, and manipulation planning. The focus of this chapter lies more on the higher execution level of robots, which is known as *task planning*.

Ghallab et al. [23] use the term *'plan synthesis'* for task planning. They explain it as a domain-independent planning model, that takes three things as input: the models of all known actions, a description of the state of the world, and some objective. It then returns an organized collection of actions (a plan) whose global effect, if they are carried out and perform as modeled, achieves the given objective. This section however does not discuss domain-independent planning, but domain-specific planning, as it is about the robot planning domain. A domain-specific planner does not take a domain as input as it already has one internally described. However, to avoid confusion it must be stated here that the planning module discussed in this thesis is not a domain-specific planner, but a domain-independent planner. Even though it plans for the robot domain, the planning domain is provided by the knowledge bases in the from of OWL-S descriptions. This allows the planning module to easily be extended with other planning domains. Not every planning system functions the same as the proposed planning module of this thesis. This chapter describes a general view on robotic task planning.

The next section is about robotic planning and control in literature. However, it is also important to know how robotic planning is done in practice. Therefore, another section takes this into account. After the background information on robotic planning and control is discussed, a final section discusses the requirements of the robotic planning system.

3.1 Robotic Planning and Control in Literature

This section is based on Ghallab's short chapter about planning in robotics in their book on automated planning. [23]

An engineer can hand-code every reaction to every possible situation in a robot's program, which is good for simple environments, such as production lines. However, the hand-coding approach stops being feasible as the diversity of tasks and the variety of environments in which a robot needs to act grow. Planning will make it simpler to program a robot in complex environments, and it will enhance the robot's usefulness and robustness. Even so, planning should be closely integrated with the reactive and learning capabilities of a robot, because of a robot's interaction with the physical world. Reactive capabilities are needed such that robots can respond fast to external events, and learning capabilities are needed to handle changing environments. Therefore, the specific requirements of planning in robotics are as follows:

- Online input comes from sensors and communication channels;
- Heterogeneous models of the environment and of the robot, as well as knowledge on the noise of information are acquired through sensors and communication channels;
- Direct integration of planning with acting, sensing, and learning.

In *Task planning* the primitives for plans are tasks such as *'navigate to location'*, *'retrieve item'*, and *'pick up object'*. These tasks are abstract and far from being primitive sensory-motor functions. Ghallab et al. write that task planning in robotics needs to handle time and resource allocation, dynamic environments, uncertainty and partial knowledge, and incremental planning consistently integrated with acting and sensing.

Chapter 4 provides more information on automated planning (section 4.1) and robotic technologies (section 4.3). The next section continues with robotic planning and control, as it is done in practice.

3.2 Robotic Planning and Control in Practice

Russell and Norvig write in their book [70] that most modern-day robotic architectures use reactive techniques at the lower level of control and deliberative techniques at the higher levels. However, many robots in use today are teleoperated and thus do not necessarily contain a higher-level planning component in the software. Still, some teleoperated robots also utilize autonomous or semi-autonomous functions (e.g. [11]). *Robotic teleoperation*, or telepresence, involves carrying out tasks with the aid of robotic devices. Examples of robots which are mainly teleoperated today are robots for hazardous environments [34], health care robots [33], and exploration robots [50]. Teleoperation is still preferred over autonomy for many robotic domains for various reasons: it is less expensive, the expertise of humans (e.g. in bomb disposal) cannot always be easily modeled, and human contact is still important and not easily simulated in robots (e.g. in health care). Most work on autonomous robots is done on robotic cars and transportation robots, and their level of autonomy is still increasing. An example of autonomous transportation robots can be found in the Rotterdam Harbor. The Rotterdam Harbor is one of several sites where transshipment operations are performed with *'Automated Ground Vehicles'* (AGVs) [63][2]. *Transshipment operations* include operations such as moving a container between two ships or moving a container from a ship to a train, and is usually performed in some storage area. AGVs (in general) are mostly teleoperated, but there are ambitious projects that aim at more flexible and autonomous operations. One such project is the Martha project. This project is explained in the next section. The section after that describes robotic systems for hospitals, and another section briefly describes a few more examples.

3.2.1 The Martha Project

The Martha project [2][3] includes a centralized planner which schedules jobs for autonomous container transportation robots. After jobs are scheduled to the robots, the individual robots communicate with each other to form their own plans such that the plans do not conflict with each other. Resources and locations can be reserved by one robot, forcing another robot to use different resources in its plan. Robots have a high level of autonomy in the Martha project.

The Martha project is an interesting project to view from the automated planning point of view, as plans are generated for multiple robots which are working together to achieve a common goal. The centralized planner is written in C-PRS [3], a procedural reasoning system, and the individual robots use a topological planner as well as a motion planner [2].

The objectives of the *Martha project* are "the operation and control of a large fleet of autonomous mobile robots for container transshipment tasks in harbors, airports and marshaling yards." A typical setting is that of a large harbor where a fleet of up to 100 robots is in charge of loading and unloading containers from arriving vehicles to departing ones. In the Martha project, a map of the environment is provided to each robot. This map is composed of metrical data and a topological graph labeled with a few types of attributes (such as: loading/unloading zones, routes, piles, docks etc.). The map is static, but the environment is changing due to other vehicles and unmodeled obstacles.

Another example where planning and control for multiple robots is performed by a centralized planner is in a hospital environment. The next section shortly discusses this. Another section after that mentions a few other examples.

3.2.2 Hospital Robots

There are several commercial robot-based logistics systems that are successfully installed in hospitals [60]. It is proven [15][68] that autonomous delivering improves the efficiency of the hospital transportation. A robot can transport many items, including: medicine, medical devices, specimens, food, documents, waste, etc. While most of the robotic systems in use in hospitals today have some form of navigation and path planning, only a few systems have a group control system to monitor and control several robots simultaneously. One such an example is the Matsushita's Robotic Blood Sample Courier System, which consists of a group of Hospi robots. A *Hospi robot* (figure 3.1) is an autonomous delivery robot, made by Panasonic. The Matsushita System is designed to control blood samples delivery and other transportation tasks within a hospital. The main computer assigns various tasks to individual robots.

Figure 3.1: A Hospi Robot, as presented by Niechwiadowicz and Khan (2008) [60].

3.2.3 Other Examples

There are many more examples of robotic planning and control in practice. For example agricultural robots such as [16], [30], [49] and [75], or space exploration robots, such as NASA's Mars exploration rovers (Spirit, Opportunity and Curiosity) [50] or the Ambler [73], a six-legged autonomous robot designed for planetary exploration. The Ambler robot (among other mobile robot systems) uses the Task Control Architecture, as presented by Simmons [74]. Simmons writes that the Task Control Architecture provides "distributed communications, hierarchical task decomposition, temporal constraints to coordinate subtasks, resource management, monitoring, and exception handling." A NASA robot used for space station assembly [17] also uses this Task Control Architecture.

Similar to most other robot planning systems, the Task Control Architecture is only applicable to planning and control for single robots, but it does provide some insight in how robotic planning is done for real-world situations. This insight can be used to set up the requirements for a robotic planning system for multi-robot control.

Chapter 4

Background

Even though the proposed planning module is mostly a combination of automated planning techniques and robotic technologies. This chapter also describes semantic web technologies to give the reader a full understanding of the complete proposed multi-robot control system (as described in chapter 2 and [35]), and because the planning module is required to handle robot knowledge described as web services.

Automated planning, robotics and semantic web technologies are required to be combined for the multi-robot control system for several reasons. First of all, robotics are essential to the system as robots need to be controlled. Automated planning is needed to compose plans in a fast and efficient way, so that multiple robots can be controlled based on these plans to achieve common goals. Furthermore, automated planning and robotics need to be combined with semantic web technologies to increase both the genericness and the expandability of the system.

This chapter discusses the theoretical foundations of these techniques and technologies. It shows all topics researched in this project before the choice of the definite design direction, and not only technologies used in the final design. The first section provides an introduction to automated planning, the second section explains the concept of the Semantic Web, and the third section provides a description of robotic technologies. A fourth section summarizes and compares specific implementations of the technologies with each other.

4.1 Automated Planning

This section provides a basic introduction to automated planning, a subfield of Artificial Intelligence. It builds heavily on the book *Automated Planning, theory and practice* written by Ghallab et al. (2004) [23], which explains automated planning in a comprehensive way, and on Nau's paper [58], which explains current trends in automated planning. Ghallab et al. write that "planning is the reasoning side of acting." A conceptual model for planning is needed to provide a general model for a dynamic system, because planning is concerned with organizing actions for changing the state of a system. This section starts with describing this conceptual model. Next it discusses classical planning, then hierarchical task network (HTN) planning, followed by situation calculus, and finally temporal planning techniques.

4.1.1 Conceptual Model for Planning

A *conceptual model*, as described in chapter 1 of Ghallab's book [23], and in Nau's paper [58], is a simple theoretical device (or tool) for describing and reasoning about a problem. Even though it can depart significantly from the computational concerns for solving that problem, it can be of good use when explaining basic concepts or when clarifying assumptions. The conceptual model for dynamic planning as presented in [23] is shown in Figure 4.1. The figure shows that the model consists of three parts:

- The *state transition system*, denoted by Σ, which is a formal model of the real-world system for which the plans should be created;
- The *planner*, which produces the plans; and,
- The *controller*, which performs the actions according to the plan, as to change the state of the system.

Figure 4.1 represents an *online planner*. Online planning is important for robotic planning, because the physical system of a robot is hard to model as a state-transition system, especially in a dynamic environment.

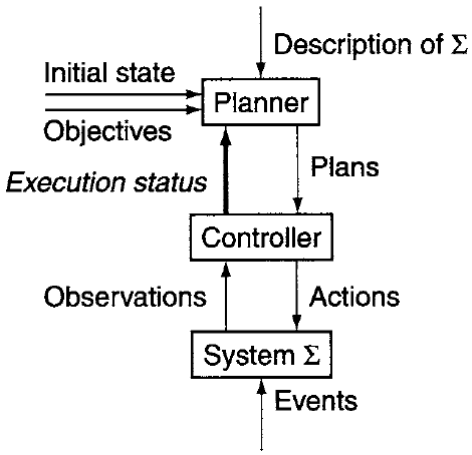

Figure 4.1: A conceptual model for dynamic planning, as presented by Ghallab et al. (2004) [23].

The distinction between an offline planner and an online planner is that an *offline* planner does not receive feedback about Σ's current state. It generates a plan which is then executed by the controller without any intervention from the planner. The plan cannot be revised in the case of unforeseen situations. There are differences between the physical system and the model (Σ) in most real-world applications. A link from the controller to the planner is required if the controller is unable to handle these differences. The planner then becomes an *online* planner, which is capable of dynamic planning. An online planner is informed of the execution status, and planning can be interleaved by introducing mechanisms for plan supervision, plan revision, and replanning.

4.1.2 Classical Planning

The simplest solutions are often the best. Therefore, classical planning is discussed here, because it is the simplest form of how a planning system can be considered. *Classical planning*, as Ghallab et al. describe in chapter 2 of their book [23], is concerned with planning problems for domains that can be modeled by a restricted state-transition system. A *restricted state-transition system* is the same as the state-transition system described above, but with the addition of the following restrictive assumptions:

Assumption A0 (Finite Σ). The system Σ has a finite set of states.

Assumption A1 (Fully Observable). The observations the controller receives concerning the current state of Σ provide complete information.

Assumption A2 (Deterministic Σ). The execution of a specific action in a particular state will always lead to the same result state.

Assumption A3 (Static Σ). The set of events E is empty, the system only goes into another state when the controller initiates a state transition by performing an action.

Assumption A4 (Restricted Goals). The planner objectives can only take the form of a specific goal state or a set of goal states.

Assumption A5 (Sequential Plans). The solution to a planning problem is a plan in the form of a linear sequence of actions. Also, no partially ordered plans are allowed.

Assumption A6 (Implicit Time). Actions have no duration, state transitions happen instantaneously.

Assumption A7 (Offline Planning). The planner works offline. The plan is generated based on the planning problem, and executed without any intervention by the planner.

The *restricted model* combines all eight restrictive assumptions. However, the restricted model is not applicable to robotic online planning. The previously stated assumptions need to be relaxed to enable applicability. This changes the restricted model into an *extended model*. This is further discussed in chapter 5, where some restrictions for the proposed planning system are relaxed in the design. The next sections

further discuss other planning techniques, since classical planning is not sufficient enough for robotic online planning.

4.1.3 HTN-Planning

HTN-planning is more advanced than classical planning, and can be designed as restrictive or extended as required. The basic idea behind HTN-planning is to recursively replace non-primitive tasks with appropriate task networks, and to instantiate primitive tasks with actions until there are only actions remaining. HTN-planning is discussed in chapter 11 of the book Automated Planning by Ghallab et al. [23]. This section builds upon that chapter. It explains HTN-planning in more detail and discusses several HTN-planning technologies.

The objective of an HTN-planner is not a set of *goal states*, but rather a collection of *goal tasks*. Tasks can be decomposed into subtasks by using domain-specific rules, called methods. These subtasks can then be decomposed further until primitive tasks are reached. The *primitive tasks* are ungrounded actions which can directly be executed by the controller, and they form the solution to the planning problem. *Non-primitive tasks* need to be decomposed into primitive tasks before they can be executed.

The primary advantage of HTN planners, compared to classical planners, is their sophisticated knowledge representation and reasoning capabilities. They can represent and solve a variety of non-classical planning problems. They can also solve classical planning problems faster than classical planners. The primary disadvantage of HTN planners is the need for the domain author to write both a set of planning operators and a set of HTN methods.

Control rule planners are similar to HTN-planners. Therefore, it is hard to say whether HTN planners or control rules are more effective. HTNs give a planner knowledge about what options to consider, and control rules give a planner knowledge about what options *not* to consider. As Bacchus and Kabanza [4] write, the two types of knowledge are useful in different situations and combining them is a useful topic for future research.

HTN-Planning Technologies

Several HTN-planners are developed, for example the Simple Hierarchical Ordered Planner (SHOP) [56] by researchers of the University of Maryland. SHOP, its successor SHOP2, and M-SHOP [57] (Multi-task-list SHOP, a modified version of SHOP) are written in Lisp. *SHOP2* [55] is an efficient planning system, which won one of the top four prizes in the 2002 International Planning Competition. There are other versions of SHOP, written in different programming languages. These versions are JSHOP, JSHOP2 [32], JSHOP2-RT [29] and Pyhop [59]. JSHOP and its family is written in Java and Pyhop is written in Python.

Both SHOP and SHOP2 [55] can do axiomatic inference, mixed symbolic/numeric computations and calls to external programs. However, SHOP2 also includes the following functionalities: partially ordered plans, the interleaving of subtasks, PDDL quantifiers and conditional effects, and temporal PDDL operators to handle temporal planning domains.

There are many other HTN-planners besides the previously discussed SHOP-like planners, some examples are: Nonlin [76], SIPE-2 [84], O-Plan [77], UMCP [14], and CHP [20]. Similar to SHOP, the planners SIPE-2, O-Plan and UMCP are all written in Lisp. Whereas Nonlin is written in POP-2, and CHP is written in C#.

4.1.4 Situation Calculus

The situation calculus is a logic formalism designed for representing and reasoning about dynamic domains. It is introduced by John McCarthy in 1963 [52]. Ghallab et al. [23] describe the situation calculus (in chapter 12 of their book) as a first-order language for representing states and actions that change states.

They continue to write that in situation calculus there is only one logical theory. Different states are represented by including the "name" of each state as an extra argument to each atom that is true in the state. For example, $at(r, l, s)$ denotes that $at(r, l)$ holds in the state s. The term s is called a *situation*. Terms and formulas containing a situation are called *fluents*. Fluents are functions and relations that can vary from one situation to the next. Every fluent is described with a *successor-state axiom*. Russell and Norvig [70] describe a successor-state axiom as an axiom that says what happens to the fluent, depending on what action is taken.

The successor-state axiom has the following form:

Action is possible ⇒
(Fluent is true in result state ⇔ Action's effect made it true
∨ It was true before and action left it alone).

Ghallab et al. [23] continue to say that, in contrast to classical planning, actions in situation calculus are represented as terms in the same first-order language $\mathcal{L}$ in which states are represented. The special function symbol *do* represents the situation resulting after the execution of an action. If α is a variable denoting an action, then $do(\alpha, s)$ is the situation resulting from the execution of α in situation s.

Action preconditions in situation calculus are also represented in the same language $\mathcal{L}$. The predicate *Poss* represents action preconditions. For example, the formula

$$\forall r \forall l \forall l' \forall s(Poss(move(r, l, l'), s) \leftrightarrow at(r, l, s))$$

states that action $move(r, l, l')$ is applicable to a situation s iff the robot is at location l in s. Such a formula is called an *action precondition axiom* or a *possibility axiom*.

There are many examples in which situation calculus is used as basis for a programming language. One such example is Golog [47] and its variants. These languages are also similar to HTN-planning, as abstract tasks are decomposed into more primitive subtasks.

MIndiGolog

Concurrent Golog (ConGolog), Incremental Deterministic (Con)Golog (IndiGolog), and Multi-Agent IndiGolog (MIndiGolog) are variants of the high-level agent programming language Golog. Of these Golog-variants, MIndiGolog is the most interesting, because it supports the control of multiple agents.

Multi-Agent IndiGolog (MIndiGolog) [39], developed by Ryan Francis Kelly, is a multi-agent variant of the high-level agent programming language IndiGolog, based on the situation calculus. It uses the Mozart Programming System for development.

The *Mozart Programming System* [54] is an advanced development platform for intelligent, distributed applications. It is based on the *Oz* language, which supports declarative programming, object-oriented programming, constraint programming, and concurrency as part of a coherent whole.

MIndiGolog 0 [40] is a preliminary implementation of MIndiGolog, written in Prolog rather than Oz. While it is only a centralized execution planner, and was not referred to explicitly in Ryan's PhD thesis [39], it may be of independent interest. This Prolog version of MIndiGolog shows more similarities with ConGolog than with IndiGolog. It gives the impression that Ryan Kelly adapted ConGolog to create MIndiGolog 0, as there are only a few differences. The main difference is that IndiGolog handles fluents differently.

4.1.5 Temporal Planning

The duration of an action can be of significant importance of a plan, especially when planning is done for multiple robots. Therefore, such a planning system is required to reason over time. Chapters 13 and 14 of Ghallab et al. [23] describe temporal planning. Temporal planning is essentially planning in which actions have a duration, instead of being instantaneous.

Actions occur over a time span in reality. Therefore, a finer model of an action should account not only for the preconditions *before* the action starts, but also for other conditions that should be in effect *while* it is taking place. Such a model should represent the effects of an action throughout its duration and the delayed effects after the action has finished. Also, goals in a plan are often only meaningful if they are achieved within a certain amount of time. Time in planning is required in two forms:

1. *qualitatively*, so the planner can handle synchronization between actions and with events (e.g. to plan for a robot to move to a destination that is not yet free), and

2. *quantitatively*, as a resources, so the duration of actions can be modeled with respect to deadlines or cost functions (e.g. to plan how to load or unload all the containers of a ship before its departure).

Also, temporal planning has to deal with two points of view, or issues:

1. The *causal point of view*, about what changes are entailed by an action and what conditions are required for the changes to hold consistently. Some conditions need to hold before an action can be executed, but also during the execution of an action.

2. The *temporal point of view*, about when other related actions can or cannot take place, as actions can be in conflict with each other.

The theory on temporal planning is quite involved and not further discussed here. However, both Goldman and Nau describe methods for adding temporal constraints to HTN-planning. These methods are shortly discussed next.

The paper Durative Planning in HTNs by Goldman (2006) [25] provides a detailed procedure of adding temporal constraints to HTN planning. Goldman encodes PDDL Level 3 durative actions in an HTN formalism compatible with the SHOP2 planner. This is based on Fox and Long's work [19], who extend PDDL for expressing temporal planning domains. Goldman writes that "the semantics of a PDDL Level 3 plan can be described in terms of a sequence of time points: the start and end points of the durative actions." Every action has a start-act and an end-act operator, and protections are added for each *'over all'* condition, such that certain conditions can hold during the execution of an action.

Nau describes an alternative approach, named Multi-timeline preprocessing (MTP), in [55]. Goldman already provides a comparison between his use of PDDL semantics and Nau's MTP [25]: "in MTP, durative actions are not split into begin and end pairs, and protections are not used. Instead, additional read- and write-time fluents are associated with base fluents, and these are manipulated, instead of manipulating a time fluent." Nau uses two additional parameters for every operator, a *start* and *duration*. The start parameter is a time-stamp denoting the start of an action, whereas duration is the duration of the action.

Even though Goldman and Nau made methods specifically for HTN-planning, the methods could be adapted to other planning systems, such as situation calculus. MIndiGolog [39] handles temporal planning is a similar way as Goldman and Nau describe their methods.

4.2 Semantic Web

The *Semantic Web* [79] proposes to help computers "read" and use the Web. It is a collaborative movement led by the international standards body, the *World Wide Web Consortium* (W3C). The term Semantic Web, which is coined by Tim Berners-Lee (the director of the W3C) in 2001 [5], refers to W3C's vision of a "Web of (linked) data." The idea is simple: meta-data added to Web pages can make the existing World Wide Web (WWW) machine-readable. This gives machines tools to "find, exchange, and interpret" information. It is thus an *extension* of the WWW.

A robot connected to the Semantic Web can find information about other robots on the Web. If all robots provide their services online, a planning system could use this information to find a robot with specific capabilities to perform a task. It is therefore useful to research semantic web technologies.

The act of reasoning over data through rules is called 'inference'. Inference is found near the top of the Semantic Web stack. W3C focuses their work on rules on translating between rule languages and exchanging rules among different systems. They do this mainly through Rule Interchange Format (RIF) and OWL. OWL is also used to build ontologies. Ontologies and knowledge organization systems organize data, but also enrich data with additional meaning, allowing more people and machines to do more with the data. This section continues to discuss web ontologies in more detail next, followed by the Web Ontology Language (OWL) and Web Ontology Language for Services (OWL-S).

Web Ontologies

An ontology in the fields of computer science and information science is a representation of knowledge as a set of concepts within a domain, and the relationships between pairs of concepts. It is used to model domains, and to support reasoning about concepts. Web ontologies are ontologies specifically made for and used by the Semantic Web. Ontologies are the basic building blocks for inference techniques on the Semantic Web.

Ontologies on the Semantic Web help with data integration. [79] Data integration involves combining data from different sources and providing a unified view of these data. In the case of Web ontologies, this is done when, for example, ambiguities exist on the terms used in the different data sets, or when a bit of extra knowledge leads to the discovery of new relationships. There is however some trouble with ontologies: they are very difficult to create, implement and maintain. Depending on their scope, they can be enormous, defining a wide range of concepts and relationships.

Web Ontology Language

In computer science and artificial intelligence, ontology languages are formal languages used to construct ontologies. They allow the encoding of knowledge about specific domains and often include reasoning rules that support the processing of that knowledge. Ontology languages are usually declarative languages, they are almost always generalizations of frame languages, and are commonly based on either first-order logic or on description logic.

The *Web Ontology Language* (OWL) [53] can be seen (as its name obviously says) as such a Web ontology language. OWL is endorsed by the W3C. It is actually a family of knowledge representation languages, characterized by formal semantics and RDF/XML-based serializations. OWL is available in three levels of complexity: *Lite*, *Description Language* (DL) and *Full*. There are also two specifications: OWL (from 2004) and OWL2 (from 2009). Protégé is one of the most used ontology editing tools for OWL.

Web Ontology Language for Services

The Web Ontology Language for (Web) Services (OWL-S) [51] is an ontology within the OWL-based framework of the Semantic Web, for describing Web services. It enables users and software agents to automatically discover, invoke, compose, and monitor Web resources offering services, under specified constraints.

Within OWL-S, the decomposition of composite processes is specified by using control constructs. [51] *Composite processes* are processes that can be decomposed into other (non-composite or composite) processes. A composite process is not a behavior a service will do, but a behavior (or set of behaviors) the client can perform by sending and receiving a series of messages. A non-composite process is also called an atomic process. An atomic process is known as a primitive task (a task that can be directly executed) in planning.

There are a total of nine OWL-S control constructs, these are:

1. **Sequence**. A sequence is a list of control constructs to be done in order.

2. **Split**. The components of a split are a bag (collection) of process components to be executed concurrently. Split completes as soon as all of its component processes have been scheduled for execution.

3. **Split+Join**. The split+join consists of concurrent execution of a bunch of process components with barrier synchronization. Split+join completes when all of its component processes have completed.

4. **Choice**. The choice construct calls for the execution of a single control construct form a given bag of control constructs. Any of the given control constructs may be chosen for execution.

5. **Any-Order**. The any-order allows the process components (specified as a bag) to be executed in some unspecified order, but not concurrently. Similar to a sequence, execution and completion of all components is required.

6. **If-Then-Else**. The if-then-else construct executes the first program (referred to as 'then') if the given condition is true, or else it executes a second program (referred to as 'else').

7. **Iterate**. The iterate construct makes no assumption about how many iterations are made or when to initiate, terminate, or resume. Iterate is an "abstract" class, in the sense that it is not detailed enough to be instantiated in a process model. It is defined to serve as the common superclass of constructs such as repeat-while and repeat-until. Because of its abstract nature, iterate does not need to be represented in the planner. It is only mentioned here for completeness.

8. **Repeat-While**. The repeat-while iterates until a condition is false. Repeat-while tests for the condition, and exits if it is false. If the condition is true, it does the operation and then loops.

9. **Repeat-Until**. The repeat-until iterates until a condition is true. Repeat-until does the operation, tests for the condition and then either exits if the condition is true or loop otherwise. Thus repeat-while may never act, whereas repeat-until always acts at least once.

Furthermore, the OWL-S ontology has three main parts:

- The *service profile*, which is used to describe a service's functionality. It is primary meant for human reading. It includes the service name and description, limitations on applicability, quality of service, publisher information, and contact information;

- The *process model*, which describes how a client interacts with the service. This includes the sets of inputs, outputs, pre-conditions, and post-conditions of the service execution; and,
- The *service grounding*, which specifies the details a client needs to interact with the service. This includes communication protocols, message formats, and port numbers.

4.3 Robotics

This section presents an introduction to the robotic technologies that are of importance to the project. It discusses the Robot Operating System, the RoboEarth project, and ubiquitous robotics. Ghallab et al. [23] write about robotics that it is a reasonably mature technology, but only when robots are restricted to operating within well-known and well-engineered environments (e.g. manufacturing robotics), or to performing single simple tasks (e.g. vacuum cleaning or lawn mowing robots). Robotics remains a very active research field when robots need to handle more diverse tasks and dynamic environments. This is still true today. Many robots use the Robot Operating System (ROS), a generic framework for robots. Among these robots is the *Autonomous Mate for IntelliGent Operations* (AMIGO), a robot developed by Tech United, the RoboCup Team of the TU/e. Therefore, this section starts by explaining the ROS framework. Then the RoboEarth Project is discussed, followed by the concept of Ubiquitous Robotics to finish this section.

4.3.1 Robot Operating System

The *Robot Operating System* (ROS) [18] is a software framework for robot software development, which provides functionalities similar to an operating system on a heterogeneous computer cluster. It is created and maintained by Willow Garage. This framework provides libraries and tools to help software developers create robot applications. It is written on the ROS website [18] that ROS provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more. ROS is licensed under an open source, BSD license. The goal of ROS is to enable software developers to build more capable robot applications quickly and easily on a common platform. The robots of the TU/e, and many other robots today, use the ROS framework. It is used both in research as well as in commercial applications of robots.

ROS has three levels of concepts: the *Filesystem level*, the *Computation Graph level*, and the *Community level*. The filesystem level concepts are ROS resources that are encountered on hard disk, such as packages (the main unit for organizing software in ROS) and message descriptions. The Computation Graph is the peer-to-peer network of ROS processes which are processing data together. The basic concepts provide data to the Graph in different ways, for example nodes (computation processes) and messages. The ROS Community Level concepts are ROS resources that enable separate communities to exchange software and knowledge, such as code repositories, the ROS Wiki, the ROS Answers Q&A site, and the Willow Garage Blog. The ROS website [18] gives a more detailed explanation.

4.3.2 RoboEarth Project

The *RoboEarth project* [66] is initiated by a multi-disciplinary partnership of robotics researchers from academia and industry. It is funded by the European Commission's Cognitive Systems and Robotics Initiative. The goals of the project are:

1. to prove that connection to a networked information repository greatly speeds up the learning and adaptation process, that allows robotic systems to perform complex tasks, and
2. to show that a system connected to such a repository will be capable of autonomously carrying out useful tasks that were not explicitly planned for at design time.

RoboEarth is meant to be a WWW for robots, that is, a giant network and database repository for robots to share information and to learn from each other about their behavior and environment. The project allows robotic systems to benefit from the experience of other robots, and enables progression in machine cognition and behavior. RoboEarth's WWW-style database stores knowledge, generated by both humans and robots, in a machine-readable format.

RoboEarth offers a Cloud Robotics infrastructure, known as the *RoboEarth Cloud Engine* (or *Rapyuta*) [31], which includes "everything needed to close the loop from robot to cloud and back to the robot." This Cloud Engine makes powerful computation available to robots and is specifically targeted for the

communication between ROS enabled systems. It allows robots to offload their heavy computation to secure computing environments in the cloud with minimal configuration.

Cloud robotics is an emerging field of robotics, rooted in cloud computing, cloud storage, and other Internet technologies centered around the benefits of converged infrastructure and shared services. It allows robots to benefit from the powerful computational, storage, and communications resources of modern data centers. Additionally, it removes maintenance and update overhead, and reduces dependence on custom middleware.

RoboEarth offers components for a ROS-compatible, robot-unspecified, high-level operating system, as well as components for robot-specific, low-level controllers accessible via a "Hardware Abstraction Layer", as seen in Figure 4.2.

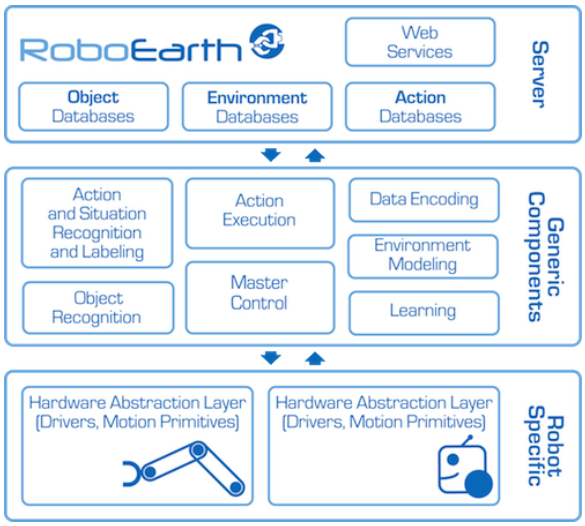

Figure 4.2: RoboEarth Components, as presented on the RoboEarth website (2013) [66].

4.3.3 Ubiquitous Robotics

Ubiquitous computing is first explained before going into the concept of ubiquitous robotics. Mark Weiser explains the concept of ubiquitous computing in a paper from 1991. [81] Ubiquitous computing comes after mainframes and personal computing as the third era of computing, and is meant to achieve the real potential of information technology. The vision of personal computers and laptops that are used today, is only a transitional step to achieve this potential, because such machines cannot truly make computing an inherent, invisible part of everyday life. Ubiquitous computing is described as another way of thinking about computers in the world, one that takes into account the natural human environment and allows the computers themselves to disappear into the background, so that people cease to be aware of it. Mark Weiser calls Ubiquitous Computing the "invisible" path of computer and interface design [82], he writes (on his website): "its highest ideal is to make a computer so imbedded, so fitting, so natural, that we use it without even thinking about it."

Jong-Hwan Kim et al. [43] see a similar trend in the evolution of robotic technology. By this reasoning, they call the *ubiquitous robot* the 'third generation of robotics', which comes after the industrial robot and personal/service robot. The concept describes various robots in a networked environment, where the intelligence lies within the network and not in the robots. Every actuator, sensor or (control) software component in a networked environment, can be seen as part of an ubiquitous robot, since those are the three main components of a robot. Just as in ubiquitous computing, the robot vanishes into the background, and people naturally interact with an intelligent robotic environment.

This section describes several ubiquitous robotics systems and projects. The systems and projects discussed are: Ubibot, Ubiquitous Robotic Companion, PEIS-Ecologies, Rubicon, Robot-Era, UPnP Robot Middleware. This section also provides a short section with material for further reading.

Ubibot

Jong-Hwan Kim et al., who are researchers at Robot Intelligence Technology (RIT) lab., KAIST, describe their idea of an ubiquitous robot as an "Ubibot" [44], "a robot incorporating three forms of robots: software

robot (Sobot), embeded [sic] robot (Embot) and mobile robot (Mobot), which can provide us with various services by any device through any network, at any place anytime in a ubiquitous space." [43] These terms are rather ambiguously, a *Sobot* is actually a proactive intelligent software agent meant for robotic control. An *Embot* is any sensor device and a *Mobot* is any actuator device. KAIST is the abbreviation of Korea Advanced Institute of Science and Technology, a South Korean public research university.

Jong-Hwan Kim et al. proposed the *ubiquitous robot* (Ubibot) system [44]. The Ubibot system contains a TCP/IP-based middleware structure, implemented in Visual C++ 6.0, and containing HTTP, FTP and socket interfaces. Users can give commands to the system through voice or using keyboard and mouse. A *"Sobot"* called Rity is used for experiments, which has a virtual environment, perception module, learning module, internal state module, behavior module, motor module. No detailed implementation of Rity is provided. Taehong Kim et al. [45] describe the communication module of a URS system in more detail. This communication module includes algorithms for link failure detection, network discovery, and robots requesting to rejoin the network.

Furthermore, In-Bae Jeong and Jong-Hwan Kim [37][42] expand on the middleware of the Ubibot system, creating a multi-layered architecture, with the intend to increase modularity and scalability. This is the first time that a task scheduler is mentioned for the Ubibot system, but specific details are not described.

Ubiquitous Robotic Companion

The *"Ubiquitous Robotic Companion"* (URC) [26][27] is a concept by the researchers of RIT lab., KAIST. Young-Guk Ha et al. present the *Service-oriented Ubiquitous Robotic Framework* (SURF) [26], which enables "automated integration of networked robots into ubiquitous computing environments" based on the OWL-S technology. All robot interfaces, networked sensors and other devices are implemented as Web Services in this framework. A SURF agent can automatically discover required knowledge in a knowledge base and compose feasible service plans. The agent also controls devices through the Simple Object Access Protocol (SOAP). The SURF agent consists of a user interface, plan composition module, knowledge discovery module, plan execution module, and a communication stack for Web Services including SOAP, XML and HTTP. Both UMCP [14] and SHOP2 [55] are used in the planner. It is not fully clear how devices are presented to the knowledge base and how knowledge discovery and parsing is executed. They do provide pseudo-code for a semantic service discovery algorithm [27]. The specific SHOP2 planning algorithm is not described in their papers either. Executable Web Service processes are generated by use of the Web Services Business Process Execution Language for Web Services (WSBPEL) [62]. SURF is later renamed to *Semantic-based Ubiquitous Robotic Space* (SemanticURS) [27].

PEIS-Ecologies

Mathias Broxvall et al. [9] have a different name for ubiquitous robotics, they call it *"PEIS-Ecologies"*. They describe a ubiquitous robot system as an "ecology of robots". Saffiotti and Broxvall [72] combine the vision of a skilled robot companion with the vision of a network of intelligent home devices. This results in the concept of an ecology of networked *Physically Embedded Intelligent Systems* (PEIS). A PEIS is any physical device with functional components. A *PEIS-Ecology* (already mentioned in section 4.3.3) is a network of cooperating PEIS. Saffiotti and Broxvall write that "a PEIS can use functionalities from other PEIS in the ecology in order to compensate or to complement its own." The power of the PEIS-Ecology comes from the ability to interact and cooperate. Three factors are named that characterize a PEIS-Ecology, these are:

1. The first factor is that there may be a large number of PEIS exchanging data;
2. The second factor is that PEIS can join and leave the ecology at any moment, and they can be temporarily unavailable (e.g., out of radio range); and
3. The third factor is that some interactions can be tightly coupled.

A reference architecture, that enables the communication and coordination between all the components in a PEIS-Ecology, is needed to implement such an ecology. Saffiotti and Broxvall's reference architecture combines an event-based and a tuple-based communication model within a component framework. The PEIS-Ecology includes a PEIS-kernel, which takes the form of a Linux run-time library. This library performs "network discovery, creates and maintains ad-hoc P2P network over TCP/IP, and takes care of providing a shared tuple-space between the different PEIS." [72] PEIS uses the sensor-based probabilistic action planner PTLplan [38] for high-level task planning.

Saffiotti et al. describe in another paper [71] that both a centralized and a distributed approach is used in the PEIS-project to generate configurations for tasks. The centralized plan-based approach uses a global hierarchical planner, while in the distributed reactive-based approach, the configurator creates a local configuration, and assumes that the connected PEIS can recursively extend this configuration. Saffiotti et al. [71] say that the "plan-based approach is guaranteed to find the optimal configuration if it exists, but it has problems to scale up and it cannot easily cope with changes in the ecology."

After the PEIS-Ecology project stopped in 2009, several other projects started to explore the concept of PEIS-Ecology further. This includes the project Rubicon [69] and Robot-Era [67].

Rubicon

The *Robotic UBIquitous COgnitive Network* (RUBICON) [69]. is a self-sustaining, self-organizing, learning and goal-oriented robotic ecology, based on the concept of PEIS. It expanded on PEIS by changing the control layer into a multi-agent system (MAS), based on BDI (Believe-Desire-Intention) agents. The two key goals of the 'Control Layer' within RUBICON are to provide sensing and actuating services that are adaptable and robust. ROS is used as hardware access provider. A mix of agent-planning integration techniques are used. Neural networks are used for the learning mechanism of the system.

Robot-Era

The objective of the *Robot-Era* project [67] is to develop, implement and demonstrate the general feasibility, both scientific and technical effectiveness, both social and legal plausibility, and acceptability by end-users of a plurality of complete advanced robotic services, integrated in intelligent environments. These robotic services cooperate with real people to favor independent living, improve the quality of life and the efficiency of care for elderly people. This project is very recent, the first experiments started in July 2013. Robot-Era is also based on PEIS. The project is still ongoing and not much details are publicly published yet.

UPnP Robot Middleware

The *Universal Plug and Play* (UPnP) Robot Middleware (Sang Chul Ahn et al. [1]) is another proposed framework for ubiquitous robot control. It offers pervasive peer-to-peer network connectivity of intelligent appliances in dynamic distributed computing environments. Ahn et al. show that internal software integration of robots and robot services in ubiquitous computing environments are the same thing. The paper by Ahn et al. [1] shows that UPnP has good performance by comparing it with TAO CORBA. *Common Object Request Broker Architecture* (CORBA) is a middleware architecture of the Object Management Group's (OMG) standard for language and platform independent distributed computing. TAO, acronym for *The ACE ORB*, is an open-source high-performance, real-time CORBA ORB end system. It is developed for distributed real-time and embedded systems by the TAO project. Ahn et al. further show that the UPnP technology could provide a unified framework by describing the usefulness of the UPnP and by showing some experimental result regarding it. The experiments show that the UPnP components can be easily and dynamically loaded, unloaded and reloaded. Every module on a robot and every device can be a UPnP component. Ahn et al. claim that the UPnP technology has good features for ambient home environments. UPnP provides a wide set of standard protocols and web technologies, such as TCP/IP, UDP, SSDP, SOAP, GENA, HTTP and XML. The UPnP Robot Middleware is, as it name says, only middleware, therefore it does not include planning, executive or ontology technologies.

Further Reading

The modularity of robot components is also present in Sang Chul Ahn's description of ubiquitous robotics. Sang Chul Ahn et al. [1] write that the background idea of their approach is that a robot system itself needs to be a dynamically distributed computing environment. They continue to write that "every component of a robot will become modular, and they will be configured dynamically in the future. Another point of view is that robots will be deployed at home or at office like a TV or a refrigerator in the future. And they will have to dynamically join and communicate with home or office networks as home and office are expected to be ubiquitous computing environment in the near future." [1]

A. Yachir et al. [85] mention two requirements of a ubiquitous robotic framework which they think are essential, these are: 1. "the ubiquitous robotic system must support ubiquity and heterogeneity of services, sensors and devices" and 2. "the services must always be available even though there are changes in the service and user situation and/or context."

Jong-Hwan Kim et al. [41] actually describe five generations of robotic technology (Figure 4.3). These are: industrial robots, service robots, ubiquitous robots, genetic robots, and bio robot. An *industrial robot* is an automatically controlled, reprogrammable, multipurpose manipulator, most commonly used in manufacturing systems. There are many kinds of service robots. The robotic vacuum cleaner is a perfect example of a service robot which is already in use in many homes. *Service robots* are robots which assist people with chores at home or at their work. They can also be tour guides [10][61], fire fighters [86], rescue robots [6] or surveillance robots [46]. Another example of a service robot is the AMIGO robot by Tech United.

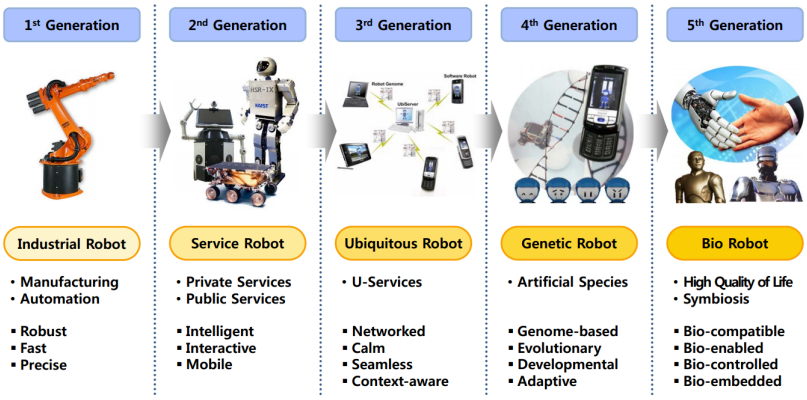

Figure 4.3: Five generations of robots, as presented by Jong-Hwan Kim (2013) [41].

4.4 Comparison of Technologies

This section provides a short summary of the previously presented technologies and theories from literature, but also compares these technologies with each other, to show why they are good for the project's intended purpose. The emphasis here is more on the technologies discussed than the theory.

Automated planning, the Semantic Web and robotic technologies are all described in this chapter, but it is also described how these technologies can be connected with each other. The section about automated planning discussed the conceptual model of planning, classical planning, HTN-planning, situation calculus and planning techniques such as temporal planning and planning with resources. The section on the Semantic Web discussed web ontologies and web ontology languages. The robotics section discussed ROS, the RoboEarth project and ubiquitous robotics.

This section starts with comparing planning techniques with each other, followed by comparing robotic systems with each other.

The comparison of the planning techniques includes: 1. a comparison between HTN, situation calculus, and control rules; and 2. a comparison between specific planners.

Planners can be based on (but not limited to) hierarchical task networks, situation calculus and control rules. All these methods are very similar to each other as they all start with a planning domain and a planning problem (or goal task) and they decompose abstract tasks into smaller subtasks. The difference between HTNs, situation calculus and control rules, is that HTNs and situation calculus give a planner knowledge about what options to consider, whereas control rules give a planner knowledge about what options not to consider. There are more HTN-planners available than both control rule planners and situation calculus-based planners.

If one compares, for example, SHOP2 [55] with MIndiGolog [39], it is clear to see that SHOP2 is based on HTN-planning and MIndiGolog is based on situation calculus. They both decompose composite tasks into subtasks and are able to handle temporal constraints. MIndiGolog however, also has support for multi-agent planning.

The comparison between robotic systems includes a comparison between various robotic systems, as presented in the ubiquitous robotics section.

There are already several ubiquitous robotic systems. Most of these systems divide robots into actuator, sensor and software components, where each component can be accessed separately by other modules and systems. The most notable systems are the *'Ubibots'* [43][27] and the *'PEIS-Ecologies'* [9]. The Ubibot project is a closed source project, but the PEIS-Ecology project is open source (available as alpha release).

The PEIS-Ecology project is therefore more interesting than the Ubibot project. Ubibot uses SHOP2 as task planner and PEIS uses a sensor-based probabilistic action planner called PTLplan (PTLplan [38] is inspired by Bacchus and Kabanza's TLplan [4]). PEIS also includes a Gazebo simulator, but Ubibot only has a basic (2D) simulation.

Design choices can be made next, based on the comparisons made here. The next chapter describes the design of the proposed system.

Chapter 5

Design

This thesis proposes a design of a planning module, which is part of a larger system (as described in chapter 2). This planning module is an adaptation of the MIndiGolog language. The planning module is designed with the thought of keeping things simple, as a planning system for multiple robots can become very complex. This choice can be seen throughout the whole design.

As written in section 4.1.4, *MIndiGolog* is a planning language based on situation calculus and written in Prolog. MIndiGolog is chosen over other planning technologies, because MIndiGolog supports both concurrency and temporal planning, which is essential for multi-robot planning. Even more, because it is written in Prolog, it enables the planner to reason about knowledge in a logical formalism. This matches with both situation calculus and web service ontologies. Also, there are technologies already available to connect Prolog to ROS Python (e.g. PySWIP [78]) and to parse web ontologies in the language OWL-S to Prolog.

Before the planning process can be discussed, it is important to consider the planning domain. Automated planning for real world applications can get very complex. It is therefore necessary to carefully consider the assumptions that can be made regarding the domain. These assumptions are addressed after the next section, which discusses what adaptations are specifically made to MIndiGolog, so that the design covers the previously presented requirements.

5.1 Contributions

Section 2.2 already listed the contributions the planning module needs to incorporate. The choice of MIndiGolog as planning language allows for these contributions to be more specific. This section lists the contributions from a design point of view. Furthermore, it connects the design to the requirements as presented in section 2.3 and appendix A.

The contributions of the planning module are listed as follows:

1. The extension of *OWL-S control constructs* in MIndiGolog,
2. The adaptation of how *temporal planning* is handled in MIndiGolog,
3. The extension of *multi-robot control* in MIndiGolog,
4. The addition of *robot capability matching* in MIndiGolog, and
5. The adaptation from the offline MIndiGolog planner into an *online planner*.

These contributions are described in more detail in the following sections. Specific implementations of these contributions are described in the next chapter.

5.1.1 OWL-S Control Constructs

Section 4.2 describes that there are nine OWL-S control constructs. Prolog has the capability to represent these OWL-S control constructs as transition and termination predicates. The language MIndiGolog already has transition and termination predicates to change its internal state and compose plans, based on certain constructs [24][39]. Even though some of these MIndiGolog constructs are already similar to OWL-S control constructs, MIndiGolog is not made specifically for OWL-S control constructs. The transition and

termination predicates in MIndiGolog are based on the application of situation calculus, regression and action theory to planning, as described by Reiter [65] and De Giacomo [24].

This design proposes that OWL-S-specific transition and termination predicates are included in the planner, to meet requirement F4 (the planning module uses OWL-S control constructs) and so that it can reason over robots represented as web services. This is also linked to requirement F5 (The planning module handles all data that is parsed by the OWL-S Ontology Layer). It is also proposed that certain OWL-S constructs (e.g. sequence, split, choice) can have any number of parameters. The planner needs a generic function to handle these control constructs. Section 6.2 describes the implementation of these predicates.

5.1.2 Temporal Planning

In order to meet requirement F2 (the planning module handles time and duration of actions) the planner reason about time. Therefore, all actions need to incorporate a duration. Sometimes the duration of actions is based on other factors, for example, the duration depends on distance when a robot has to drive to a location. It is proposed to include distance calculation in the planner based on locations available in the knowledge base.

Temporal planning in MIndiGolog is extended with distance calculations and selecting the cheapest plan from a set of valid plans, which also links it to requirement F1 (the planning module finds the most optimal plan (with the given knowledge), if a plan exists). Distance calculation between robots and locations is included to determine the expected execution time of a plan. The planner must know where the robots are for the distance calculation to work. The robot locations must be tracked. This information needs to be updated inbetween planning steps, but only if a robot's location is changed. The planner finds multiple plans, and it selects the cheapest plan from this set of plans.

Temporal planning in MIndiGolog is also simplified, natural actions are not used in the current prototype. Natural actions are actions of which the planning system has no control of, and thus depends on external factors. Examples of natural actions are actions performed by humans in the environment or the recharging of a robot (which is time-dependent). Furthermore, in order to keep temporal planning simple, the design is based on Nau's Multi-timeline preprocessing (MTP) [55] rather than Goldman's use of PDDL semantics [25]. Therefore, actions are not divided into a start and stop action. Section 6.3 describes the implementation of temporal planning.

5.1.3 Multi-Robot Control

MIndiGolog already handles concurrency and multiple agents. This is extended to be applicable for multi-robot control, to meet requirement F3 (The planning module handles concurrency of actions) and requirement F6 (the planning module provides plans which can be executed by the Execution Layer). The planner can produce plans for multiple robots simultaneously and plans with concurrency. Also, the planner selects the amount of robots it needs based on available robots and number of tasks. Section 6.4 describes the implementation details of multi-robot control.

5.1.4 Robot Capability Matching

The design proposes to include robot capability matching. This is linked to both requirement F5 (the planning module handles all data that is parsed by the OWL-S Ontology Layer) and requirement F7 (the planning module uses web service composition to find available robots). The planner chooses robots for its plan based on robot capabilities. This means that not all robots can do the same task as any other robot. If, for example, a robot has an arm with a gripper, its robot capability is then described as "has a gripper". These robot capabilities are described within an ontology. This increases the genericness and expandability of the system.

The planner uses robot capabilities as preconditions and as parameters to primitive actions. For example, if the primitive action is to pick up some item, then only a robot with an arm can perform this action. The robot capabilities work on actuator/sensor level. This means that the same actuator or sensor cannot be used for more than one task at the same time. Therefore, robot capabilities also create action conflicts within the planner.

Section 6.4 describes the implementation of robot capability matching.

5.1.5 Online Planner

MIndiGolog is turned into an online planner. The planner contains a feedback loop with the executive and keeps its backtracking capabilities. The planner plans a whole plan in an offline-manner (i.e. backtracking) every time the executive requests a new action from the plan. This is called replanning. Replanning is required to handle the dynamic environment for which the planner needs to plan. The environment is too complex to model. The planning module needs current knowledge about the robot environment to continue to produce valid plans. Section 6.1 discusses the adaptation of MIndiGolog to an online planner.

5.2 Planning Domain

This section describes the planning domain and discusses which assumptions about the planning system must be restricted and which must be relaxed to meet the previously defined requirements. The analysis of the planning domain is important for making a decision on the planning technology, and it also provides a theoretical basis for the planning system. These assumptions strengthen the choice for MIndiGolog as a planning language for the system. Section 4.1.2 describes the (restrictive) assumptions [23] to be made for a planning system to clearly define the planning domain. Some assumptions need to be relaxed, this is discussed for each assumption separately.

Assumption A0 (Finite Σ). This assumption does not need to be relaxed and remains restrictive because the planner can always terminate. Every path the planner takes either ends in failure or termination of the plan.

Assumption A1 (Fully Observable). This assumption is not relaxed, as full observability is assumed for the system. The planner does not reason, for example, about the probabilities of certain states.

Assumption A2 (Deterministic Σ). The real-world is nondeterministic, this means that the execution of a specific action in a particular state does not always lead to the same result state. Therefore, the assumption that the system is nondeterministic needs to be relaxed. The planner is designed to replan for every next action request by the executive, but does not take other measures to deal with nondeterminism. Thus the planner will repair its plan in some cases (depending on the design of specific task procedures and on the situation), due to the constant replanning.

Assumption A3 (Static Σ). This assumption holds as transitions are only performed in the planner by actions controlled by the system. External events (such as humans moving around) is not taken into account while planning.

Assumption A4 (Restricted Goals). The assumption of restricted goals remains restrictive. The planner only accepts simple goal tasks (i.e. tasks which need to be accomplished) and does not handle goals that need to be maintained.

Assumption A5 (Sequential Plans). This assumption needs to be relaxed in order to handle concurrency. A planning system for multiple robots needs to produce plans where robots can execute actions in parallel. Also, in some cases, the order of actions does not matter.

Assumption A6 (Implicit Time). The assumption that actions have no duration obviously has to be relaxed to allow temporal planning and to make the planner able to find the cheapest plan from a set of possible plans. It is explained in the previous section how temporal planning is designed for the proposed system.

Assumption A7 (Offline Planning). An online planner is assumed. The proposed system is designed as an online planner such that it checks at every request from the executive if the plan is still valid. MIndiGolog is adapted from an offline planner to an online planner in the proposed system. Even though MIndiGolog already has a predicate for online planning, this predicate does not support backtracking over plans (i.e. once it selects an action for a task, it is permanent, and it cannot go back to select a different action for that task) and thus does not guarantee that it can find a valid plan. The proposed design includes online planning with backtracking capabilities.

5.3 Planning Process

Once a task is given by a user, and the task is recognized by the system, the executive queries the planning module. The planner accepts an action URI, such as *'https://roboticssrv.wtb.tue.nl/svn/ros/user/rob/owl-s/knowledge.owl#objectDetection'(_)*, as input. After receiving this input, the planner checks how many robots are available to help with accomplishing the goal task. Based on the number of robots, and details of the goal task (e.g. amount of order locations when the goal task is to take orders from people), the planner decides how many robots are needed and generates a procedure for which a plan needs to be produced. After deciding how many robots are required, the planner searches a plan to execute the user-given task. The process of the planning module (also shown in figure 5.1) is then as follows:

1. The planner receives a goal task as input,
2. The planner matches the task with a known procedure,
3. The planner tries to find a plan for the procedure:
   (a) The planner first checks if the procedure can be terminated. If it cannot be terminated, the planner checks if a transition to another state is legal. If it can be terminated, the planner ends the plan.
   (b) The planner checks in the ontology for robot capabilities when selecting robots for performing actions in a plan.
4. The planner searches for a multitude of plans.
5. Then the planner selects the cheapest plan, based on the criteria of shortest expected execution time of the plan.
6. Then the planner gets the first action of this plan, which can then be used for further purposes (i.e. used by the executive to control the robot(s)).

The action the planner produces for the executive is in the form of a URI or, in the case of concurrency, a list of URIs. This URI is similar to the URI that is given as input to the planner, with the exception that it always represents a primitive task (an action that can be directly executed).

After the executive finishes executing an action, it requests a new action from the planner. The planner remembers where it left off in the plan, and starts to replan from there. The replanning process creates some robustness in the planner and is the reason why the planner keeps its backtracking capabilities. The choice for including a replanning process in the planner is made because the planner needs to handle a dynamic environment.

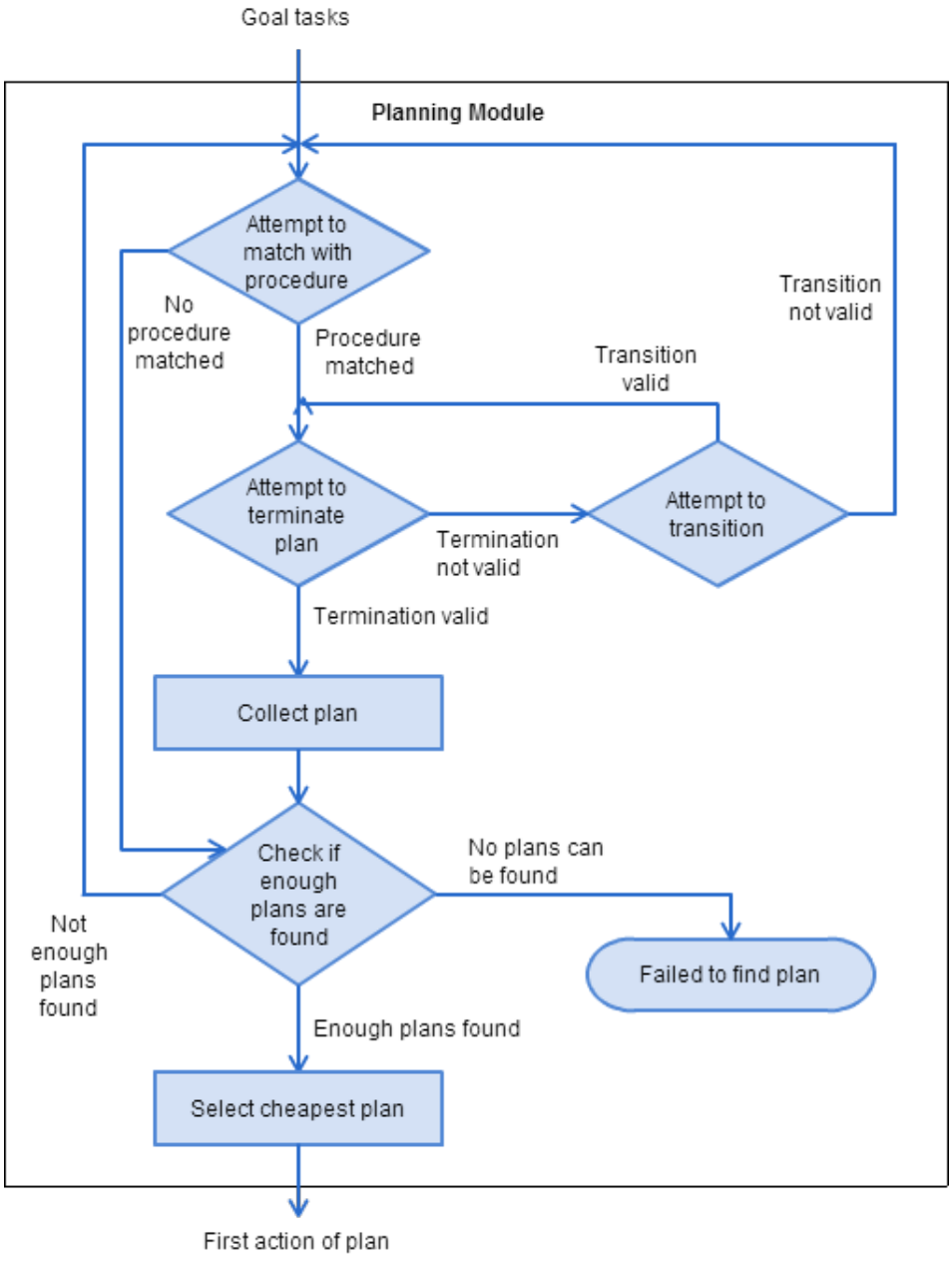

Figure 5.1: Planning Process

Chapter 6

Prototype

A prototype is implemented to test the design of the proposed planning system. This chapter discusses this prototype and the choices that were made during the implementation. The focus lies on the planning module of the system. For the implementation of the executive and ontology part of the system (as presented in chapter 2), see the work of Janssen [35], Geerts [21] and Denissen [13]. Their work show that the executive is in ROS Python, the ontology is in OWL-S and developed with Protégé, and that Rapyuta is also used in the implementation for communication between various ROS components.

This chapter explains the implemented prototype of the proposed framework in more detail. The prototype is an implementation of the framework discussed in the previous chapter. The framework is however not fully implemented due to time and resource constraints. The next section describes the process flow. This is followed by transition and termination predicates, temporal planning, and mechanisms for multi-robot control.

There are short code listings provided in this chapter, however, for the full source code, the reader is referred to appendix B.

6.1 Process Flow

The process flow of the prototype is divided in two parts: the process of starting a new plan, and the process of getting the next action of a plan. They only differ in the beginning of the process. The general part is explained in more detail after the start of both processes are discussed. The process flow is also shown in figure 6.1. In addition, this section discusses the implementation of online planning and replanning. First the process of starting a new plan is described.

A plan needs to be initiated by the executive to start the planning process. The process of starting a new plan is as follows:

1. On initiation, data from a previous plan is removed so a new plan can be formed.
2. All procedures previously generated by the planner are cleared.
3. The amount of required robots is determined. (This is described in section 6.4)
4. Then the planner searches for a plan for the given goal-task. (See the generic part below)

Later, when the executive asks for the next action, the planner continues from the point in the plan where it previously left off. The process of getting the next action of a plan is as follows:

1. First, the remainder of the program, that is to be executed, is retrieved.
2. The remainder is removed from the database, because the information is already retrieved.
3. If necessary, some flags could also be removed at this point. (These flags make planning with while-loops possible, see section 6.2 for more detail)
4. Then the planner searches for plan for the remainder of the given goal-task. (See the generic part below)

The general part of the processes, in which an actual plan is formed, is as follows:

1. First, the planner searches for the cheapest plan for the given goal-task. This is further referred to as the planner's *'offline planning mode'*.
   (a) The planner takes all steps of the plan into account.
   (b) The planner searches for valid transition and termination steps, such that the plan can be terminated.
   (c) The planner searches for a multitude of plans.
   (d) The planner then compares the plans to each other and selects the cheapest plan.
2. The planner gets the first action of the plan. The first action of the plan is also the return value for the executive.
3. The planner updates its knowledge base by doing an action step with the first action of the plan. This is further referred to as the planner's *'online planning mode'*. This is equal to a state-transition, but it could also terminate the state.
4. The new remainder of the previously given goal-task is inserted into the database for later use. That is, when the executive requests for a new action again.

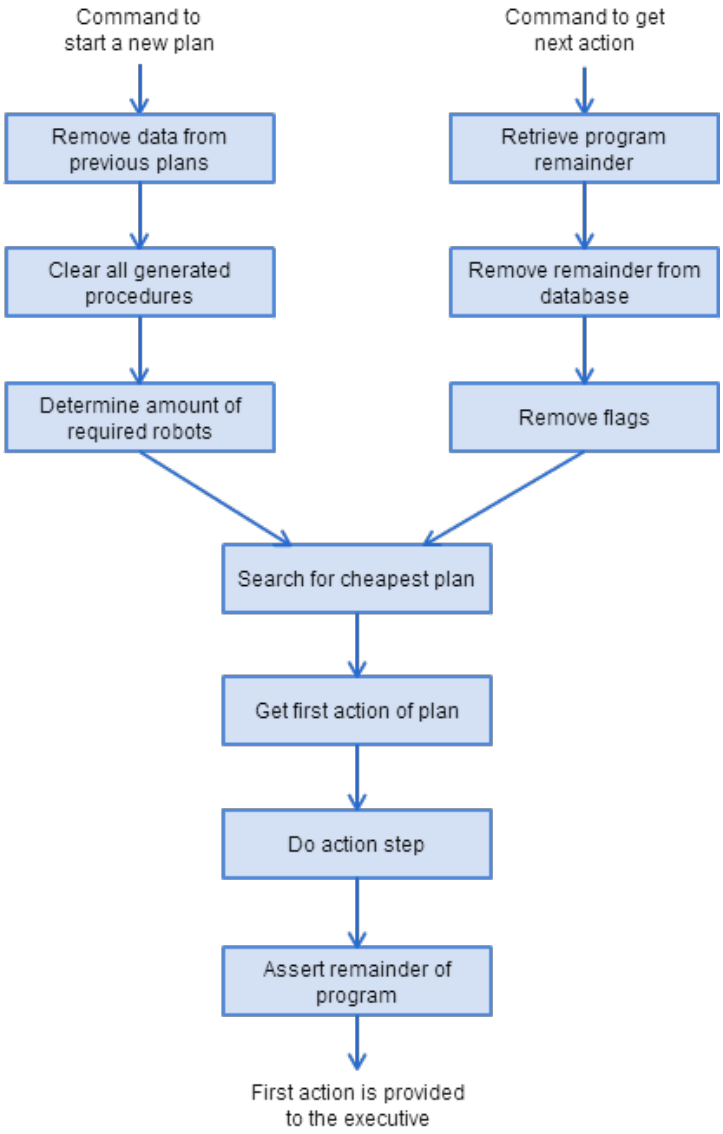

Figure 6.1: The Complete Process Flow of the Planning Module

The feedback loop between the executive and the planner is clearly present in this description of the process flow. The next section goes deeper into the details of online planning. The previously described general part of the process flow is actually a description of the predicate *do_action*/2. This predicate is also explained in more detail in the next section.

6.1.1 Feedback Loop and Replanning

Additional to starting a new plan and requesting the next action of a plan, the executive also tells the planner when an action is finished. This is especially important to get out of while-loops in a program.

The feedback loop is implemented by using PySWIP in Python (figure 6.1 provides an example of this). Because of this, the executive can do queries in Prolog. Results of the queries are given back as parameters, which can be further handled by the executive.

```
# initiates a new plan and gets the first action:
resp_action_list = list(self.prolog.query("start(X)"))

# gets the next action of a plan:
resp_action_list = list(self.prolog.query("get_action(X)"))
```

Listing 6.1: Example of PySWIP usage in Python

The process of replanning is implemented as offline planning and uses the *do*/3 predicate which is already present in MIndiGolog. A predicate similar to Prolog's *findall*/3-predicate, called *find_unique*/4 [83], is used to find a multitude of plans. A maximum is set on the amount of plans when searching for all valid plans to limit the search time of the replanning process. The maximum is currently set to 100. It is assumed (by means of empirically observing the system) this is high enough and fast enough for most planning problems.

The predicate *do_action*/2 (listing 6.2) calls the predicate *do_cheapest_first*/3, which finds the cheapest plan by replanning. This predicate in turn calls the *find_unique*/4-predicate. After collecting a multitude of valid plans, the *do_cheapest_first*/3-predicate finds the plan with the shortest duration and returns this as the cheapest solution.

```
do_action(D, C) :-
    set_planning(true),
    do_cheapest_first(D,s0,Sp),
    show_action_history(Sp),
    set_planning(false),
    get_first_action(Sp, C),
    step(D, s0, Dr, do(C,_,s0)),
    assert(remainder(Dr)),
    !.
```

Listing 6.2: The *do_action*/2-Predicate

6.2 State Transitions and State Termination

The transition and termination predicates of the planner differ from the way they were defined in the original MIndiGolog planner. This is mostly because of the inclusion of the OWL-S control constructs in the planning module. This section describes all transition and termination clauses that the planner uses. Both the transition as the termination clauses are based on situation calculus. First the transition clauses are discussed, followed by the termination clauses, and finally, conversion of control constructs is discussed. The conversion of control constructs makes the planner more generic and handle a wider variety of domain data.

6.2.1 State Transitions

The transition clauses make the planner move from state to state. They are rules on how a state can transition to another state.

A transition clause is denoted by the predicate *trans*/4, which is originally present in ConGolog [24] and MIndiGolog [40]. Its four parameters (as described by De Giacomo [24]) are:

- **D**, the program (or list of programs) to be executed;
- **S**, the current situation;
- **Dp**, the part of the program that remains to be executed in situation Sp (if transition is successful);
- **Sp**, the next situation (if transition is successful).

Thus, the *trans*/4-predicate defines how execution of a program may be single-stepped from one situation to another. The control constructs in MIndiGolog are renamed to the OWL-S control constructs with the same functionality. Furthermore, because MIndiGolog has no construct with similar functionality to the repeat-until construct, the following contribution is made to the transition rules:

> Repeat-until may transition to the loop program in sequence with another loop, as long as the test condition does not hold and the loop program may transition.

Next to this rule, even more rules are added to support constructs which include a list of programs as a parameter instead of only two programs.

The transition-predicate of the repeat-while construct (listing 6.3) is the most interesting, as it has an exception on its normal execution. The planner is designed such that the condition of a while-loop is that an action is not finished yet. In the normal execution of the repeat-while construct, it will simply transition the body of the while-loop when in 'offline planning mode', such that the planner does not get stuck in the while-loop, and, in the case of 'online planning', the behavior is as is described in the transition rules above. However, in the situation where a while-loop has been terminated before, and another while-loop with the same condition is present in the same procedure, then it is desired that this next while-loop is not skipped. Therefore, the planner must forget that a certain action is finished (making the condition true again) such that the next while-loop can be accessed. After a while-loop has been successfully terminated in the 'online planning mode', some flags are set to ensure the planner forgets that the action has been finished. The next time the executive requests a new action, the planner automatically forgets during its 'offline planning mode' that the action (that was required to be finished to get out of the while-loop) is actually finished.

```
trans(repeat_while(Cond,D),S,Dp,Sp) :-
    planning(true),
    holds(Cond,S),
    trans(D,S,Dp,Sp).
trans(repeat_while(Cond,D),S,Dp,Sp) :-
    planning(false),
    Dp = sequence(Dr,repeat_while(Cond,D)),
    holds(Cond,S),
    trans(D,S,Dr,Sp).
trans(repeat_while(Cond,D),S,Dp,Sp) :-
    planning(true),
    holds(neg(Cond),S),
    \+ someCheck(Cond),
    breakBlock(Cond),
    retract(breakBlock(Cond)),
    Cond = neg(finished(C)),
    finished(C),
    retract(finished(C)),
    trans(repeat_while(Cond,D),S,Dp,Sp).
```

Listing 6.3: The Transition-Predicate for the Repeat-While Construct

A control construct can also be converted into a different format (i.e. listed) to see if a transition is valid. If multiple programs are presented as multiple parameters of a control construct, then the parameters can be converted into a list so it can be handled by the transition predicates. This provides a general approach to handling control constructs as any number of parameters are now accepted. The conversion of control constructs is described in more detail after the next section.

6.2.2 State Termination

The termination clauses, denoted in the planner as final-predicates, make the planner terminate a state. When a state is terminated, the end of a plan is found. This can be either a complete plan, or simply a branch of a plan. For example, when actions need to be executed concurrently, the program branches out, and every branch needs to be terminated individually before the whole plan can be terminated.

A final-predicate has only two parameters (D and S) instead of four, since it does not have a new state to transition to. This is already present in ConGolog [24]. However, the control constructs in this prototype are renamed to the OWL-S control constructs with the same functionality. Furthermore, the following termination rule is added to the implementation:

Repeat-until may terminate if the loop program may terminate, or if the test is true.

Similar as with transitions, even more rules are added to support constructs which include a list of programs as a parameter instead of only two programs. Furthermore, a control construct can also be converted into a different format to see if termination is valid. The same reasoning as for the state transitions, which is discussed in the previous section, applies here. The conversion of control constructs is described in more detail in the next section.

6.2.3 Conversion Predicates

If multiple programs are presented as multiple parameters of a control construct, then the parameters can be converted into a list. The predicates for handling this conversion are discussed here.

Both the transition and termination predicates have a predicate that calls the *controlConstructConverter/2*-predicate, which converts the format of the control construct. After calling this predicate, the transition-predicate attempts to do a transition again with the "re-formatted" control construct and the termination-predicate a termination.

The *controlConstructConverter*/2-predicate (listing 6.4) simply takes a term apart, converts the arguments of the term into a single list by use of the *getArgs*/2-predicate, then creates a new term with an arity of one, and adds the list as argument to this term. The term given to this predicate needs to be in the form of a control construct which can be in a listed format.

```
controlConstructConverter(Term, ListedTerm) :-
    functor(Term, Construct, Arity),
    canBeListed(Term, Construct),
    Arity \= 2,
    getArgs(Term, Args),
    length(Args, Arity),
    functor(ListedTerm, Construct, 1),
    arg(1, ListedTerm, Args).

getArgs(Term,Args) :-
    getArgs(Term,Args,1).
getArgs(_, [], _).
getArgs(Term, [H|T], N0) :-
    arg(N0, Term, H),
    N1 is N0 + 1,
    getArgs(Term, T, N1).
```

Listing 6.4: The Conversion Predicates

6.3 Temporal Planning

Temporal planning is done by calculating distances between robot positions and specific locations. The distance, divided by an average robot moving speed, produces a time in seconds, which can be used to give a cost to a plan. This cost is then used to select the cheapest plan from all previously found plans.

Temporal planning is handled within the transition-predicate for primitive actions. In this predicate, the duration of the previous primitive action is calculated to determine the start-time of the current action in the planning process. If no other primitive action precedes the current action, than a duration of one

second is used. This transition-predicate calls the *getDuration*/2-predicate, in which the duration is either calculated, provided by the knowledge base, or automatically set to one second. It is only calculated if a distance calculation can be done for the action. Distance calculation is done by applying the Pythagorean theorem. Positions and locations are denoted as Cartesian coordinates in a two-dimensional plane, because the robots move on a flat floor.

6.4 Multi-Robot Control

This section describes the additions to MIndiGolog to provide for multi-robot control in the system. It includes a robot selection mechanism for procedure generation, robot capabilities and action conflicts based on these robot capabilities.

6.4.1 Number of Robots Selection Mechanism

As previously mentioned, the planner selects the amount of robots it needs based on available robots in the environment and other criteria based on the specificness of the given task. This mechanism heavily depends on the planning domain that is provided by the knowledge base.

The predicate *determineRobots*/1 (listing 6.5) determines the amount of robots required for the task given to the planner. It start with counting all robots and, by using information from the knowledge base, counts other things as well (e.g. amount of drinks to pickup, amount of locations of people to take orders from) based on the specific task. The planner will always assign the minimum amount of robots for a task, such that other robots can be used for other tasks. An example: there are three robots available, two 'order locations' (locations where orders can be taken from), and there is a task to take orders from people. In this situation, the planner generates a procedure where only two robots are needed. The generated procedure is also based on information from the knowledge base. If more than one robot is required for a task, then the planner generates a procedure with a split+join construct.

```
determineRobots(D) :-
    findall(X, robot(X), L1),
    length(L1, Length1),
    checkOnWhat(D, CheckFunc),
    functor(CheckTerm, CheckFunc, 1),
    arg(1, CheckTerm, Y),
    findall(Y, CheckTerm, L2),
    length(L2, Length2),
    Min is min(Length1, Length2),
    functor(Procedure, proc, 2),
    arg(1, Procedure, D),
    fillInRobots(D, Procedure, Min),
    assert(Procedure).
```

Listing 6.5: The *determineRobots*/1-Predicate

6.4.2 Robot Capabilities

The system is expanded with the use of robot capabilities in the planner. These robot capabilities are defined in the knowledge base of the system and the planner uses this information to select the right agents for certain tasks. The word agents is used because the robot capabilities are not only applicable for robots in the traditional sense, but also for single actuators and sensors, and software agents that provide, for example, reasoning capabilities.

Action Conflicts

Robot capabilities also create conflicts in the planner. These conflicts, denoted by the MIndiGolog predicate $conflicts/3$, is used by the planner to determine whether it is possible to execute a certain action at a certain time. Certain actions can be in conflict with each other. The predicate as it is in MIndiGolog is adapted to work with robot capabilities, such that actions conflict when multiple actions need to use the same actuator or sensor at the same time.

Chapter 7

Evaluation and Experiments

The evaluations and experiments are mainly to validate that the design is a good approach to robotic planning with multiple robots in a generic way. Several test cases are set up to test all functionalities of the proposed planning system. These test cases are described in detail in appendix C. Additionally, a bigger experiment is performed to validate the complete system, including the executive and ontology layers.

First, the test cases are explained in the next section. This is followed by the results from the tests based on the test cases. Finally, the experiment is described and results of the experiment are shown.

7.1 Test Cases

This section provides a summarized description of the test cases, as presented in section C.1. The test cases are grouped in the same five categories as the previously defined contributions of the planning module (see section 2.2). All tests are marked with a test ID, such that test results are more easily coupled with test descriptions. Table 7.1 displays the test IDs for every test group, and also the total number of tests (including sub-tests, such as A1.1) per test group.

Test Group	**Test IDs**	**Total # of Tests**
OWL-S Control Constructs	Test A1 to A2	10
Temporal Planning	Test B1 to B3	17
Multi-Robot Control	Test C1 to C3	5
Robot Capabilities	Test D1 to D2	6
Online Planning	Test E1 to E2	2

Table 7.1: Test Cases

The test cases are all meant to validate that the implemented prototype of the planning module functions as required. Every test case focuses on a different feature of the planning module. Tests A1 to A2 are meant to validate that the OWL-S control constructs are implemented conform to the official OWL-S description [51] and to validate that the control construct conversion predicate functions as expected. Tests B1 to B3 are made to validate that temporal planning is functioning the same way as it is designed. Tests C1 to C3 are made to validate that the planner uses all available robots in an as efficient fashion as possible. Tests D1 to D2 are meant to validate that the planner makes the right choices based on robot capabilities and to validate that certain robot capabilities create conflicts and that the planner handles these conflicts correctly. Tests E1 and E2 are meant to validate that the planner handles communication between itself and the executive program correctly and to clearly show how the feedback loop functions, which is the essential part of online planning.

7.2 Test Results

This section provides a summarized discussion of the test results, as presented in section C.2. All test cases described in section C.1 are performed and all performed tests are successful. Every valid plan that is found by the planner in the tests was able to terminate successfully. The tests show that the prototype functions as required and as expected.

Both Prolog and Python are used to implement the tests. A *Python test application* is created, which is based on Erik Geerts's executive layer of the planning system [21], to simulate the real executive for test purposes. The interaction between Python and Prolog (through PySWIP [78]) is not required for most tests, and is therefore not used for every test. Listing 7.1 displays part of the output of test A1.1 as an example of a test for which no Python is used. It shows a plan with sequential execution. The first action is expected to be executed at time step one and the second action at time step two. The time is in seconds. The value for "Action" at the bottom of the listing is what is generally sent to the executive by the planner.

```
?- start_new_plan('test_A1_1', Action).
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 2
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].
```

Listing 7.1: Example of Prolog Test Output from Test A1.1

Listing 7.2 displays part of the output of test B2.1 as an example of a test for which the Python test application is used. It shows part of a log file generated by the Python test application. The log file shows when an action is started or finished, the duration of an action, and also which actions the executive received from the planning module.

```
Starting executive

Start getting plan at time:
1392972981.89

Action(s) received from planner:
['Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)']

Starting action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392972981.9

Finished action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392972981.9
Duration: 0.000142812728882
```

Listing 7.2: Example of Python Test Output from Test B2.1

None of the tests produced unusual or unexpected behavior. Therefore, no other test is further discussed here. The reader is referred to section C.2 for more details about the test results.

7.3 Experiment

One experiment, incorporating the whole system, is also performed to validate the functionality of the complete proposed robotic planning system for multi-robot control. This experiment, with its results, is described in this section.

First, the experiment is explained. It is then followed with the results of the experiment.

7.3.1 Experiment Details

The use case for the experiment is the so-called *'cocktail party challenge'*, which is part of the RoboCup challenges for service robots. In this experiment, a user says to the planning system to have a cocktail party. Drinks then need to be served, but also cleaned up. Two robots are available, these are AMIGO and PICO. The web agent detects several available web services, which are actually actuators and sensors on the service robots. The robots can then help serving drinks and cleaning up. The planning system then needs to make a plan and send commands to the robots.

For the use case, the cocktail party is to be held at the TU/e robotics lab. The cocktail party is a composite task, consisting of four main subtasks:

1. *TakeOrder*, a robot must take drink orders from people.
2. *ServeDrink*, a robot must serve the ordered drinks to people.
3. *FindEmptyDrink*, a robot must find empty drinks.
4. *CleanupDrink*, a robot must clean up empty drinks.

There are also four types of locations in the cocktail party use case. These are:

1. *Order location.* The locations where robots get orders from people.
2. *Serve location.* The locations where robots serve drinks to people.
3. *Storage location.* The locations where robots find drinks.
4. *Dispose location.* The locations where robots dispose of empty drinks.

The experiment described here, focuses on the 'FindEmptyDrink'-subtask. The environment of this experiment (described in more detail in the next section) only has storage locations. Six computational ROS nodes are deployed in the RoboEarth Cloud Engine for this experiment. These nodes are:

- **LocMap**. This node computes an occupancy grid which is used for localization.
- **NavMap**. This node computes an occupancy grid which is used for navigation.
- **Path**. This node computes a path from location parameters A and B. A and B are bound by the scheduler to locations obtained from the OWL-S knowledge base.
- **Pose**. This node computes a pose that indicates the current position of the robot.
- **Detection**. This node detects objects. Successful detection is concluded from the action return value for 'success'.
- **VelCmd**. This node computes velocity commands, based on desired path and current pose. It only returns true if the final point in the path is reached.

The robot capabilities described in the knowledge base for this experiment are identical for both the AMIGO as the PICO robot. These robot capabilities are:

- **hasSensor**(amigo_1, Kinect)
- **hasSensor**(amigo_1, Laser)
- **hasActuator**(amigo_1, Base)
- **hasSensor**(pico_1, Kinect)
- **hasSensor**(pico_1, Laser)
- **hasActuator**(pico_1, Base)

The experiment is performed both in simulation as in the real world. This section first describes the simulation before it discusses the real world experiment.

Simulation

A simple test environment is created in the ROS Gazebo simulator to allow a fast development cycle and easy parameter tuning. Figure 7.1 shows this environment.

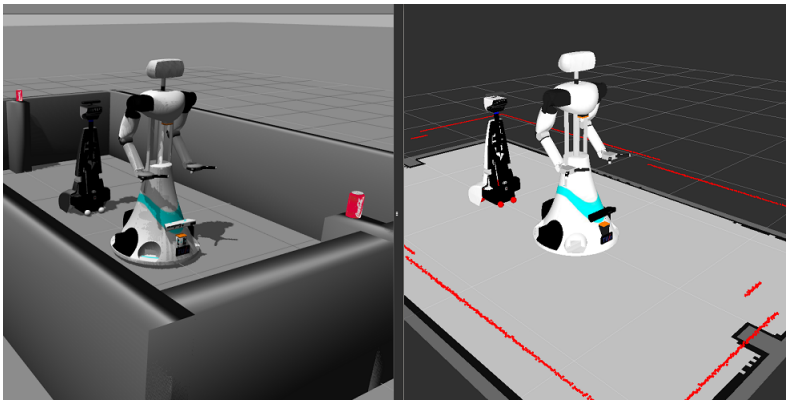

Figure 7.1: Simulation Environment, as presented by Rob Janssen (2014) [35]. Gazebo simulator (left) and Rviz visualizer (right)

The simulation spawns two robots in the test environment, together with two 'empty' drinks. These empty drinks are located at 'storage locations'. The goal is to execute the *'FindEmptyDrink'* task as devised in OWL-S. Both robots are supposed to search for the empty drinks at the locations that are nearest to their current positions.

Real World

There are also two robots present in the real world experiment, these are (as mentioned before) AMIGO and PICO of the TU/e. There are RoboEarth client interfaces deployed on both robots to enable communication through Rapyuta. The hardware of both robots include a Intel i5 Quad-Core processor, a Kinect and laser scanners.

Figure 7.2 shows the initial position of the two robots in the real world experiment. AMIGO (here on the right side of the figure) is the tallest of the duo and has two robotic arms. PICO (on the left side of the figure) has a tray for carrying objects. They are both standing in the middle. Two drink cans (posing as 'empty drinks') are visible. Just as in the simulation, the empty drinks are placed at 'storage locations' and both robots are supposed to search for the empty drinks at the locations that are nearest to their current positions.

Figure 7.2: Real world experiment start position, as presented by Rob Janssen (2014) [35].

Planner data is logged for the experiment. The next section shows the result of this log and other results of the experiment.

7.3.2 Experiment Results

This section shows and discusses the results of the previously described experiment. The findings described here are a collective effort by *Rob Janssen*, *Erik Geerts*, *Jean-Pierre Denissen* and me.

Figure 7.3 shows the final positions of the two robots in the real world experiment. The robots detect the two 'empty drinks' at these positions. Rob Janssen provided a video of this experiment [36] on the Internet.

(a) (b)

Figure 7.3: Real world experiment final position, as presented by Rob Janssen (2014) [35].

The incremental execution of the plan is logged during this experiment, as shown in listing 7.3. The times in the log represent the times when the executive performs the actions and not when the planner plans for the actions to happen. The log shows that the time between two succeeding localization steps (which is started when the second 'LocMap(Agt,Env)' is performed), is 9.46 seconds. Therefore, it has an update frequency of approximately 0.1 Hz. This is a lot lower than native ROS localization components such as AMCL [22], which typically run at frequency of 20 to 40 Hz. The cause of this is mainly because the service-based interface used in the system is considered slower than AMCL's topic-based interface.

The packet size per service on AMIGO is also logged, as shown in table 7.2. The communication data for AMIGO is assumed to be identical to the communication data of PICO, because the composed plans and hardware components are identical to both robots.

service	**send**	**received**
Laser	4136	0
Base	1	48
Kinect	2150929	0

Table 7.2: Communication Data on the AMIGO Client (in Bytes), as presented by Rob Janssen (2014) [35].

An average data transfer rate of 442 Bps (Bytes per second) for one combined localization and navigation step is obtained as a result of combining the previously presented planner log with the packet sizes of table 7.2. This is based on the update rate of 0.1 Hz, which was previously concluded.

Furthermore, CPU usage is logged on both client robots in this experiment. The CPU usage does not exceed 4% during navigation. However, the CPU usage increases temporarily to 170% (which is possible because of the Quad-core processors) when the Kinect-service is called upon.

From the planning module point of view, this experiment shows that the feedback loop is working as expected for the planner. This means that the executive successfully communicates with the planner and a valid plan can be found for every presented situation. Furthermore, the experiment shows that robot capabilities are used and that the planner can get out of while-loops due to the feedback from the executive. The procedures used in the experiment include various OWL-S control constructs: sequence, split+join, if-then-else and repeat-while. Therefore the experiment also shows that these control constructs are handled correctly. These findings strengthen the results of the previously described test cases.

```
do [NavMap(amigo_1,"tue_lab")] at time 19.02
do [NavMap(pico_1,"tue_lab")] at time 19.02
do [LocMap(amigo_1,"tue_lab")] at time 20.41
do [LocMap(pico_1,"tue_lab"] at time 20.43
do [Laser(amigo_1)] at time 21.98
do [Laser(pico_1)] at time 22.04
do [Pose(amigo_1)] at time 22.19
do [Pose(pico_1)] at time 22.25
do [Path(amigo_1,coke_1)] at time 23.98
do [Path(pico_1,coke_2)] at time 24.03
do [LocMap(amigo_1,"tue_lab")] at time 25.43
do [LocMap(pico_1,"tue_lab"')] at time 25.55
do [Laser(amigo_1)] at time 26.9
do [Laser(pico_1)] at time 26.98
do [Pose(amigo_1)] at time 27.05
do [Pose(pico_1)] at time 27.17
do [VelCmd(amigo_1)] at time 31.86
do [VelCmd(pico_1)] at time 31.96
do [Base(amigo_1)] at time 33.29
do [Base(pico_1)] at time 33.52
do [LocMap(amigo_1,"tue_lab")] at time 34.89
do [LocMap(pico_1,"tue_lab")] at time 34.94
...
do [Kinect(amigo_1)] at time 57.59
do [Detection(coke_1)] at time 58.71
do [Kinect(pico_1)] at time 61.04
do [Detection(coke_2)] at time 62.45
```

Listing 7.3: Planner Log, as presented by Rob Janssen (2014) [35].

7.4 Evaluation

This section provides a discussion of the experiment and test results and evaluates if the proposed planning system has value to the scientific world, based on the results presented in this chapter.

The test cases show that the planning module functions correctly as designed since all tests are successful. Furthermore, since the tests show that the prototype of the planning module functions as required, the design and implementation choices made for the planning module are considered a feasible approach for a planning system for multi-robot control.

Furthermore, the experiment and test cases show that the four primary goals of the system as well as the planning-specific requirements with a *"must have"* priority (as described in section 2.3) are achieved by the implemented prototype. The four primary goals are as follows:

1. The system serves as a centralized task controller for a wide variety of robot platforms and computational algorithms.
2. The coordination of robots is dictated by the central controller only.
3. The computational algorithms are deployed in a distributed and highly optimized computing environment, instead of running locally on the robots.
4. The system needs to have fast and direct access to a knowledge base that is capable of storing and retrieving task-related knowledge securely and efficiently.

Based on the experiment, it can be noticed that the current interface to the robot components and the computational algorithms on Rapyuta, which uses ROS services, is not as fast as required for certain procedures. A robot's pose is computed by receiving the robot's laser scanner data. This laser scanner data is processed at 30 Hz in standard ROS navigation architectures, whereas in the service call implementation, only a rate of ± 0.1 Hz is achieved. These low update rates, combined with the high bandwidth requirements for the Kinect data, also makes this system impractical for dynamic look-and-move visual servoing applications. One solution might be to not move all computationally heavy algorithms into the cloud, but to keep some on the physical robots, so that less data needs to be transmitted between clients and servers.

Finally it can be stated that the proposed planning system indeed has value to the scientific world. It can be concluded from the low CPU usage in the experiment that it helps to offload the computational part of the robot by use of cloud robotic technologies such as Rapyuta. It is therefore useful to continue to research ubiquitous and cloud robotic technologies. Furthermore, with this research it is shown that multi-robot control is a feasible approach to ubiquitous robotics. Robots are becoming more affordable every year, it is expected that more experiments with multiple robots will follow in the near future.

However, due to time and resources limitations, only a simple experiment is conducted. If a more complex experiment use case is used, for example the complete *'cocktail party'* case instead of only the *'detect empty drinks'* part, then the value of the proposed system can be measured in a more complete fashion. It is therefore required to do more research, especially in the form of experiments, if one wants to understand the full power of the proposed system.

Furthermore, not every requirement (see appendix A for a full list) is achieved. Seventeen of all nineteen requirements are achieved. The requirements that are not achieved are requirement F7 (The planning module uses web service composition to find available robots) and requirement NF10 (High-level modules include human-interfacing layers). Both of these requirements have a "nice to have" priority, therefore it is not significant that these are not achieved. Requirement F7 is not achieved because the planning module does not actually discover available robots through web service composition. Nonetheless, the planning module still reasons over robots in the form of web services. Requirement NF10 is not achieved because no effort is made on a human-interface for the planning system.

Chapter 8

Discussion

The aim of this project is, as written in the introduction, to develop a web-based entity that uses service robots represented as collections of web services to create and execute plans for completing tasks provided by an user, and to investigate how such a web-based ubiquitous intelligent software agent for robotic planning and control can be developed and realized. This chapter discusses whether this goal is accomplished by the project. Next to that, this chapter also discusses work related to this project, and discusses issues for potential future research. The next section is about similar and related work.

8.1 Related Work

This section puts the proposed system in context with other robotic planning systems and other ubiquitous robotic systems. There are several examples of middleware and frameworks of ubiquitous robots, among these are the Ubiquitous Robotic Companion [26][27], PEIS-ecologies [72], and the UPnP-Middleware framework [1]. This section shortly mentions a few related projects and then highlights the differences between the proposed system and several other frameworks for ubiquitous robotics and robotic planning systems.

8.1.1 Related Projects

First of all, it should be mentioned here that the works of *Rob Janssen* [35], *Erik Geerts* [21] and *Jean-Pierre Denissen* [13] are related to the project described in this thesis. While this thesis mainly describes the planning module of the proposed system for multi-robot planning and control, the works of Janssen, Geerts and Denissen describe the other parts of this proposed system.

Furthermore, the work of *Ziyang Li* [48] is also related. Li also worked under the guidance of Rob Janssen at Eindhoven University of Technology. He combined a Golog-language (IndiGolog) with ROS and OWL-S. His planning system is however only applicable to situations with only one robot. It includes parallel planning for a single robot and some form of plan failure recovery.

8.1.2 Comparison

This section provides a comparison between the system proposed in this thesis and the following systems and projects: Ubiquitous Robotic Companion [26][27], Ubibot [44], PEIS-Ecologies [72], Rubicon [69], RobotEra [67], UPnP Robot Middleware [1], Hybrid Deliberative Layer (HDL) [28], and the Martha project [2][3]. All these systems are described in section 4.3.3 with the exception of the Martha project, which is discussed in section 3.2.1, and of the HDL, as it has nothing to do with ubiquitous robotics. HDL is described in section 8.2.5. Furthermore, a comparison of the various planning systems is displayed in table 8.1. Some systems are omitted from this table, because their implementation details are not publicly published (yet).

The main strength of our system is that it focuses on robot cooperation in a dynamic environment. Not many other systems focus on this. Most ubiquitous robotic systems only show a single robot cooperating with the environment, or a robot being used/controlled by an intelligent environment.

Many projects have not made their source code publicly available, and are therefore marked as closed source in the table. It is impossible to increment on closed source projects. Our proposed system is partially open source, as the source code of the planning module is added in appendix B of this thesis. Hartanto published the source code of HDL in his thesis and the source code of PEIS is available online.

Our proposed system is the only ubiquitous robotic system in which the planner is based on situation calculus. Furthermore, no other system uses Rapyuta for handling communication in the system.

System or Project	**Planner Language**	**Planning Technology**	**Communication Technology**	**Execution Technology**	**Ontology Technology**	**Open Source**
Our Proposed System	Adapted version of MIndiGolog 0	Situation calculus	Rapyuta and ROS messages	ROS Python	OWL-S	No
HDL	JSHOP2	Hybrid of HTN and DL	*unknown*	*N/A*	OWL-DL	Yes
Martha	C-PRS	HTN-planning	*unknown*	*unknown*	*N/A*	No
PEIS-Ecologies	PTLplan	sensor-based probabilistic action-planning	TCP/IP	Fuzzy Logic	XML	Yes
Rubicon	PTLplan	PEIS and MAS	TCP/IP	ROS	*N/A*	No
Ubibot	*unknown*	*unknown*	C++ Middleware with HTTP and FTP	Visual C++ 6.0	*unknown*	No
SURF/ SemanticURS	UMCP and SHOP2	HTN-planning	SOAP, XML and HTTP	WSBPEL and Java	OWL-S and Jena	No

Table 8.1: Comparison of the Various Planning Systems

Some systems use learning mechanisms, including the Ubibot system and the RUBICON project. While the Ubibot system uses complex virtual agents (called "Sobots"), our proposed system does not use such a model. I see no need in something as complex as a learning agent with emotions for ubiquitous robotics, it is more important to have a good generic central mission planner first. No mission planner is clearly described for the Ubibot system.

There are some systems containing middleware which is specifically designed to increase modularity and scalability. Our system has no detailed design for such a middleware yet, but we do use Rapyuta. I believe that Rapyuta is a good starting point for coming up with approaches to increase modularity and scalability for our proposed system.

The goal of the Martha project is to have as little centralized control as possible [3], whereas in ubiquitous robotic systems, the goal is to have as little as possible computation on the individual robots. This is one of the main differences between our proposed system and the Martha project.

Our proposed system has no advanced knowledge discovery module as in, for example, SURF [26]. This is still a point where many improvements can be made. Which leads to the next section, where the direction for future research is discussed.

8.2 Future Research

It must be determined whether the goals of the project are accomplished, before a direction can be given for future research.

The goal of the project, as written in the introduction of this thesis, is to investigate how a web-based ubiquitous intelligent software agent for multi-robot planning and control can be developed and realized. As mentioned in the evaluation (section 7.4), the four primary requirements of the system as well as the planning-specific requirements (as described in section 2.3) are achieved by the implemented prototype.

The following sections discuss the points that can be improved or expanded upon.

8.2.1 Planning with Time and Resources

Temporal planning is not yet very sophisticated in the current planner, and thus it can be improved upon. For example, it does not update the robot positions yet while planning, and the planner does not remember what locations in the environment are already checked by a robot. A dynamic world model is required to be included in the system to make this work.

A more advanced model of temporal planning could be added to the planner, including optimal use of both time and resources, but also uncertainty about the world state. The book automated planning by Ghallab et al. [23] is a good starting point, but there are also many papers published on temporal planning (e.g. [19], [25], [55] and [39]).

8.2.2 Web Service Composition

The proposed system has limited web service composition mechanisms. The planner selects robots, actuators and sensors from a knowledge base, and all these devices are described as web services. So the planner does include web service composition. However, there is no algorithm yet that includes the discovery of web services. Other systems, such as SURF [26], do have such a discovery algorithm. Our system can be expanded upon by including a similar algorithm.

8.2.3 Experiments

Eventually, due to time and resource limitations, only a few small experiments were performed with the system. First of all, it is advised to perform a larger experiment with the system. The demo of the experiment needs to be expanded, so that all subtasks of the 'cocktail party' use case are included. Also, the environment needs to be upscaled to the full TU/e robotics lab, as this increases the work space and the number of available 'order', 'serve', 'storage' and 'dispose' locations.

Especially the executive and the ontology modules of the system need to be expanded before such a large-scale experiment can be conducted.

8.2.4 Performance

The experiments showed that the performance of the communication interface needs to be improved as it is not as fast as required for certain procedures. For example, if communication updates can be scaled up

to a rate of 30 Hz (comparable to that of AMCL [22]) then an average data transfer rate of 130 KBps can be obtained. Rob Janssen provides more information on this topic in his paper [35].

It might also be interesting to compare various ubiquitous robotic systems on performance. However, a full comparison cannot be made, as some systems are closed-source.

8.2.5 Future Vision

There are many more directions in which the system can be extended. Some of these directions are described in this section. The subjects discussed here are: plan failure recovery, dynamic world model representation, relaxing more planning domain assumptions, and other issues such as scalability and safety.

Plan Failure Recovery

Some planners include a plan failure recovery mechanism. For example, Ziyang Li [48] describes failure recovery for IndiGolog. It can be researched if a similar approach can be used for the system proposed in this thesis.

Such a recovery mechanism is needed after a previously generated plan failed due to certain events. A repair plan needs to be generated to get the system back into a safe state or to finish the previously given task from the current state of the system. It needs to be investigated how such a recovery mechanism can be designed. First of all, the system must monitor where the plan (or execution of the plan) went wrong.

Relaxing More Assumptions

The assumptions as made in section 5.2 can be relaxed even further, such that the model of the system matches the real world even better. As these planning domain assumptions are relaxed even further, methods and measures must be included in the planner for handling the relaxed assumptions.

For example, expected events can be taken into account, making the system more dynamic (see assumption A3). A lot of events can be expected by learning from the humans in the environment, humans are sometimes said to be creatures of habit, and as such, their behaviors can be anticipated by an intelligent system if it includes human-behavior or user modeling.

Furthermore, the user-given goals can be extended to include maintenance goals (see assumption A4). Maintenance goals are goals that tell the planner that some states may not be reached at any point in a plan. A classic example is that a robot must not harm a human and must therefore avoid every situation where it could harm a person. However, it could also be a simpler goal, such as keeping water in a glass when handling it.

Ghallab et al. [23] describe for each assumption how it can be relaxed and how a planning system can handle this.

Other Challenges

There are many more challenges which will be important in the future, especially when robot work environments get bigger. At some point, the environment and the knowledge base of the system becomes too large for a planning system to produce plans within a reasonable time. Therefore, modularity and scalability are important issues. An approach is to move context-awareness out of software agents and into middleware, such as In-Bae Jeong and Jong-Hwan Kim propose in their paper [37]. When a software agent, or planning agent, needs certain information, it can request it when it needs it. Nevertheless, the search space of the planner needs to be pruned to decrease planning time. Hartanto also proposed an approach for this in his Hybrid Deliberative Layer [28]. Hartanto's Hybrid Deliberative Layer (HDL) combines HTN-planning with Description Logic into a new hybrid planning approach. It uses JSHOP2 [32] for the planner and OWL-DL [53] for the ontology model. The DL reasoner is used for automatically extracting a tailored representation for each and every concrete planning problem, and handling this to the HTN-planner.

Furthermore, in a dynamic environment full of humans, the system must be well aware of these humans and their behavior. Cirillo et al. [12] proposed a framework for human-aware robot planning, which might be a good starting point when such techniques are to be implemented in our proposed system.

Finally, one could think of several legal, moral, privacy, and safety challenges. For example, there are more security issues on a distributed system than a centralized system, and even more in ubiquitous robotics. Especially when it is widely used in ambient home environments. Every household might have multiple robots in the future, all connected to the Web, with their intelligence ubiquitously on the Web. It is important to research what damage a hacker can do to such a system. For example, could a hacker rewrite the intelligence without the users knowing it?

Chapter 9

Conclusion

I set out to to investigate how a web-based ubiquitous intelligent software agent for multi-robot planning and control can be developed and realized in the research project described in this thesis. The field of automated planning for robotic systems, with a focus on multi-robot systems, is analyzed, and a literature study is performed. Requirements based on these preliminary studies are established and a design for a framework is made. Then, based on this design, a prototype is implemented and experiments were performed on this prototype of the planning system.

The main contribution of this project is a planning module, specifically designed for a centralized task-planning framework for multi-robot control. The planning module includes a situation calculus-based approach on planning. The situation calculus is chosen for its representation of states and actions. MIndiGolog, a planning language based on situation calculus, is chosen because it supports concurrent and temporal planning. Furthermore, OWL-S control constructs are successfully incorporated in the planner, such that the planner can reason over data represented on the Semantic Web. OWL-S is chosen as representation because it is a standard for describing web services. Robot components are described as OWL-S services, such that the system can easily be extended with more Semantic Web technologies. Robot positions, the distance between robots and locations in the environment and the assumed average velocity of robots are used to calculate the durations of actions. The total duration of all actions in a plan represent the cost of a plan. Based on these costs, the planner can select the cheapest plan out of a multitude of valid plans. The planner handles concurrency, and even selects how many robots it requires for planning, based on the user-given task and certain factors in the environment. It then generates procedures for the planning domain based on the amount of required robots. Also, the planner handles robot capability descriptions, such that it only selects robots that have the right capabilities to perform certain tasks. Finally, the planner is adapted into an online planner, such that it has a feedback loop with the executive (or control layer). However, in contrast to other online planners that only consider selecting an action for the current situation, the planning module keeps its backtracking capabilities. These backtracking capabilities are typical of offline planners, as offline planners plan for the whole plan. Backtracking means that when no valid action can be found to transition from or terminate a situation, the planner can step back to the previous situation and try a different action. Backtracking can also be used to find alternative plans. Backtracking over plans is performed when the planner is busy with replanning in an offline manner.

The results show that the proposed framework is a feasible approach to multi-robot planning and control. However, there are still some limitations in the performance of the system, including latency and bandwidth issues. These limitations need to be tackled before the planning system can really be of use. One solution might be to not move all computationally heavy algorithms into the cloud, but to keep some on the physical robots, so that less data needs to be transmitted between clients and servers. A good balance in this still needs to be researched for the system. Further testing is also desired to get a clear idea on the full power of the proposed system.

Several additional directions for further research are suggested based on the results of the tests and experiments, and the analysis of comparable robotic planning systems.

As a final note, I believe that eventually, the word robot should disappear for ubiquitous robotics to be successful. People need to think they are interacting with their environment, their homes and work spaces, instead of with a robot, for the concept to be fully ubiquitous.

Bibliography

[1] Sang Chul Ahn, Ki-Woong Lim, Jung-Woo Lee, Heedong Ko, Yong-Moo Kwon, and Hyoung-Gon Kim. UPnP robot middleware for ubiquitous robot control. In *Proc of the 3rd Int Conf on Ubiquitous Robots and Ambient Intelligence (URAI 2006)*, 2006.

[2] R. Alami, S. Fleury, M. Herrb, F. Ingrand, and F. Robert. Multi-robot cooperation in the MARTHA project. *IEEE Robotics and Automation Magzine*, pages 36–47, 1998.

[3] Rachid Alami, Frédéric Robert, Félix Ingrand, and Sho'ji Suzuki. Multi-robot cooperation through incremental plan-merging. *IEEE Robotics and Automation Magzine*, 3:36–47, May 1995.

[4] Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116:123–191, Jan 2000.

[5] Tim Berners-Lee, James Hendler, and Lassila Ora. The Semantic Web. *Scientific American Magazine*, May 2001.

[6] Andreas Birk, Kausthub Pathak, Soeren Schwertfeger, and Winai Chonnaparamutt. The IUB Rugbot: an intelligent, rugged mobile robot for search and rescue operations. In *IEEE International Workshop on Safety, Security, and Rescue Robotics (SSRR)*. IEEE Press, 2006.

[7] Patrick Blackburn, Johan Bos, and Kristina Striegnitz. Learn Prolog Now! Website, `http://www.learnprolognow.org/`, 2012. Accessed on 2014, February 12.

[8] Antonio Brogi, Sara Corfini, and Razvan Popescu. Semantics-based composition-oriented discovery of web services. *ACM Transactions on Internet Technology (TOIT)*, 8(Issue 4):1–19, Sep. 2008.

[9] Mathias Broxvall, Beom-Su Seo, and Woo-Young Kwon. The PEIS kernel: A middleware for ubiquitous robotics. In *Proc. of the IROS-07 Workshop on Ubiquitous Robotic Space Design and Applications*, 2007.

[10] Wolfram Burgard, Armin B. Cremers, Dieter Fox, Dirk Hähnel, Gerhard Lakemeyer, Dirk Schulz, Walter Steiner, and Sebastian Thrun. Experiences with an interactive museum tour-guide robot, June 1998.

[11] Kristoffer Cavallin and Peter Svensson. Semi-autonomous teleoperated search and rescue robot. Master's Thesis, Umeå University, Sweden, Feb. 2009.

[12] M. Cirillo, L. Karlsson, and A. Saffiotti. A framework for human-aware robot planning. In *Proc. of the Scandinavian Conference on Artificial Intelligence*, 2008.

[13] Jean-Pierre Denissen. Title yet unknown. Master's Thesis, Eindhoven University of Technology, March 2014.

[14] K. Erol, J. Hendler, and D.S. Nau. UCMP: A sound and complete procedure for hierarchical task-network planning. In *Proc. Int. Conf. on Artificial Intelligence Planning Systems*, pages 249–254, Chicago, IL, USA, 1994.

[15] J.M. Evans. Helpmate: An autonomous mobile robot courier for hospitals. In *Proceedings IROS*, pages 1695–1700, 1994.

[16] K.P. Ferentinos, K.G. Arvanitis, and N. Sigrimis. Heuristic optimization methods for motion planning of autonomous agricultural vehicles. *Journal of Global Optimization*, 23:155–170, 2002.

[17] T. Fong, B. Hine, and M. Sims. Intelligent mechanisms group research summary. Technical report, NASA Ames Research Center, Jan. 1992.

[18] Tully Foote. Documentation - ROS Wiki. Website, http://www.ros.org/wiki/, July 2013. Accessed on 2013, July 30.

[19] M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *JAIR*, 20:61, 2003.

[20] Pieterjan van Gastel. PJ's AI and Other Dev: My New HTN-Planner Written in C#. Website, http://pjvgastel-programming.blogspot.nl/2013/10/chp-my-new-htn-planner-written-in-c.html, Oct. 2013. Accessed on 2014, January 22.

[21] Erik Geerts. The design of a generic high-level multi-robot execution layer. Master's Thesis, Eindhoven University of Technology, March 2014. CST 2014.020.

[22] Brian P. Gerkey, David V. Lu, and Sarah Osentoski. amcl - ROS Wiki. Website, http://wiki.ros.org/amcl, Aug. 2011. Accessed on 2014, February 03.

[23] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning: theory and practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[24] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. Congolog, a concurrent programming language based on the situation calculus, 2000.

[25] Robert P. Goldman. Durative planning in HTNs. In Derek Long, Stephen F. Smith, Daniel Borrajo, and Lee McCluskey, editors, *Proc of the Int Conf on Automated Planning and Scheduling (ICAPS 2006)*, pages 382–385. AAAI, 2006.

[26] Young-Guk Ha, Joo-Chan Sohn, and Young-Jo Cho. Service-oriented integration of networked robots with ubiquitous sensors and devices using the semantic web services technology. In *Proc of the IEEE/RSJ Int Conf on Intelligent Robots and Systems (IROS 2005)*, 2005.

[27] Young-Guk Ha, Joo-Chan Sohn, Young-Jo Cho, and Hyunsoo Yoon. Towards a ubiquitous robotic companion: Design and implementation of ubiquitous robotic service framework. *ETRI Journal*, vol. 27(no. 6):666–676, 2005.

[28] Ronny Hartanto. *Fusing DL Reasoning with HTN Planning as a Deliberative Layer in Mobile Robotics*. PhD Thesis, Universität Osnabrück, Aug. 2009.

[29] Jørgen Havsberg Seland and Emanuel Greisen. jshop2-rt - A fork off of the JSHOP2 HTN planner project optimized for real-time planning - Google Project Hosting. Website, https://code.google.com/p/jshop2-rt/, May 2009. Accessed on 2014, January 28.

[30] E.J. van Henten, J. Hemming, B.A.J. van Tuijl, J.G. Kornet, J. Meuleman, J. Bontsema, and E.A. van Os. An autonomous robot for harvesting cucumbers in greenhouses. *Autonomous Robots*, vol. 13(issue 3):241–258, Nov. 2002.

[31] Dominique Hunziker, Mohanarajah Gajamohan, Markus Waibel, and Raffaello D'Andrea. Rapyuta: The RoboEarth Cloud Engine. In *Proc of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, pages 438–444, 2013.

[32] O. Ilghami. Documentation for JSHOP2. Technical Report CS-TR-4694, Department of Computer Science, University of Maryland, 2005.

[33] InTouch Technologies Inc. Making Remote Presence Routine | InTouch Health. Website, http://www.intouchhealth.com/products-and-services/products/, 2014. Accessed on 2014, February 16.

[34] iRobot Corporation. iRobot 510 PackBot - Overview. Website, http://www.irobot.com/en/us/learn/defense/packbot.aspx, 2013. Accessed on 2014, February 16.

[35] R.J.M. Janssen, M.J.G. v.d. Molengraft, H. Bruyninckx, and M. Steinbuch. Centralized task control with computational offloading for ROS enabled service robots. *Transactions on Automated Science and Engineering Special Issue on Cloud Robotics and Automation*, 2014. (submitted).

[36] Rob Janssen. Centralized Task Control For ROS Enabled Service Robots - YouTube. Website, http://www.youtube.com/watch?v=4jCGcRs6GZI, January 2014. Accessed on 2014, February 03.

[37] In-Bae Jeong and Jong-Hwan Kim. Multi-layered architecture of middleware for ubiquitous robot. In *Proc of the IEEE Int Conf on Systems, Man and Cybernetics (SMC 2008)*, pages 3479–3484, Oct. 2008.

[38] L. Karlsson. Conditional progressive planning under uncertainty. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 431–438. AAAI Press, 2001.

[39] Ryan Francis Kelly. *Asynchronous Multi-Agent Reasoning in the Situation Calculus*. PhD Thesis, The University of Melbourne, Oct. 2008.

[40] Ryan Francis Kelly. RFK | Research | Thesis. Website, `http://www.rfk.id.au/ramblings/research/thesis/`, Oct. 2008. Accessed on 2014, January 28.

[41] Jong-Hwan Kim, Seung-Hwan Choi, In-Won Park, and Sheir Afgen Zaheer. Intelligence technology for robots that think. *IEEE Computational Intelligence Magazine*, Mar. 2013.

[42] Jong-Hwan Kim, In-Bae Jeong, In-Won Park, and Kang-Hee Lee. Multi-layer architecture of ubiquitous robot system for integrated services. *International Journal of Social Robotics*, Volume 1(Issue 1):19–28, Jan. 2009.

[43] Jong-Hwan Kim, Yong-Duk Kim, and Kang-Hee Lee. The third generation of robotics. In *Proc of the 2nd Int Conf on Autonomous Robots and Agents (ICARA)*, 2004.

[44] Jong-Hwan Kim, Kang-Hee Lee, Yong-Duk Kim, Naveen Suresh Kuppuswamy, and Jun Jo. Ubiquitous robot: A new paradigm for integrated services. In *Proc of the IEEE Int Conf on Robotics and Automation*, pages 2853–2858, April. 2007.

[45] Taehong Kim, Young-Guk Ha, Jihoon Kang, Daeyoung Kim, Chong Poh Kit, and Joo-Chan Sohn. Experiments on building ubiquitous robotic space for mobile robot using wireless sensor networks. In *Proc of the 22nd Int Conf on Advanced Information Networking and Applications (AINAW 2008)*, pages 662–667, Mar. 2008.

[46] D. Kingston, R.W. Beard, and R.S. Holt. Decentralized perimeter surveillance using a team of UAVs. *IEEE Transactions on Robotics*, Volume 24(Issue 6):1394–1404, Dec. 2008.

[47] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains, 1994.

[48] Ziyang Li. Applying IndiGolog to the domestic service robot domain. Master's Thesis, Eindhoven University of Technology, Eindhoven, Aug. 2013. CST 2013.082.

[49] A. Mandow, J. M. Gómez-de Gabriel, J. L. Martínez, V. F. Muñoz, A. Ollero, and A. García-Cerezo. The autonomous mobile robot AURORA for greenhouse operation. *IEEE Robotics Automation Magazine*, 3(4):18–28, 1996.

[50] Adam Mann. Almost Being There: Why the Future of Space Exploration Is Not What You Think - Wired Science. Website, `http://www.wired.com/wiredscience/2012/11/telerobotic-exploration/all/`, Nov. 2012. Accessed on 2014, February 16.

[51] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. OWL-S: Semantic Markup for Web Services, World Wide Web Consortium (W3C). Website, `http://www.w3.org/Submission/OWL-S/`, Nov. 2004. Accessed on 2014, January 08.

[52] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969. reprinted in McC90.

[53] Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview, World Wide Web Consortium (W3C). Website, `http://www.w3.org/TR/owl-features/`, Feb. 2004. Accessed on 2013, July 31.

[54] Mozart Consortium. Mozart Programming System. Website, `http://mozart.github.io/`, 2013. Accessed on 2014, January 28.

[55] Dana Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, Volume 20:379–404, Dec. 2003.

[56] Dana Nau, Yue Cao, Amnon Lotem, and Hector Muñoz Avila. SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, volume 2 of *IJCAI'99*, pages 968–973, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[57] Dana Nau, Yue Cao, Amnon Lotem, and Hector Muñoz Avila. SHOP and M-SHOP: Planning with ordered task decomposition. Technical Report CS-TR-4157, Department of Computer Science, University of Maryland, June 2000.

[58] Dana S. Nau. Current trends in automated planning. *AI Magazine, AAAI*, Volume 28(4):43–58, 2007.

[59] Dana S. Nau. dananau / Pyhop – Bitbucket. Website, `https://bitbucket.org/dananau/pyhop`, June 2013. Accessed on 2014, January 28.

[60] Karol Niechwiadowicz and Zahoor Khan. Robot based logistics system for hospitals - survey. In *Proc of the IDT Workshop on Interesting Results in Computer Science and Engineering*, 2008.

[61] Illah R. Nourbakhsh, Clay Kunz, and Thomas Willeke. The mobot museum robot installations: A five year experiment. In *Proc of the IEEE/RSJ Int Conf on Intelligent Robots and Systems (IROS 2003)*, volume 4, 2003.

[62] OASIS. OASIS Web Services Business Process Execution Language (WSBPEL) TC | OASIS. Website, `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`, May 2007. Accessed on 2014, February 03.

[63] Port of Rotterdam. Port of Rotterdam Authority - Major renovation for mother of automated terminals. Website, `http://www.portofrotterdam.com/en/News/pressreleases-news/Pages/20100428_02.aspx`, April 2010. Accessed on 2014, January 27.

[64] Azzurra Ragone, Tommaso Di Noia, Eugenio Di Sciascio, Francesco M. Donini, Colucci Simona, and Francesco Colasuonno. Fully automated web services discovery and composition through concept covering and concept abduction. In *Int. J. Web Service Res.*, pages 85–112, 2007.

[65] Raymond Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.

[66] RoboEarth. RoboEarth. Website, `http://www.roboearth.org/`, July 2013. Accessed on 2013, July 30.

[67] Robot-Era Project. Robot-Era - Implementation and integration of advanced Robotic systems and intelligent environments in real scenarios for the ageing population. Website, `http://www.robot-era.eu/robotera/`, Dec. 2013. Accessed on 2014, January 29.

[68] M.D. Rossetti, A. Kumar, and R.A. Felder. Mobile robot simulation of clinical laboratory deliveries. In *Proceedings of the 1998 Winter Simulation Conference*, pages 1415–1422, 1998.

[69] RUBICON. FP7 Project Rubicon - Home. Website, `http://fp7rubicon.eu/`, June 2013. Accessed on 2014, January 29.

[70] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (3. internat. ed.)*. Pearson Education, third edition, 2010.

[71] A. Saffiotti, M Broxvall, M. Gritti, K. LeBlanc, R. Lundh, Rashid J., B.S. Seo, and Y.J. Cho. The PEIS-Ecology project: Vision and results. In *Proc of the IEEE/RSJ Int Conf on Intelligent Robots and Systems (IROS-08)*, pages 22–26, Nice, France, 2008.

[72] Alessandro Saffiotti and Mathias Broxvall. PEIS ecologies: Ambient intelligence meets autonomous robotics. In *Proc of the Int Conf on Smart Objects and Ambient Intelligence (sOc-EUSAI)*, pages 275–280, 2005.

[73] R. G. Simmons and E. Krotkov. An integrated walking system for the Ambler planetary rover. In *IEEE International Conference on Robotics and Automation*, pages 2086–2091, April 1991.

[74] Reid G. Simmons. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*, 10(1):34–43, 1994.

[75] C.G. Sørensen, T Bak, and R.N. Jørgensen. Mission planner for agricultural robotics. *AgEng 2004*, 2004.

[76] Austin Tate. Project planning using a hierarchic non-linear planner. Technical Report D.A.I. Research Report No. 25, Department of Artificial Intelligence, University of Edinburgh, August 1976.

[77] Austin Tate, Brian Drabble, and Richard Kirby. O-plan2: an open architecture for command, planning and control. In *Intelligent Scheduling*, pages 213–239. Morgan Kaufmann, 1994.

[78] Yuce Tekol and Rodrigo Starr. pyswip - PySWIP is a bridge between Python and SWI-Prolog. - Google Project Hosting. Website, `https://code.google.com/p/pyswip/`, Dec. 2012. Accessed on 2014, January 07.

[79] W3C. Semantic Web – W3C, World Wide Web Consortium (W3C). Website, `http://www.w3.org/standards/semanticweb/`, 2013. Accessed on 2013, July 30.

[80] Xia Wang and Wolfgang A. Halang. *Discovery and Selection of Semantic Web Services*, volume 453 of *Studies in Computational Intelligence*. Springer, Berlin, 2013.

[81] Mark Weiser. The computer for the 21st century. *Scientific American*, 9:66–75, 1991.

[82] Mark Weiser. Ubiquitous computing, Mark Weiser. Website, `http://www.ubiq.com/hypertext/weiser/UbiHome.html`, Mar. 1996. Accessed on 2013, July 23.

[83] Jan Wielemaker. Re: findall with limited solution count - Google Groups. Website, `https://groups.google.com/forum/#!topic/comp.lang.prolog/hYTmHjrjQq8`, Feb. 2006. Accessed on 2014, January 31.

[84] David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann, San Mateo, CA, 1988.

[85] A. Yachir, K. Tari, A. Chibani, and Y. Amirat. Towards an automatic approach for ubiquitous robotic services composition. In *Proc of the IEEE/RSJ Int Conf on Intelligent Robots and Systems (IROS 2008)*, pages 3717–3724, Sept. 2008.

[86] Guang Ying Yang. The strategy study of the international fire fighting robot competition. *Applied Mechanics and Materials*, 26:443–447, June 2010.

Contact

ing. Pieterjan van Gastel
Utrecht University
Artificial Intelligence division, Intelligent Systems group
Princetonplein 5, De Uithof
3584 CC Utrecht
The Netherlands
Email UU: P.J.G.vanGastel@students.uu.nl
Email TU/e: P.J.G.v.Gastel@tue.nl

Appendix A

Project Requirements

This appendix is an extension to section 2.3 and describes the requirements of the robotic planning system in more detail. The requirements are divided in both functional and non-functional requirements. The requirements described in this section give direction to the research project and provide in a base for the design of the system. This appendix first describes the functional requirements, followed by the non-functional requirements.

A.1 Functional Requirements

ID	F1
Requirement	The planning module finds the most optimal plan (with the given knowledge), if a plan exists.
Acceptance Criteria	For every given planning domain and real-world situation, and for every given user task, if a plan exists, the planning module finds the most optimal plan. The most optimal plan is the plan with the shortest expected execution time, but also the plan in which the robots with the right capabilities for certain given task(s) are chosen for those tasks for which those capabilities are required.
Priority	Must have

ID	F2
Requirement	The planning module handles time and duration of actions (temporal planning).
Acceptance Criteria	The planning module only schedules the next action in a sequence after the preceding actions are finished. This continues to function even when the planner plans for concurrent execution.
Priority	Must have

ID	F3
Requirement	The planning module handles concurrency of actions.
Acceptance Criteria	The planning module schedules actions for concurrent execution when required, but only when possible. The planning module attempts to schedule actions in sequence if concurrency is not possible.
Priority	Must have

ID	F4
Requirement	The planning module uses OWL-S control constructs.
Acceptance Criteria	The planning module uses OWL-S control constructs when performing state transitions or state terminations. All OWL-S control constructs need to function as described on the OWL-S website [51].
Priority	Must have

ID	F5
Requirement	The planning module handles all data that is parsed by the *OWL-S Ontology Layer*.
Acceptance Criteria	The planning module can produce valid plans with everything that is parsed by the OWL-S Ontology Layer [13].
Priority	Must have

ID	F6
Requirement	The planning module provides plans which can be executed by the *Execution Layer*.
Acceptance Criteria	All plans provided by the planning module can be executed by the Execution Layer [21].
Priority	Must have

ID	F7
Requirement	The planning module uses web service composition to find available robots.
Acceptance Criteria	The planning module finds at least one available robot, in the form of a collection of web services, by use of web service composition.
Priority	Nice to have

A.2 Non-Functional Requirements

ID	NF1
Requirement	The planning system serves as a centralized task controller for a wide variety of robot platforms and computational algorithms.
Acceptance Criteria	At least two robots are controlled by the planning system.
Priority	Must have

ID	NF2
Requirement	The coordination of robots is dictated by the central controller only.
Acceptance Criteria	Robots are not autonomous. The central controller is the only unit that controls robots.
Priority	Must have

ID	NF3
Requirement	There is as little software as possible on the involved robot platforms themselves.
Acceptance Criteria	The involved robot platforms only contain software for performing basic functionalities. All higher level computing algorithms are in the cloud.
Priority	Must have

ID	NF4
Requirement	The involved robot platforms contain only low-level modules for hardware control.
Acceptance Criteria	The involved robot platforms only contain low-level modules and no high-level modules.
Priority	Nice to have

ID	NF5
Requirement	The computational algorithms are deployed in a distributed and highly optimized computing environment, instead of running locally on the robots.
Acceptance Criteria	The computational algorithms are deployed in a cloud environment.
Priority	Must have

ID	NF6
Requirement	The system has fast and direct access to a knowledge base that is capable of storing and retrieving task-related knowledge securely and efficiently.
Acceptance Criteria	The system stores and retrieves knowledge from a knowledge base.
Priority	Nice to have

ID	NF7
Requirement	Interfaces with web entities make the connection with the high-level modules.
Acceptance Criteria	The system calls computational algorithms that are deployed in a cloud environment.
Priority	Must have

ID	NF8
Requirement	High-level modules include execution layers.
Acceptance Criteria	There is a functioning execution layer defined as high-level module and present on the cloud environment (or server).
Priority	Must have

ID	NF9
Requirement	High-level modules include planning layers.
Acceptance Criteria	There is a functioning planning layer defined as high-level module and present on the cloud environment (or server).
Priority	Must have

ID	NF10
Requirement	High-level modules include human-interfacing layers.
Acceptance Criteria	There is a functioning human-interfacing layer defined as high-level module and present on the cloud environment (or server).
Priority	Nice to have

ID	NF11
Requirement	All ROS robots should be able to easily become a collection of web services and be accessible by the task planner.
Acceptance Criteria	A ROS robot is defined as a collection of web services. The task planner uses these web services to produce a plan for a given task.
Priority	Nice to have

ID	NF12
Requirement	For evaluation of the system, at least two robots must be present to test and validate the proposed design.
Acceptance Criteria	Two robots are present to test the system for its full performance, because the system is meant for multi-robot control.
Priority	Must have

Appendix B

Source Code

This appendix shows the source code of the planning system.
The source code is divided over six files:

1. *main.pl*
2. *conditions.pl*
3. *sitcalc.pl*
4. *synsugar.pl*
5. *transfinal.pl*
6. *utils.pl*

B.1 main.pl

The file *main.pl*:

```
% Constraint solving library.
:- use_module(library('clpq')).

% Remove warning about clauses not being together in the same source file
:- discontiguous(trans/4, final/2, start/2, prim_action/1, poss/3, proc/2, holds/2).

% include all files
:- include(sitcalc).
:- include(transfinal).
:- include('../output/experiment_final').
:- include(synsugar).
:- include(conditions).
:- include(utils).

%  Specify a 2 GB limit for the global stack
set_prolog_stack(global, limit(2*10**9)).

% Initial conditions
start(s0,0).

% Start plan for navigate with every argument unbounded
start(Action) :- start_new_plan('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_
multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#goal_plan', Action).

% Get the next action of the plan
get_action(Action) :- do_next_action(Action).
```

```
% Tell the planner an action is finished
finishAction(Action) :- assert(finished(Action)).

failAction(_Action) :- true.

finishAction(FinishedAction, NewAction) :- assert(finished(FinishedAction)),
get_action(NewAction).

failAction(_FailedAction, NewAction) :- get_action(NewAction).
```

B.2 conditions.pl

The file *conditions.pl*:

```
%%
%%  holds(Cond,S):  check whether a condition holds in a situation
%%
%%  This predicate is used to evaluate reified condition terms from
%%  MIndiGolog programs.  It recursively reduces the formula to equivalent
%%  forms which can be tested directly by the prolog theorem prover,
%%  and hence includes the standard prolog negation-as-failure semantics.
%%
holds(and(C1,C2),S) :-
    holds(C1,S), holds(C2,S).
holds(or(C1,C2),S) :-
    holds(C1,S) ; holds(C2,S).
holds(all(V,C),S) :-
    holds(neg(some(V,neg(C))),S).
holds(some(V,C),S) :-
    sub(V,_,C,Cr), holds(Cr,S).
holds(neg(neg(C)),S) :-
    holds(C,S).
holds(neg(and(C1,C2)),S) :-
    holds(or(neg(C1),neg(C2)),S).
holds(neg(or(C1,C2)),S) :-
    holds(and(neg(C1),neg(C2)),S).
holds(neg(all(V,C)),S) :-
    holds(some(V,neg(C)),S).
holds(neg(some(V,C)),S) :-
    \+ holds(some(V,C),S).
holds(P_Xs,S) :-
    P_Xs \= and(_,_), P_Xs \= or(_,_), P_Xs \= neg(_), P_Xs \= all(_,_),
    P_Xs \= some(_,_), sub(now,S,P_Xs,P_XsS), P_XsS.
holds(neg(P_Xs),S) :-
    P_Xs \= and(_,_), P_Xs \= or(_,_), P_Xs \= neg(_), P_Xs \= all(_,_),
    P_Xs \= some(_,_), sub(now,S,P_Xs,P_XsS), \+ P_XsS.
```

B.3 sitcalc.pl

The file *sitcalc.pl*:

```
:- dynamic prim_action/1.

%%
%%  prim_action(A):  define a primitive action
%%
%%  This predicate specifies the terms which represent primitive actions
%%  in the domain.  The following below are examples of both an agent-initiated
```

```
%%  and a natural "no-op" action.  As the action as no effect, successor
%%  state axioms are not necessary.
%%
prim_action(noop(A)) :-
    agent(A).
prim_action(noop).

%%
%%  natural(A):  specify natural actions
%%
%%  For each natural action, this predicate must be defined to represent
%%  that fact.  Actions marked as natural must occur if it is possible for
%%  them to occur.
%%
natural(noop).


%%
%%  actor(Actn,Agt):  performing agent for Actions
%%
%%  This predicate binds Agt to the agent performing primitive action Actn.
%%  It requires that the action not be natural
%%
actor(Actn,Agt) :-
    prim_action(Actn), \+ natural(Actn), functor(Actn,_,N), N > 0, arg(_,Actn,Agt),
agent(Agt).

agent(Agt) :- 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_
multirobot_scheduling/owl/ros_grounding.owl#Robot'(Agt).

%%
%%  actuator(Actn,Agt,Actuator):
%%
actuator(Actn,Agt,Actuator) :-
    prim_action(Actn), \+ natural(Actn), functor(Actn,_,N), N > 1,
    arg(_,Actn,Agt), agent(Agt),
    arg(_,Actn,Actuator), 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multi-
robot_stack/tue_multirobot_scheduling/owl/ros_grounding.owl#Actuator'(Actuator),
    'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multirobot_
scheduling/owl/ros_grounding.owl#hasActuator'(Agt,Actuator).

%%
%%  sensor(Actn,Agt,Sensor):
%%
sensor(Actn,Agt,Sensor) :-
    prim_action(Actn), \+ natural(Actn), functor(Actn,_,N), N > 1,
    arg(_,Actn,Agt), agent(Agt),
    arg(_,Actn,Sensor), 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multi-
robot_stack/tue_multirobot_scheduling/owl/ros_grounding.owl#Sensor'(Sensor),
    'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multirobot_
scheduling/owl/ros_grounding.owl#hasSensor'(Agt,Sensor).

%%
%%  location(Actn,Loc):  location for Actions
%%
%%  This predicate binds Loc to a location matching the primitive action Actn.
%%  It requires that the action not be natural
%%
```

```
location(Actn,Loc) :-
    prim_action(Actn), \+ natural(Actn), functor(Actn,_,N), N > 0, arg(_,Actn,Loc),
loc(Loc).

loc(Loc) :- 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_
multirobot_scheduling/owl/ros_grounding.owl#Location'(Loc).

%%
%%  start(S,T):  start time of situation S
%%
%%  This predicate binds T to the start time of situation S.  This is
%%  defined as the occurance time of the last action performed in the
%%  situation.
%%
%%  The start time of s0 is not defined here, but could be defined by
%%  adding an additional clause for start/2.
%%
start(S,T) :-
    do(_,T,_) = S.


%%
%%  precedes(S1,S2):  ordering over situations
%%
%%  This predicate is true when S2 is reachable from S1 by some finite
%%  sequence of actions.  Note that no situation precedes s0, by definition.
%%
precedes(_,s0) :- fail.
precedes(S1,do(C,T,S2)) :-
    poss(C,T,S2), precedes_eq(S1,S2),
    start(S2,S2start), {S2start =< T}.

%%
%%  precedes_eq(S1,S2):  precedes-or-equals
%%
%%  This predicate is to precedes/2 as <= is to <, it allows for the
%%  two arguments to be equal.
%%
precedes_eq(S1,S2) :-
    S1 = S2 ; precedes(S1,S2).


%%
%%  legal(S1,S2):  checks legality of situation progression
%%
%%  This predicate is true if the situation S2 can legally be reached
%%  from situation S1.  This means that for each transition from S1
%%  to S2, the performed actions were possible and there are no natural
%%  actions that could have occured but didnt.
%%
legal(S,S).
legal(S1,do(C,T,S2)) :-
    legal(S1,S2),
    poss(C,T,S2), start(S2,S2start), {S2start =< T},
    \+ ( natural(NA), poss(NA,T2,S2), \+ member(NA,C), {T2 =< T} ).

%%
%%  legal(S):   checks legality of a situation
```

```
%%
%%  A situation is considered legal if it is legally reachable from the
%%  initial situation.  The initial situation itself is always legal.
%%
legal(s0) :- !.
legal(S) :-
    legal(s0,S).


%%
%%  poss(A,T,S):   possibility of executing an action
%%
%%  The predicate poss/3 must be true whenever it is possible to perform
%%  action A in situation S at time T.  A may be either a primitve action or
%%  a list of actions to be performed concurrently.
%%
%%  The domain axiomatiser is required to provide implementations of
%%  poss/3 for all primitive actions.  Concurrent actions are considered
%%  possible if each constituent action is possible, and conflicts/3
%%  does not hold.
%%

:- dynamic poss/3.

poss([A],T,S) :- % enable action preconditions
    poss(A,T,S).
%poss([_A],_T,_S). % disable action preconditions, actions are always possible
%poss(A,_T,_S) :- A \= [], A \= [_|_]. % disable action preconditions, actions are always
%possible
poss([A|C],T,S) :-
    C \= [], poss_all([A|C],T,S), \+ conflicts([A|C],T,S).


%%
%%  poss_all(C,T,S):  all given actions are possible
%%
%%  This predicate checks that all primitive actions in concurrent action
%%  C are possible in situation S at time T.  It is the basic possibility
%%  check for concurrent actions.
%%
poss_all([],_,_).
poss_all([A|C],T,S) :-
    poss(A,T,S), poss_all(C,T,S).


%%
%%  conflicts(C,T,S):  test for conflicting actions
%%
%%  This predicate must be true if some of the primitive actions in concurrent
%%  action C cannot be executed together is situation S at time T.  The
%%  clause below provides that an empty action never conflicts, other
%%  clauses must be supplied as appropriate.
%%
conflicts([],_,_) :- fail.

%%  Agents cannot do more than one action at a time
conflicts(C,_,_) :-
    member(A1,C), actuator(A1,Agt,A),
```

```
    member(A2,C), actuator(A2,Agt,A),
    A2 \= A1.

%%  Agents cannot do more than one action at a time
conflicts(C,_,_) :-
    member(A1,C), sensor(A1,Agt,S),
    member(A2,C), sensor(A2,Agt,S),
    A2 \= A1.

%%  Locations cannot be given as argument to more than one action at a time
conflicts(C,_,_) :-
    member(A1,C), location(A1,Loc),
    member(A2,C), location(A2,Loc),
    A2 \= A1.

%%
%%  lntp(S,T):   least-natural-time-point for a situation
%%
%%  This predicate determines the least natural time point (LNTP) T for
%%  the given situation S.  This is the earliest time at which it is possible
%%  for a natural action to occur in the situation.  It represents the time
%%  at which, given no outside influences, the situation will change.
%%
lntp(S,T) :-
    natural(A), poss(A,T,S), start(S,SStart), {SStart =< T},
    \+ (natural(A2), poss(A2,T2,S), {T2 < T}).

%%
%%  to_cact(A,C):   convert a primitive to a concurrent action
%%
%%  This predicate can be used as a "cast" operator to turn a primitve
%%  action A into concurrent action C by wrapping it in a list.  If A
%%  is already a concurrent action, it is simply unified with C.
%%
to_cact([],[]).
to_cact([H|T],[H|T]).
to_cact(A,C) :-
    prim_action(A), C = [A].
```

B.4 synsugar.pl

The file *synsugar.pl*:

```
% Syntactic sugar
syn_sugar('http://www.daml.org/services/owl-s/1.2/Process.owl#Sequence'(P1,P2),
sequence(P1,P2)).
syn_sugar('http://www.daml.org/services/owl-s/1.2/Process.owl#Sequence'(L),
sequence(L)).
syn_sugar('http://www.daml.org/services/owl-s/1.2/Process.owl#Split'(P1,P2),
split(P1,P2)).
syn_sugar('http://www.daml.org/services/owl-s/1.2/Process.owl#Split-Join'(P1,P2),
split_join(P1,P2)).
syn_sugar('http://www.daml.org/services/owl-s/1.2/Process.owl#Any-Order'(P1,P2),
any_order(P1,P2)).
syn_sugar('http://www.daml.org/services/owl-s/1.2/Process.owl#Any-Order'(L),
any_order(L)).
syn_sugar('http://www.daml.org/services/owl-s/1.2/Process.owl#Choice'(P1,P2),
choice(P1,P2)).
```

```
syn_sugar('http://www.daml.org/services/owl-s/1.2/Process.owl#Choice'(L), choice(L)).
syn_sugar('http://www.daml.org/services/owl-s/1.2/Process.owl#If-Then-Else'(C,P1,P2),
if_then_else(C,P1,P2)).
syn_sugar('http://www.daml.org/services/owl-s/1.2/Process.owl#Repeat-Until'(C,P),
repeat_until(C,P)).
syn_sugar('http://www.daml.org/services/owl-s/1.2/Process.owl#Repeat-While'(C,P),
repeat_while(C,P)).
syn_sugar(Proc,pcall(Proc)) :- proc(Proc,_).

%%  A program may also transition if it contains syntactic sugar, and is
%%  equivalent to a program that may transition.
trans(D,S,Dp,Sp) :-
    syn_sugar(D,Ds),
    trans(Ds,S,Dp,Sp).

%%  A program is also final if it contains syntactic sugar, and is equivalent
%%  to a program that is final.
final(D,S) :-
    syn_sugar(D,Ds),
    final(Ds,S).
```

B.5 transfinal.pl

The file *transfinal.pl*:

```
%%  It is always legal for an empty program to terminate
final(nil,_).

%%  It is never legal for a program consisting of a single action
%%  to terminate, hence there is no clause for this case.

%%  It is never legal for a program consisting of a test to terminate,
%%  hence there is no clause for this case.

%%  Sequential performance of two programs may terminate only if both
%%  programs may terminate.
final(sequence(D1,D2),S) :-
    final(D1,S), final(D2,S).

%%  Sequential performance of a list of programs may terminate if all programs
%%  in the list may terminate.
final(sequence([]),_).
final(sequence([D]),S) :-
    final(D,S).
final(sequence([D1|Dr]),S) :-
    final(sequence(D1,sequence(Dr)),S).

%%  Split of two programs may always terminate
final(split(_,_),_).

%%  Split of multiple programs may always terminate
final(split(_),_).

%%  Split-join of two programs may terminate only if both programs may terminate.
final(split_join(D1,D2),S) :-
    final(D1,S), final(D2,S).

%%  Split-join of multiple programs may terminate only if all programs may terminate.
```

```
final(split_join([]),_S).
final(split_join([D]),S) :-
    final(D,S).
final(split_join([D1|Dr]),S) :-
    final(D1,S), final(split_join(Dr)).

%%  Any-order may terminate if any sequential performance of the two programs may
%%  terminate.
final(any_order(D1,D2),S) :-
    final(sequence(D1,D2),S)
    ;
    final(sequence(D2,D1),S).

%%  Any-order (list) may terminate if any sequential performance of all programs in
%%  the list may terminate.
final(any_order([]),_). %        It is always legal for an empty order to terminate
final(any_order([D]),S) :-                % there is only one program in the list
    final(D,S).
final(any_order([D1|Dr]),S) :-
    final(sequence(D1,any_order(Dr)),S)
    ;
    (
      member(D2,Dr),
      delete(Dr,D2,NewDr),
      append([D1],NewDr,NewNewDr),
      final(sequence(D2,any_order(NewNewDr)),S)
    )
    .

%%  Nondeterministic choice between two programs may terminate if either
%%  program may terminate.
final(choice(D1,D2),S) :-
    final(D1,S)
    ;
    final(D2,S).

%%  Nondeterministic choice between multiple programs in a list may terminate if any
%%  program may terminate.
final(choice([]),_).        %        It is always legal for an empty choice to terminate
final(choice(L),S) :-
    member(D,L), final(D,S).

%%  If-Then-Else may terminate if the test is true and the true option
%%  may terminate, or the test is false and the false option may terminate.
final(if_then_else(Cond,D1,D2),S) :-
    holds(Cond,S), final(D1,S)
    ;
    holds(neg(Cond),S), final(D2,S).

%%  Repeat-until may terminate if the loop program may terminate, or if the test is true
final(repeat_until(Cond,D),S) :-
    final(D,S)
    ;
    holds(Cond,S).

%%  Repeat-while may terminate if the test is false, or if the
%%  loop program may terminate.
final(repeat_while(Cond,_D),S) :-
```

```
    holds(neg(Cond),S),
    (
      (        % some flags for handling while-loops correctly are asserted
        planning(false), % when not in offline planning mode
        \+ breakBlock(Cond), % if there is no block set on breaking out of a while-loop
        % if there is a break block, then the while-loop cannot terminate and the planner
        % will try to do a transition instead (go to the trans-predicates to see how this
        % is handled further)
        assert(breakBlock(Cond)), % then set this block
        assert(someCheck(Cond)) % also set a check flag
        % check flags are removed the next time the executive asks for an action
      )
      ;
      (
        true
      )
    ).
final(repeat_while(_Cond,D),S) :-
    final(D,S).


%%  A procedure call may terminate if the corresponding body, with arguments
%%  substituted appropriately, may terminate.
final(pcall(PArgs),S) :-
    sub(now,S,PArgs,PArgsS), proc(PArgsS,P), final(P,S).

%%  An offline search can terminate if the program to be searched can
%%  terminate.
final(search(D),S) :-
    final(D,S).

%%  Converts control constructs into a different format (listed)
%%  to see if termination is valid
final(Term,S) :-
    controlConstructConverter(Term, ListedTerm),
    !,
    final(ListedTerm,S).


%%
%%  trans(D,S,Dp,Sp):  program transition is possible
%%
%%  The predicate trans/4 is true when it is possible for situation S
%%  to evolve to situation Sp by executing the first part of program D.
%%  Dp is the part of D that remains to be executed in situation Sp.
%%
%%  Thus, trans/4 defines how execution of a program may be single-stepped
%%  from one situation to another.  The specifics of each individual
%%  clause are explained below.
%%

%%  It is never legal to transition an empty program, hence there
%%  is no clause for this case.

%%  A program consisting of a single (possibly concurrent) action may
%%  transition in several ways, depending on the natural actions which
%%  may occur in situation S.
%%
```

```
%%  * If situation S has no LNTP, perform the action at any time and
%%    set Dp to the empty program
%%  * If situation S has an LNTP, there are three possible transitions:
%%      * If the action has no natural actions and it is possible to
%%        do it before the LNTP, do so and set Dp to the empty program
%%      * Do the natural actions at the predicted time, leaving the
%%        program unaltered
%%      * Do the given action and the natural actions concurrently
%%        at the LNTP, and set Dp to the empty program
%%
trans(C,S,Dp,Sp) :-
    sub(now,S,C,CS), to_cact(CS,CA), start(S,SStart),
    ( lntp(S,LNTP) ->
      (
        % Get the list of LNTP actions
        findall(NA,(natural(NA),poss(NA,LNTP,S)),NActs),
        (
          % Can do them before the LNTP actions
          % This requires that no actions in the set are natural
          (
            \+ ( member(A,CA), natural(A) ),
            {T >= SStart+1}, {T < LNTP}, poss(CA,T,S),
            Sp = do(CA,T,S), Dp = nil
          )
          ;
          % Can do them at the same time
          (
            union(CA,NActs,CANat),
            poss(CANat,LNTP,S),
            Sp = do(CANat,LNTP,S), Dp = nil
          )
          ;
          % Can do the LNTP actions first, leaving program unaltered
          % TODO: this poss() call should always be true, right?
          (
            poss(NActs,LNTP,S),
            Sp = do(NActs,LNTP,S), Dp = C
          )
        )
      )
    ;
      poss(CA,T,S), S = do(CAprev,_,_), getDuration(CAprev, Dur), T is SStart+Dur,
Sp = do(CA,T,S), Dp = nil
    ;
      poss(CA,T,S), S = s0, T is SStart+1, Sp = do(CA,T,S), Dp = nil
    )
    .

%%  A test may transition to the empty program if it holds, leaving the
%%  situation unaltered.
trans(test(Cond),S,Dp,Sp) :-
    holds(Cond,S), S=Sp, Dp=nil
    ;
    lntp(S,LNTP),
    findall(NA,(natural(NA),poss(NA,LNTP,S)),NActs),
    Sp = do(NActs,LNTP,S), Dp = test(Cond).

%%  Sequential execution of two programs may transition by transitioning
```

```
%%  the first program, leaving the remainder the be executed in sequence
%%  with the second.  If the first program may terminate, it is also legal
%%  to transition the second program.
trans(sequence(D1,D2),S,Dp,Sp) :-
    trans(D1,S,D1r,Sp), Dp = sequence(D1r,D2).
trans(sequence(D1,D2),S,Dp,Sp) :-
    final(D1,S), trans(D2,S,Dp,Sp).

%%        Sequential execution of a list of programs
trans(sequence([D]),S,Dp,Sp) :-
    trans(D,S,Dp,Sp).
trans(sequence([D1|Dr]),S,Dp,Sp) :-
    trans(D1,S,D1r,Sp), Dp = sequence(D1r,sequence(Dr)).
trans(sequence([D1|Dr]),S,Dp,Sp) :-
    final(D1,S), trans(sequence(Dr),S,Dp,Sp).

%%  Concurrent execution, or split, may transition in three ways:
%%
%%    * Transition the first program, leaving its remainder to be
%%      executed concurrently with the second program
%%    * Transition the second program, leaving its remainder to be
%%      executed concurrently with the first program
%%    * Find a concurrent action that will transition the first program
%%      and a concurrent action that will transition the second program,
%%      and perform both concurrently.  The remaining program is the
%%      concurrent execution of the remainders of the individual programs
trans_true_split(split(D1,D2),S,Dp,Sp) :-
    step(D1,S,Dr1,do(C1,T,S)),
    step(D2,S,Dr2,do(C2,T,S)),
    \+ ( member(A,C1), member(A,C2), actor(A,_) ),
    union(C1,C2,CT), trans(CT,S,nil,Sp),
    Dp = split(Dr1,Dr2).

trans(split(D1,D2),S,Dp,Sp) :-
    trans_true_split(split(D1,D2),S,Dp,Sp)
    ;
    Dp = split(Dr1,D2), trans(D1,S,Dr1,Sp)
    ;
    Dp = split(D1,Dr2), trans(D2,S,Dr2,Sp).

%%  Split in listed format may transition the same as a normal split, there are only
% more programs.
trans(split([D]),S,Dp,Sp) :-
    trans(D,S,Dp,Sp).
trans(split([D1|D2]),S,Dp,Sp) :-
                D2 \= [_|_],
    trans(split(D1,D2),S,Dp,Sp).
trans(split([D1|Dr]),S,Dp,Sp) :-
    trans(split(D1,split(Dr)),S,Dp,Sp).

%%  Split-join may transition in the same way as split
trans_true_split_join(split_join(D1,D2),S,Dp,Sp) :-
    step(D1,S,Dr1,do(C1,T,S)),
    step(D2,S,Dr2,do(C2,T,S)),
    \+ ( member(A,C1), member(A,C2), actor(A,_) ),
    union(C1,C2,CT), trans(CT,S,nil,Sp),
    Dp = split_join(Dr1,Dr2).
```

```
trans(split_join(D1,D2),S,Dp,Sp) :-
    trans_true_split_join(split_join(D1,D2),S,Dp,Sp)
    ;
    Dp = split_join(Dr1,D2), trans(D1,S,Dr1,Sp)
    ;
    Dp = split_join(D1,Dr2), trans(D2,S,Dr2,Sp).

%%  Split-join in listed format may transition in the same way as
%%  split in listed format
trans(split_join([D]),S,Dp,Sp) :-
    trans(D,S,Dp,Sp).
trans(split_join([D1|D2]),S,Dp,Sp) :-
    D2 \= [_|_],
    trans(split_join(D1,D2),S,Dp,Sp).
trans(split_join([D1|Dr]),S,Dp,Sp) :-
    trans(split_join(D1,split_join(Dr)),S,Dp,Sp).

%%  Any-order may transition if any sequential execution of the two programs
%%  may transition.
trans(any_order(D1,D2),S,Dp,Sp) :-
    trans(sequence(D1,D2),S,Dp,Sp)
    ;
    trans(sequence(D2,D1),S,Dp,Sp).

%%  Any-order (list) may transition if any sequential execution of all programs
%%  in the list may transition.
trans(any_order([D]),S,Dp,Sp) :-    % there is only one program in the list
    trans(D,S,Dp,Sp).
trans(any_order([D1|Dr]),S,Dp,Sp) :-
    trans(D1,S,D1r,Sp), Dp = sequence(D1r,any_order(Dr))
    ;
    (
      member(D2,Dr),
      delete(Dr,D2,NewDr),
      append([D1],NewDr,NewNewDr),
      trans(D2,S,D2r,Sp), Dp = sequence(D2r,any_order(NewNewDr))
    )
    .

%%  Nondeterministic choice of programs may transition if either of the
%%  programs may transition.
trans(choice(D1,D2),S,Dp,Sp) :-
    trans(D1,S,Dp,Sp) ; trans(D2,S,Dp,Sp).

%%  Nondeterministic choice between multiple programs may transition if any
%%  of the programs may transition.
trans(choice(L),S,Dp,Sp) :-
    member(D,L), trans(D,S,Dp,Sp).

%%  If-Then-Else may transition if the test is true and the true option
%%  may transition, or the test is false and the false option may transition.
trans(if_then_else(Cond,D1,D2),S,Dp,Sp) :-
    holds(Cond,S), trans(D1,S,Dp,Sp)
    ;
    holds(neg(Cond),S), trans(D2,S,Dp,Sp).
trans(if_then_else(Cond,D1,nil),S,Dp,Sp) :-
    holds(Cond,S), trans(D1,S,Dp,Sp)
    ;
```

```
    holds(neg(Cond),S), Dp = nil, S=Sp.

%%  Repeat-until may transition to the loop program in sequence with another
%%  loop, as long as the test condition does not hold and the loop program
%%  may transition.
trans(repeat_until(_,D),S,Dp,Sp) :-
    planning(true), trans(D,S,Dp,Sp).
trans(repeat_until(Cond,D),S,Dp,Sp) :-
    % if the condition holds, transition the program once
    % (or one last time, depending how you look at it)
    planning(false), holds(Cond,S), trans(D,S,Dp,Sp).
trans(repeat_until(Cond,D),S,Dp,Sp) :-
    % if the condition does not hold, transition to the loop program in
    % sequence with another loop, as long as (the condition does not hold
    % and) the loop program may transition.
    planning(false), Dp = sequence(Dr,repeat_until(Cond,D)), holds(neg(Cond),S),
trans(D,S,Dr,Sp).

%%  Repeat-while may transition to the loop program in sequence
%%  with another loop, as long as the test condition holds and the loop
%%  program may transition.
trans(repeat_while(Cond,D),S,Dp,Sp) :-
    planning(true), holds(Cond,S), trans(D,S,Dp,Sp).
trans(repeat_while(Cond,D),S,Dp,Sp) :-
    planning(false), Dp = sequence(Dr,repeat_while(Cond,D)), holds(Cond,S),
trans(D,S,Dr,Sp).
% The following rule is needed to ensure that if multiple while-loops with the same
% condition
% are in a procedure, and after the first time an action is finished and breaks the first
% loop, that
% all following while-loops must not be automatically skipped because the condition was
% already
% previously made true. Hence the retraction of the finished-predicate at the end of the
% body of
% this transition-predicate.
trans(repeat_while(Cond,D),S,Dp,Sp) :-
    planning(true),         % if offline planning mode
    holds(neg(Cond),S), % and the condition does not hold
    \+ someCheck(Cond), % the check flag must not be set
    breakBlock(Cond), % there must be a block set on breaking out of a while loop
    retract(breakBlock(Cond)), % this break block is removed now
    Cond = neg(finished(C)), % the condition must be in the form of "action not finished"
    finished(C), % the action of the condition (which is just obtained) must be finished
    retract(finished(C)), % decides that an action is not finished anymore,
    %        so that the next while loop can be accessed:
    trans(repeat_while(Cond,D),S,Dp,Sp).

%%  A proceduce call may transition if the body program, with arguments
%%  substituted in, may transition.
trans(pcall(PArgs),S,Dp,Sp) :-
    sub(now,S,PArgs,PArgsS),
    proc(PArgsS,P), trans(P,S,Dp,Sp).

%%  Converts control constructs into a different format (listed)
%%  to see if a transition is valid
trans(Term,S,Dp,Sp) :-
    controlConstructConverter(Term, ListedTerm),
    !,
```

```
    trans(ListedTerm,S,Dp,Sp).


%%
%%  trans*(D,S,Dp,Sp):  Transitive Closure of Transition Rules
%%
%%  This predicate is true if Dp,Sp are in the transitive closure of
%%  the trans/4 predicate for D,S.  It is a simplistic encoding of the
%%  transitive closure in prolog using a recursive definition.
%%
trans*(D,S,D,S).
trans*(D,S,Dp,Sp) :-
    trans(D,S,Dr,Sr),
    trans*(Dr,Sr,Dp,Sp).


%%
%%  Predicates to convert control constructs into a different format (listed)
%%

%%  The main conversion predicate
controlConstructConverter(Term, ListedTerm) :-
    functor(Term, Construct, Arity),
    canBeListed(Term, Construct),
    Arity \= 2,   %% there is no need for conversion if the arity is two
    getArgs(Term, Args),
    length(Args, Arity),
    functor(ListedTerm, Construct, 1),
    arg(1, ListedTerm, Args).

%%  Predicate to see if the control construct can be listed
%%  (Makes the conversion predicate more generic)
canBeListed(Term, Construct) :-
    member(Construct, [sequence, choice, any_order, split, split_join]),    %% is the
% right control construct
    not(member(Term, [sequence([_|_]), choice([_|_]), any_order([_|_]), split([_|_]),
split_join([_|_])])). %% does not have a list as single argument already (prevents
% infinite loops)

%%  Given a term, this predicate gets all the term's arguments in a list
getArgs(Term,Args) :-
    getArgs(Term,Args,1).

getArgs(_, [], _).
getArgs(Term, [H|T], N0) :-
    arg(N0, Term, H),
    N1 is N0 + 1,
    getArgs(Term, T, N1).
```

B.6 utils.pl

The file *utils.pl*:

```
%%
%%  sub(Name,Value,Old,New):  substitue values in a term
%%
%%  This predicate is true when New is equal to Old with all occurances
%%  of Name replaced by Value - basically, a symbolic substitution
```

```
%%  routine.  For example, it is usually used to produce a result such
%%  as:
%%
%%      sub(now,S,fluent(now),fluent(S)).
%%
sub(_,_,T,Tr) :-
    var(T), Tr = T.
sub(X,Y,T,Tr) :-
    \+ var(T), T = X, Tr = Y.
sub(X,Y,T,Tr) :-
    T \= X, T =.. [F|Ts], sub_list(X,Y,Ts,Trs), Tr =.. [F|Trs].

%%
%%  sub_list(Name,Value,Old,New):  value substitution in a list
%%
%%  This predicate operates as sub/4, but Old and New are lists of terms
%%  instead of single terms.  Basically, it calls sub/4 recursively on
%%  each element of the list.
%%
sub_list(_,_,[],[]).
sub_list(X,Y,[T|Ts],[Tr|Trs]) :-
    sub(X,Y,T,Tr), sub_list(X,Y,Ts,Trs).

%%
%%  step(D,S,Dp,Sp):  single-step a program
%%
%%  This predicate takes a program D and a situation S in which to execute it,
%%  and returns a new situation Sp and remaining program Dp such that the
%%  execution of D has progressed by a single action.  It may be used
%%  repeatedly to find a possible next action to perform for a given program.
%%
step(D,S,Dp,Sp) :-
    %%  Naive implementation is simply:  trans*(D,S,Dp,do(C,T,S))
    %%  This implementation is more efficient as it does not generate
    %%  transitions that go beyond one action from S (which will always fail).
    Sp = do(_,_,S), trans(D,S,Dp,Sp)
    ;
    trans(D,S,Dr,S), step(Dr,S,Dp,Sp).

%%
%%  do(D,S,Sp):  offline execution of MIndiGolog Programs
%%
%%  This predicate takes a program D and starting situation S, and
%%  finds a new situation Sp which can be legally reached by executing
%%  program D to termination.  The actions executed can be retreived
%%  from the situation Sp.
%%
%%  This predicate is capable of backtracking over action choices to
%%  find a legal execution, and so is not suitable for executing programs
%%  on-line.
%%

%% Default do predicate with printing action history (debug info)
do_debug(D,S,Sp) :-
    trans*(D,S,Dp,Sp),
    final(Dp,Sp),
    show_action_history(Sp).
```

```
%% Default do predicate
do(D,S,Sp) :-
    trans*(D,S,Dp,Sp),
    final(Dp,Sp).

%% Do predicate, which always only returns the first solution
%% (also prints action history)
do_first(D,S,Sp) :-
    trans*(D,S,Dp,Sp),
    final(Dp,Sp),
    !, show_action_history(Sp).

%% Do predicate, which returns (all) the cheapest solution(s)
do_cheapest(D, S, Sp) :-
    find_unique(X, do(D,S,X), 100, L), % first, find limited number of solutions
    make_times_list(L, List),   % make a list containing only the durations
    list_min(List, MinT),     % get the shortest duration
    indexOf(List, MinT, I),   % get the index of the shortest duration
    indexOf(L, Sp, I).      % use the index to get the cheapest solution

%% Do predicate, which returns only the first cheapest solution
do_cheapest_first(D, S, Sp) :-
    do_cheapest(D, S, Sp), !.

%% Do predicate, which returns only the first cheapest solution
%% (also prints action history)
do_cheapest_first_debug(D, S, Sp) :-
    do_cheapest(D, S, Sp),
    !, show_action_history(Sp).

%%
%%  Predicates to get one action C at a time of a plan
%%

%% Starts a new plan for program D and returns an action C
start_new_plan(D, C) :-
    retractall(remainder(_)), % first, clear the remainder of any previous plan
    retractall(finished(_)),  % clear all actions marked finished
    retractall(breakBlock(_)), % clear all blocking flags
    retractAllGeneratedProcs, % clear all generated procedures
    determineRobots(D), % determine amount of robots needed for planning and control
    do_action(D, C),  % then plan and return first action
    !.  % cut so only one solution is found

%% Gets the next action C of the previously initiated plan
do_next_action(C) :-
    remainder(Dr),  % get the remainder of the plan
    retract(remainder(Dr)), % erase the remainder from the database
    retractall(someCheck(_)), % retract all check flags
    do_action(Dr, C), % plan and return next action
    !.  % cut so only one solution is found

%%
%%  Predicates to retract all generated procedures
%%  (For enabling the selection mechanism for multi-robotic planning)
%%

%% Finds all generated procedures and retracts them
```

```
retractAllGeneratedProcs :-
    findall(X, (specificPlan(X,_), proc(X,_)), List),
    retractAllProcInList(List).

%% Retracts procedures in a list
retractAllProcInList([]).
retractAllProcInList([H|T]) :-
    retract(proc(H,_)),
    retractAllProcInList(T).

%%
%%  Predicate containing the specific procedure for a composite task
%%  (This is supposed to be parsed knowledge)
%%
specificPlan('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_
multirobot_scheduling/owl/action_domain.owl#goal_plan',
    sequence(
        'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multi-
robot_scheduling/owl/action_domain.owl#navigate'(_, _, A),
        'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multi-
robot_scheduling/owl/action_domain.owl#perceive'(A, _)
    )
).

%%
%%  Predicate containing what needs to be checked on for a composite task
%%  (This is supposed to be parsed knowledge)
%%
checkOnWhat('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_
multirobot_scheduling/owl/action_domain.owl#goal_plan',
    'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multirobot_
scheduling/owl/ros_grounding.owl#Coke'
).

%%
%%  Predicates to determine the amount of robots for which a plan needs to be made
%%

%%  The main predicate for determining the amount of robots
determineRobots(D) :-
    findall(X, 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_
multirobot_scheduling/owl/ros_grounding.owl#Robot'(X), L1),
    length(L1, Length1),
    checkOnWhat(D, CheckFunc),
    functor(CheckTerm, CheckFunc, 1),
    arg(1, CheckTerm, Y),
    findall(Y, CheckTerm, L2),
    length(L2, Length2),
    Min is min(Length1, Length2),
    functor(Procedure, proc, 2),
    arg(1, Procedure, D),
    fillInRobots(D, Procedure, Min),
    assert(Procedure).

%%  Fill-in predicate if only one robot is needed
fillInRobots(D, Procedure, Number) :-
    Number = 1,
    specificPlan(D, Value),
```

```
    arg(2, Procedure, Value).

%%  Fill-in predicate if more than one robot is needed. This creates a split-join
% construct.
fillInRobots(D, Procedure, Number) :-
    functor(SplitJoin, split_join, Number),
    fillSplitArguments(D, SplitJoin, Number),
    arg(2, Procedure, SplitJoin).

%%  Predicate to fill in the arguments of the previously created split-join construct.
fillSplitArguments(_, _, 0).
fillSplitArguments(D, SplitJoin, Number) :-
    specificPlan(D, Value),
    arg(Number, SplitJoin, Value),
    NewNumber is Number - 1,
    fillSplitArguments(D, SplitJoin, NewNumber).

%%
%%  Predicate to plan for a program D and return the first action to execute C
%%
do_action(D, C) :-
    set_planning(true),
    do_cheapest_first(D,s0,Sp), % first plan the whole plan
    show_action_history(Sp), % show (debug) output information about the solution
    set_planning(false),
    get_first_action(Sp, C),  % then get the first action of the plan
    step(D, s0, Dr, do(C,_,s0)),  % do a step forward in the plan (using the first action)
    assert(remainder(Dr)),  % assert the remainder of the program to be executed
    !. % cut so only one solution is found

%%
%%  Predicate that sets the planning variable
%%  X should be true or false
%%
set_planning(X) :-
    retractall(planning(_)),
    assert(planning(X)).

%%
%%   Predicate that backtracks a plan to get the first action F
%%
get_first_action(do(F,_,s0),F).
get_first_action(do(_,_,S),F) :- get_first_action(S,F).

%%
%%  Predicates that resets the "finished"-fluents, except for the
%%  fluent finished(NotThis), that one remains untouched.
%%

%% to start resetting fluents
resetFinishedFluents(NotThis) :-
    findall(finished(X), finished(X), L),
    resetFinishedFluent(NotThis, L),
    !.        % cut so reset is only done once

%% to continue recursively resetting fluent after fluent in the given list
resetFinishedFluent(NotThis, [H|T]) :-
    H \= finished(NotThis),
```

```
    retract(H),
    resetFinishedFluent(NotThis, T).
resetFinishedFluent(NotThis, [H|T]) :-
    H = finished(NotThis),
    resetFinishedFluent(NotThis, T).
resetFinishedFluent(_, []).

%%
%%  Temporary predicate for printing the action history of a situation,
%%  for debugging purposes only.
%%
show_action_history(s0) :-
    nl.
show_action_history(do(C,T,S)) :-
    show_action_history(S),
    ( inf(T,MinT) ->
        T = MinT
    ;
        start(S,SStart), T = SStart
    ),
    write('do '), write(C), write(' at '), write(T), nl.

%%
%%  Predicate for getting the time T of a situation
%%
get_time(do(_,T,S),T) :-
    inf(T,MinT) ->
        T = MinT
    ;
        start(S,SStart), T = SStart.

%%
%%  Predicate for making a list of the durations of the solutions
%%
make_times_list(Solutions, List) :-
    make_times_list(Solutions, [], List).

make_times_list([], List, List).
make_times_list([Sol|Tail], Temp, List) :-
    get_time(Sol, Time),        % get the duration
    append(Temp, [Time], NewTemp),
    make_times_list(Tail, NewTemp, List).

%%
%%  Predicate for getting the minimum value (Min) in a list ([L|Ls])
%%
list_min([L|Ls], Min) :-
    list_min(Ls, L, Min).

list_min([], Min, Min).
list_min([L|Ls], Min0, Min) :-
    Min1 is min(L, Min0),
    list_min(Ls, Min1, Min).

%%
%%  Getting the index of the element in a list
%%  Can also be used for getting the element at index in a list
%%  (Or even for checking if element is at index in a list)
```

```
%%
indexOf([Element|_], Element, 0). % We found the element
indexOf([_|Tail], Element, Index):-
  indexOf(Tail, Element, Index1), % Check in the tail of the list
  Index is Index1+1.  % and increment the resulting index

%%
%%  Findall with limited solutions
%%  Provided by Jan Wielemaker
%%  Source: https://groups.google.com/forum/#!topic/comp.lang.prolog/hYTmHjrjQq8
%%
:- meta_predicate
find_unique(-, :, +, -).

find_unique(T, G, inf, Ts) :- !,
  findall(T, G, Raw),
  sort(Raw, Ts).
find_unique(T, G, Max, Ts) :-
  empty_nb_set(Set),
  State = count(0),
  ( G,
    add_nb_set(T, Set, true),
    arg(1, State, C0),
    C is C0 + 1,
    nb_setarg(1, State, C),
    C == Max
    -> true
    ; true
  ),
  nb_set_to_list(Set, Ts).


%%
%%  Predicates for getting duration of actions
%%
getDuration([A], Dur) :-
  getDuration(A, Dur).
getDuration([A|C], Dur) :-
  C \= [], getDuration(A, DurA), getDuration(C, DurC), Dur is max(DurA, DurC).

getDuration(A, Dur) :-
  A \= [], A \= [_|_], hasDuration(A, Dur).
getDuration(A, Dur) :-
  A \= [], A \= [_|_], \+ hasDuration(A, _), Dur = 1.

hasDuration(A, Dur) :-
  canCalculateDuration(A),
  arg(_, A, Arg1),
  arg(_, A, Arg2),
  Arg1 \= Arg2,
  getPos(Arg1, Pos1),
  getPos(Arg2, Pos2),
  calcDistancePow(Pos1, Pos2, Dur).

canCalculateDuration(A) :-
  member(A, ['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_
multirobot_scheduling/owl/action_domain.owl#Perform_1'(_,_)]).
```

```
getPos('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multi-
robot_scheduling/owl/action_domain.owl#AMIGO_1', pos(-1,0)).
getPos('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multi-
robot_scheduling/owl/action_domain.owl#PICO_1', pos(1,0)).
getPos('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multi-
robot_scheduling/owl/ros_grounding.owl#Loc_1', pos(-5,0)).
getPos('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multi-
robot_scheduling/owl/ros_grounding.owl#Loc_2', pos(5,0)).


%%
%%  Predicates to calculate distance between objects
%%

%% 2D space
calcDistance(pos(X1,Y1),pos(X2,Y2),Dist) :-
    A is X2-X1, B is Y2-Y1,        Dist is sqrt((A*A)+(B*B)).

%% 3D space
calcDistance(pos(X1,Y1,Z1),pos(X2,Y2,Z2),Dist) :-
    A is X2-X1, B is Y2-Y1, C is Z2-Z1,        Dist is sqrt((A*A)+(B*B)+(C*C)).

%% 2D space (power of 2)
calcDistancePow(pos(X1,Y1),pos(X2,Y2),Dist) :-
    A is X2-X1, B is Y2-Y1,        Dist is (A*A)+(B*B).

%% 3D space (power of 2)
calcDistancePow(pos(X1,Y1,Z1),pos(X2,Y2,Z2),Dist) :-
    A is X2-X1, B is Y2-Y1, C is Z2-Z1,        Dist is (A*A)+(B*B)+(C*C).

%% Compares which location is closest to a target location and returns that location
compareDistance(Loc1, Loc2, TargetLoc, ClosestToTarget) :-
    calcDistancePow(Loc1, TargetLoc, Dist1),
    calcDistancePow(Loc2, TargetLoc, Dist2),
    (
      (
        Dist1 =< Dist2,
        ClosestToTarget = Loc1
      )
      ;
      (
        Dist2 < Dist1,
        ClosestToTarget = Loc2
      )
    ).

%% Finds in a list the location that is closest to a target location
findNearestLoc([ClosestToTarget|[]], _, ClosestToTarget).
findNearestLoc([Loc|T], TargetLoc, ClosestToTarget) :-
    findNearestLoc(T, TargetLoc, Closest),
    getPos(Loc, Pos1),
    getPos(Closest, Pos2),
    (
      (
        functor(TargetLoc, pos, _),
        compareDistance(Pos1, Pos2, TargetLoc, ClosestPos)
      )
      ;
```

```
  (
    getPos(TargetLoc, Pos3),
    compareDistance(Pos1, Pos2, Pos3, ClosestPos)
  )
),
getPos(ClosestToTarget, ClosestPos).
```

Appendix C

Test Report

This appendix is an extension to section 7.1 and section 7.2 and describes all tests and their results in full detail.

This test report first explains the test cases in the next section. This is followed by the results from the tests based on the test cases.

C.1 Test Cases

The test cases are divided in five groups to fit with the contributions of the implemented system as described in section 5.1. Therefore, the groups are as follows:

1. OWL-S Control Constructs
2. Temporal Planning
3. Multi-Robot Control
4. Robot Capabilities
5. Online Planning

The test cases are described in the next sections. Every group has its own subsection. Every test also has its own test ID, such that the test results are more easily coupled with the test descriptions.

Several test cases require custom procedures to be created. Each procedure described in the following sections needs to be in the same form as procedures are generally parsed by the *'OWL-S ontology parser'*. This ontology parser is not in the scope of the project described here and therefore not described in this thesis. If the reader wants to know more about the parser, then he or she is referred to the work of *Jean-Pierre Denissen* [13], who made the OWL-S parser for the planning system for multi-robot control as described here.

Also, arbitrary numbers are used in most test cases. The reason for this is to ensure that the proofs of the test cases (i.e. the results) are complete and applicable to all possible situations, instead of being only applicable to singular cases.

The following sections are in the same order as the test case groups are presented above.

C.1.1 OWL-S Control Constructs

The first group of test cases are meant to validate that the OWL-S control constructs are implemented conform to the official OWL-S description [51]. Every transition and termination predicate needs to be tested, but also the *controlConstructConverter*/2-predicate, which converts control constructs into a different format. Both the implementations of these control constructs and the converter predicate are described in section 6.2.

Control Constructs

There are a total of eight OWL-S control constructs implemented in the planner, and every construct needs its own test. These tests have test ID A1.1 to A1.8.

The constructs are:

1. Sequence
2. Split
3. Split+Join
4. Choice
5. Any-Order
6. If-Then-Else
7. Repeat-While
8. Repeat-Until

For each construct, a simple procedure needs to be made to validate that its behavior is as expected. Each procedure must contain a control construct and at least two arbitrary non-composite task. The procedures are than provided to the planner. The planner needs to produce a plan for each procedure. A simulated executive program keeps asking the planner for the next action, until the planner does not give an action anymore.

Test A1.1
Test Description:
Test to validate the behavior of the sequence control construct.

Procedure for test:

```
proc('test_A1_1',
    sequence('Arbitrary_Action_1'(_), 'Arbitrary_Action_2'(_))
).
```

Expected Behavior:
It is expected that a valid plan can be found, and eventually be terminated by the simulated executive.

Test A1.2
Test Description:
Test to validate the behavior of the split control construct.

Procedure for test:

```
proc('test_A1_2',
    split('Arbitrary_Action_1'(_), 'Arbitrary_Action_2'(_))
).
```

Expected Behavior:
It is expected that a valid plan can be found, and eventually be terminated by the simulated executive.

Test A1.3
Test Description:
Test to validate the behavior of the splt+join control construct.

Procedure for test:

```
proc('test_A1_3',
    split_join('Arbitrary_Action_1'(_), 'Arbitrary_Action_2'(_))
).
```

Expected Behavior:
It is expected that a valid plan can be found, and eventually be terminated by the simulated executive.

Test A1.4
Test Description:
Test to validate the behavior of the choice control construct.

Procedure for test:

```
proc('test_A1_4',
    choice('Arbitrary_Action_1'(_), 'Arbitrary_Action_2'(_))
).
```

Expected Behavior:
It is expected that a valid plan can be found, and eventually be terminated by the simulated executive.

Test A1.5
Test Description:
Test to validate the behavior of the any-order control construct.

Procedure for test:

```
proc('test_A1_5',
    any_order('Arbitrary_Action_1'(_), 'Arbitrary_Action_2'(_))
).
```

Expected Behavior:
It is expected that a valid plan can be found, and eventually be terminated by the simulated executive.

Test A1.6.1
Test Description:
Test to validate the behavior of the if-then-else control construct. It is meant to show that the first program is executed when the condition is true.

Procedure for test:

```
proc('test_A1_6_1',
    if_then_else(true, 'Arbitrary_Action_1'(_), 'Arbitrary_Action_2'(_))
).
```

Expected Behavior:
It is expected that a valid plan can be found, and eventually be terminated by the simulated executive.

Test A1.6.2
Test Description:
Test to validate the behavior of the if-then-else control construct. It is meant to show that the second program is executed when the condition is false.

Procedure for test:

```
proc('test_A1_6_2',
    if_then_else(false, 'Arbitrary_Action_1'(_), 'Arbitrary_Action_2'(_))
).
```

Expected Behavior:
It is expected that a valid plan can be found, and eventually be terminated by the simulated executive.

Test A1.7
Test Description:
Test to validate the behavior of the repeat-while control construct. The simulated executive needs to execute the body of the while-loop at least once and then inform the planner that the while-condition is false such that it can break out of the loop.

Procedure for test:

```
proc('test_A1_7',
    repeat_while(neg(finished('Arbitrary_Action_1'(A))), 'Arbitrary_Action_1'
        (A))
).
```

Expected Behavior:
It is expected that a valid plan can be found, and eventually be terminated by the simulated executive.

Test A1.8

Test Description:

Test to validate the behavior of the repeat-until control construct. The simulated executive needs to execute the body of the repeat-until-loop at least twice and then inform the planner that the until-condition is true such that it can break out of the loop. It needs to be executed at least twice, because, since the body of an until-loop is always executed at least once, this proofs that the plan indeeds gets in a loop.

Procedure for test:

```
proc('test_A1_8',
    repeat_until(finished('Arbitrary_Action_1'(A)), 'Arbitrary_Action_1'(A))
).
```

Expected Behavior:

It is expected that a valid plan can be found, and eventually be terminated by the simulated executive.

Control Construct Conversion Predicate

The control construct conversion predicate (*controlConstructConverter*/2) needs to be tested to proof that it indeed converts multiple parameters into a list and that a plan can be found with the converted control construct.

Test A2

Test Description: An arbitrary control construct is taken for this test. This construct needs to have an arbitrary number of arguments n, such that $n > 0 \wedge n \neq 2$. The *controlConstructConverter*/2-predicate is then called with the control construct provided as first argument. The second argument of the converter predicate is an unbound variable: V.

Procedure for test:

```
proc('test_A2',
    sequence('Arbitrary_Action_1'(_), 'Arbitrary_Action_2'(_), '
        Arbitrary_Action_3'(_))
).
```

Expected Behavior:

The expected behavior of this predicate is that after calling the predicate, the unbound variable V is bound to the converted control construct. This converted control construct then only has one parameter, which is a list with a length equal to the arity of the control construct before conversion. The length of a list can be checked with the predicate *length*/2 and the arity of the control construct can be obtained with the $functor$/3-predicate.

C.1.2 Temporal Planning

The second group of test cases are made to validate that temporal planning is functioning the same way as it is designed. It includes tests to show that the planner handles actions with duration correctly, robustness tests for which the intend is to find the limit of temporal planning, and a test to validate that the planner always returns the cheapest plan.

Actions with Durations

The tests to show that the planner handles actions with durations correctly are described first. There are two groups of tests: the first group only contains one test showing solely sequential execution and the second group contains eight tests with concurrent execution.

Test B1.1

Test Description:

The *test with sequential execution* includes a simple procedure with a sequence construct and an arbitrary number of non-composite tasks n in the sequence, such that $n > 1$. This procedure is provided to the planner. This test is essentially the same as test A1.1, therefore, no procedure is given here.

Expected Behavior:

The expected behavior is that the planner produces a valid plan in which the actions are in the same order as given in the sequence construct, and the start time of every subsequent action succeeds the start time of its predecessor.

Test B1.2

Test Description:

The *test with concurrent execution* includes a simple procedure with a split construct, at least two sequence constructs inside the split construct, and an arbitrary number of non-composite tasks n in each sequence constructs, such that $n > 1$. This procedure is provided to the planner.

Expected Behavior:

The expected behavior is that the planner produces a valid plan in which the sequences are to be executed in parallel, and the behavior for each separate sequence is expected to be the same as test B1.1.

Test B1.2.1

Test Description:

Variation of test B1.2 with two different branches in the split construct and only anonymous variables.

Procedure for test:

```
proc('test_B1_2_1',
    split(
        sequence('Arbitrary_Action_1'(_), 'Arbitrary_Action_2'(_)),
        sequence('Arbitrary_Action_3'(_), 'Arbitrary_Action_4'(_))
    )
).
```

Expected Behavior:

The expected behavior is that the planner produces a valid plan in which the sequences are to be executed in parallel, and the behavior for each separate sequence is expected to be the same as test B1.1. Also, it is expected that the same robot is used for every action.

Test B1.2.2

Test Description:

Variation of test B1.2 with two different branches in the split construct and all variables are the same.

Procedure for test:

```
proc('test_B1_2_2',
    split(
        sequence('Arbitrary_Action_1'(A), 'Arbitrary_Action_2'(A)),
        sequence('Arbitrary_Action_3'(A), 'Arbitrary_Action_4'(A))
    )
).
```

Expected Behavior:

The expected behavior is that the planner produces a valid plan in which the sequences are to be executed in parallel, and the behavior for each separate sequence is expected to be the same as test B1.1. Also, it is expected that the same robot is used for every action.

Test B1.2.3

Test Description:

Variation of test B1.2 with two identical branches in the split construct and only anonymous variables.

Procedure for test:

```
proc('test_B1_2_3',
    split(
        sequence('Arbitrary_Action_1'(_), 'Arbitrary_Action_2'(_)),
        sequence('Arbitrary_Action_1'(_), 'Arbitrary_Action_2'(_))
```

```
    )
).
```

Expected Behavior:

The expected behavior is that the planner produces a valid plan in which the sequences are to be executed in parallel, and the behavior for each separate sequence is expected to be the same as test B1.1. Also, it is expected that two different robots are used by the planner.

Test B1.2.4

Test Description:

Variation of test B1.2 with two identical branches in the split construct and all variables are the same.

Procedure for test:

```
proc('test_B1_2_4',
    split(
        sequence('Arbitrary_Action_1'(A), 'Arbitrary_Action_2'(A)),
        sequence('Arbitrary_Action_1'(A), 'Arbitrary_Action_2'(A))
    )
).
```

Expected Behavior:

The expected behavior is that the planner produces a valid plan in which the sequences are to be executed in parallel, and the behavior for each separate sequence is expected to be the same as test B1.1. Also, it is expected that the same robot is used for every action.

Test B1.2.5

Test Description:

Variation of test B1.2 with two different branches in the split construct, an additional argument to enable conflicts between actions, and only anonymous variables.

Procedure for test:

```
proc('test_B1_2_5',
    split(
        sequence('Arbitrary_Action_5'(_,_), 'Arbitrary_Action_6'(_,_)),
        sequence('Arbitrary_Action_7'(_,_), 'Arbitrary_Action_8'(_,_))
    )
).
```

Expected Behavior:

The expected behavior is that the planner produces a valid plan in which the sequences are to be executed in parallel, and the behavior for each separate sequence is expected to be the same as test B1.1. Also, it is expected that two different robots are used by the planner.

Test B1.2.6

Test Description:

Variation of test B1.2 with two different branches in the split construct, an additional argument to enable conflicts between actions, and all variables that can match with a robot are the same.

Procedure for test:

```
proc('test_B1_2_6',
    split(
        sequence('Arbitrary_Action_5'(A,_), 'Arbitrary_Action_6'(A,_)),
        sequence('Arbitrary_Action_7'(A,_), 'Arbitrary_Action_8'(A,_))
    )
).
```

Expected Behavior:

The expected behavior is that the planner produces a valid plan in which the sequences are to be executed in sequence (because of the conflicting actions), and the behavior for each separate sequence is expected to

be the same as test B1.1. Also, it is expected that the same robot is used for every action.

Test B1.2.7

Test Description:

Variation of test B1.2 with two identical branches in the split construct, an additional argument to enable conflicts between actions, and only anonymous variables.

Procedure for test:

```
proc('test_B1_2_7',
    split(
        sequence('Arbitrary_Action_5'(_,_), 'Arbitrary_Action_6'(_,_)),
        sequence('Arbitrary_Action_5'(_,_), 'Arbitrary_Action_6'(_,_))
    )
).
```

Expected Behavior:

The expected behavior is that the planner produces a valid plan in which the sequences are to be executed in parallel, and the behavior for each separate sequence is expected to be the same as test B1.1. Also, it is expected that two different robots are used by the planner.

Test B1.2.8

Test Description:

Variation of test B1.2 with two identical branches in the split construct, an additional argument to enable conflicts between actions, and all variables that can match with a robot are the same.

Procedure for test:

```
proc('test_B1_2_8',
    split(
        sequence('Arbitrary_Action_5'(A,_), 'Arbitrary_Action_6'(A,_)),
        sequence('Arbitrary_Action_5'(A,_), 'Arbitrary_Action_6'(A,_))
    )
).
```

Expected Behavior:

The expected behavior is that the planner produces a valid plan in which the sequences are to be executed in sequence (because of the conflicting actions), and the behavior for each separate sequence is expected to be the same as test B1.1. Also, it is expected that the same robot is used for every action.

Planner Robustness

The following test case describes the robustness tests. The intention of these tests is to find the planner's limit for temporal planning. In other words, it is meant to find out when temporal planning goes wrong. It is important to know if the plan stays valid even when the real execution times of actions are completely different from the expected execution time of these actions. It is not probable that the planner's model of the world is completely true, therefore, there are always situations when the expected execution time of an action is wrong.

There are six robustness tests, they have test ID B2.1 to B2.6. The first three are with sequential execution and the last three are with concurrent execution. The following applies to both sets of three tests: in the first test, the actual durations are a lot shorter than assumed, for the second test the durations are a lot longer than assumed, and it varies between both in the third test. Communication between a (simulated) executive program is vital for these tests, as the actual durations of the actions must be presented in some way. A simulated executive program is required for every robustness test. Every test starts with the executive querying the planner for a plan, and for every test, the actual duration of actions is simulated by a timer. The executive only queries for the next action after the timer is finished.

Table C.1 displays the robustness tests. Tests B2.1 to B2.3 include a procedure with a sequence construct and tests B2.4 to B2.6 include a procedure with a split construct containing two branches with each a sequence construct. Every sequence construct contains an arbitrary number of non-composite tasks n. Every non-composite task has a duration d of arbitrary length. This procedure is provided to the planner. The tests either have all timers in the simulated executive set to go off after a timespan that is arbitrarily

shorter than the expected durations as provided to the planner, all timers in the simulated executive set to go off after a timespan that is arbitrarily longer than the expected durations as provided to the planner, or have some timers in the simulated executive set to go off after a timespan that is arbitrarily longer than the expected durations as provided to the planner and all other timers set to go off after an arbitrarily shorter timespan (denoted as "mix" in the table).

Test ID	**Construct**	n	d	**Timer Set At**
Test B2.1	Sequence	$n > 1$	$d > 1$	Arbitrarily shorter than expected.
Test B2.2	Sequence	$n > 1$	$d > 0$	Arbitrarily longer than expected.
Test B2.3	Sequence	$n > 1$	$d > 1$	Mix of shorter and longer.
Test B2.4	Split	$n > 1$	$d > 1$	Arbitrarily shorter than expected.
Test B2.5	Split	$n > 1$	$d > 1$	Arbitrarily longer than expected.
Test B2.6	Split	$n > 1$	$d > 1$	Mix of shorter and longer.

Table C.1: Tests B2.1 to B2.6

Procedure for tests B2.1 - B2.3:

```
proc('test_B2_1',
    sequence('Arbitrary_Action_1'(_), 'Arbitrary_Action_2'(_), '
        Arbitrary_Action_3'(_))
).
```

Procedure for tests B2.4 - B2.6:

```
proc('test_B2_4',
    split(
        sequence('Arbitrary_Action_1'(_), 'Arbitrary_Action_2'(_), '
            Arbitrary_Action_3'(_)),
        sequence('Arbitrary_Action_1'(_), 'Arbitrary_Action_2'(_), '
            Arbitrary_Action_3'(_))
    )
).
```

Additional predicate for these tests:

```
hasDuration(_, 10).
```

This additional predicate defines that all action durations are by default 10 seconds.

Expected Behavior:
The expected behavior of these tests is that planner continues to function properly.

The Cheapest Plan

The following test case is meant to proof that the planner always returns the cheapest plan from all valid plans found.

Test B3
Test Description:
At least two procedures must be created for this test. The procedures are both made to accomplish the same task but vary in duration. Both procedures contain an arbitrary number of control constructs and non-composite actions, but their total durations must significantly differ. The procedures are provided to the planner. The durations of all valid plans found must be logged to proof that the cheapest plan is indeed returned by the planner. The planner then runs multiple times to proof that it returns the cheapest plan on every attempt and to decrease the chance that the result is coincidental.

Procedures for test:

```
proc('test_B3',
    sequence('Arbitrary_Action_1'(_), 'Arbitrary_Action_2'(_))
).
```

```
proc('test_B3',
    sequence('Arbitrary_Action_3'(_), 'Arbitrary_Action_4'(_))
).

proc('test_B3',
    sequence('Arbitrary_Action_5'(_), 'Arbitrary_Action_6'(_))
).
```

Additional predicate for test:

```
hasDuration(_, Dur) :- RNo is random(9), Dur is RNo+1.
```

Expected Behavior:
The expected behavior is that the planner always returns the cheapest plan from all valid plans found.

C.1.3 Multi-Robot Control

The group of test cases described in this section are made to validate that the planner uses all available robots in an as efficient fashion as possible. It includes tests with situations where there are too many robots, another with not enough robots, and a test with a focus on selecting robots for concurrent execution. All tests described here include a procedure with an abstract task 'take order', which is represented by a non-composite action to simplify the test case. This procedure is provided to the planner.

Tests C1 to C3 all make use of the following predicates:

```
specificPlan('test_C',
    'Arbitrary_Action_1'(_,_)
).

checkOnWhat('test_C',
    'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
        tue_multirobot_scheduling/owl/ros_grounding.owl#Location'
).
```

Too Many Robots

There are two tests in the test case of "too many robots". For both tests, the task is to have robots get orders from people and there are more robots available than there are "order locations" in the environment. The tests differ in that the second test has no order locations in the environment at all. These tests are described in more detail next.

Test C1.1

Test Description:

In the *first test*, at least two robots are available and one 'order location' is present in the environment. The planner should choose no more robots than are needed. The expected result is that the planner selects one robot to go to the 'order location' and ignores all other robots.

Two robots:

```
'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multirobot_sched
'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multirobot_sched
```

One location:

```
'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multirobot_sched
```

Expected Behavior:
The resulting plan must be a plan for only one robot.

Test C1.2

Test Description:

For the *second test*, there is at least one robot available and no 'order location' present.

One robot:

```
'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multirobot_sched
```

No locations.

Expected Behavior:
The planner is expected to fail in finding a valid plan in this case.

Not Enough Robots

There are also two test in the test case of "not enough robots". For both tests, the task is to have robots get orders from people, but there are *less* robots available than there are 'order locations' in the environment. These tests differ in that the second test has no robots available at all. These tests are described in more detail next.

Test C2.1
Test Description:
In the *first test*, only one robot is available and at least two 'order locations' are present. The planner still needs to find a valid plan.

One robot:

```
'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multirobot_sched
```

Two locations:

```
'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multirobot_sched
'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multirobot_sched
```

Expected Behavior:
The expected result is that the planner selects the robot to go to one of the 'order locations'. No robot shall go to the other order locations.

Test C2.2
Test Description:
For the *second test*, there are no robots available at all, and an arbitrary number of 'order locations' are present in the environment.

No robots.
One location:

```
'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multirobot_sched
```

Expected Behavior:
The expected result is that the planner cannot find a valid plan.

Robot Teamwork

This test case has the purpose to validate that the planner makes concurrent plans when it can use more than one robot.

Test C3
Test Description:
For this test, at least two robots are available to the planner and the amount of 'order locations' that are present in the environment must be equal to the amount of available robots.

Two robots:

```
'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multirobot_sched
'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multirobot_sched
```

Two locations:

```
'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multirobot_sched
'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multirobot_sched
```

Expected Behavior:
The expected result is that the planner selects all available robots and starts to plan for concurrent execution. A valid concurrent plan is then produced.

C.1.4 Robot Capabilities

There are two test cases in the 'robot capabilities group', these are 1. to validate that the planner makes the right choices based on robot capabilities and 2. to validate that certain robot capabilities create conflicts and that the planner handles these conflicts correctly.

Making Choices based on Capabilities

There are three tests in this test case. All tests in this test case include a simple procedure with one abstract task 'pick up an object', a non-composite action represents this task to simplify the test case. The robot capability that a robot has arms is required for this task. This procedure is provided to the planner for each test.

Test D1.1
Test Description:
There are two robots available in the *first test*. One of these robots has the robot capability that it has arms and the other robot does not have this robot capability.
Procedure for test

```
proc('test_D1_1',
    'Arbitrary_Action_1'(_,_)
).
```

Expected Behavior:
The expected result is that the planner selects the robot with arms to do the task where arms are needed. A valid plan produced by the planner needs to show this expected result.

Test D1.2
Test Description:
The *second test* is the same as the first test, with the exception that there are now two different tasks to be executed. The new second task does not require any robot capabilities.

Expected Behavior:
The expected result is that the planner selects the robot with arms to do the task where arms are needed, and the other robot for the other task. A valid plan produced by the planner needs to show this expected result.

Test D1.2.1
Test Description:
Variation of test D1.2 in which the procedure for this test is defined as follows:

```
proc('test_D1_2_1',
    split_join(
      'Arbitrary_Action_1'(_,_),
      'Arbitrary_Action_3'(_)
    )
).
```

Test D1.2.2
Test Description:
Variation of test D1.2 in which the procedure for this test is defined as follows:

```
proc('test_D1_2_2',
    split_join(
      'Arbitrary_Action_3'(_),
      'Arbitrary_Action_1'(_,_)
    )
).
```

Test D1.3

Test Description:

The *third test* is the same as the first test, with the exception that there is only one robot available. This robot however, does not have the right robot capability to perform the given task.

Procedure for test:

```
proc('test_D1_3',
    'Arbitrary_Action_1'(_,_)
).
```

Expected Behavior:

The expected result is that the planner cannot find a valid plan.

Conflicts

The following test case is meant to validate that the planner handles conflicts between actions correctly. It contains only one test. The test needs to prove that one robot cannot do two conflicting actions at the same time.

Test D2

Test Description:

This test includes a procedure with a split construct, which contains two non-composite actions which need to be executed in parallel. These two actions conflict with each other due to robot capabilities. The procedure is provided to the planner.

Procedure for test:

```
proc('test_D2',
    split_join(
      'Arbitrary_Action_1'(A,B),
      'Arbitrary_Action_2'(A,B)
    )
).
```

Expected Behavior:

The expected behavior is that the planner produces a valid plan in which the two actions are to be performed in sequence instead concurrently.

C.1.5 Online Planning

The last group of test cases is about online planning. These test cases are meant to validate that the planner handles communication between itself and the executive correctly and to clearly show how the feedback loop functions, which is the essential part of online planning. It includes two test cases: the first is to show the communication between the planner and the executive, and the second is to show that the planner can get out of while-loops if it gets feedback from the executive that the blocking action succeeded.

Feedback Loop

The test case about the feedback loop consists of one test. It is created to show the communication between the planner and the simulated executive program.

Test E1

Test Description:

This test includes an arbitrary procedure which is provided to the planner. The focus of the test is to show the following communication features:

- The executive queries for a new plan.
- The planner initiates a new plan and returns the first action to the executive.
- The executive asks for the next action.
- The planner returns the next action of a plan.

Procedure for test:

```
proc('test_E1',
    sequence('Arbitrary_Action_1'(_), 'Arbitrary_Action_2'(_))
).
```

Expected Behavior:

The expected result is that all these communication features function correctly.

Breaking out of While-Loops

The test case of breaking out of while-loops also consists of only one test. The purpose of this test is to show that the planner can only get out of a while-loop if the executive tells the planner that the blocking action (the action given as precondition of the loop) succeeded.

Test E2

Test Description:

This test includes a procedure with a sequence of three arbitrary non-composite actions, of which the second arbitrary action is inside a repeat-while construct. The condition of the while-loop is that the arbitrary action inside the while-loop is not finished. This means that breaking out of the while-loop is only possible if this action is successfully executed. The feedback of the (simulated) executive must contain the information that the action is successfully executed.

Procedure for test

```
proc('test_E2',
    sequence(
        'Arbitrary_Action_1'(_),
        repeat_while(neg(finished('Arbitrary_Action_2'(A))),
            'Arbitrary_Action_2'(A)
        ),
        'Arbitrary_Action_3'(_)
    )
).
```

Expected Behavior:

The progress flow is expected to be as follows:

1. The planner will always do the first action.
2. The first time the second action needs to be executed, the (simulated) executive should tell the planner that it failed.
3. The planner tells the executive to execute the second action again upon getting the feedback that the action failed.
4. The second time the second action needs to be executed, the (simulated) executive should tell the planner it succeeded.
5. The planner should return the third action now.

C.2 Test Results

This section displays the results of the previously defined tests. The test results are divided in the same five groups as the previously defined test cases.

C.2.1 OWL-S Control Constructs

This section displays the results of tests A1 to A2.

Control Constructs

This section displays the results of tests A1.1 to A1.8.

Test A1.1
Test successful: yes.

Execution of test:

```
?- start_new_plan('test_A1_1', Action).
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 2
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(Action).
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
Action = ['Arbitrary_Action_2'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(Action).
false.
```

Discussion of result:
As expected, a valid plan containing sequential execution is found and terminated.

Test A1.2
Test successful: yes.

Execution of test:

```
?- start_new_plan('test_A1_2', Action).
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1),Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob
    /tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1'), 'Arbitrary_Action_2'('http://roboticssrv.wtb.tue.nl/svn/ros/
    user/rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.
    owl#AMIGO_1')].

?- get_action(Action).
```

```
false.
```

Discussion of result:
As expected, a valid plan containing parallel execution is found and terminated.

Test A1.3
Test successful: yes.

Execution of test:

```
?- start_new_plan('test_A1_3', Action).
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1),Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob
    /tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1'), 'Arbitrary_Action_2'('http://roboticssrv.wtb.tue.nl/svn/ros/
    user/rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.
    owl#AMIGO_1')].

?- get_action(Action).
false.
```

Discussion of result:
As expected, a valid plan containing parallel execution is found and terminated.

Test A1.4
Test successful: yes.

Execution of test:

```
?- start_new_plan('test_A1_4', Action).
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(Action).
false.
```

Additional test to showcase the choice construct gives multiple options:

```
?- trans('test_A1_4', s0, Dp, Sp).
Dp = nil,
Sp = do(['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob
    /tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')], 1, s0) ;
Dp = nil,
Sp = do(['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob
    /tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1')], 1, s0) ;
Dp = nil,
Sp = do(['Arbitrary_Action_2'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob
    /tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')], 1, s0) ;
Dp = nil,
```

```
Sp = do(['Arbitrary_Action_2'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob
    /tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1')], 1, s0) ;
false.
```

Discussion of result:

As expected, a valid plan in which one action is chosen is found and terminated. Furthermore, the additional test shows that multiple options can be found for the choice construct.

Test A1.5

Test successful: yes.

Execution of test:

```
?- start_new_plan('test_A1_5', Action).
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 2
Action = ['Arbitrary_Action_2'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(Action).
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(Action).
false.
```

Additional test to showcase the any-order construct gives multiple options:

```
?- trans('test_A1_5', s0, Dp, Sp).
Dp = sequence(nil, 'Arbitrary_Action_2'(_G24)),
Sp = do(['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob
    /tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')], 1, s0) ;
Dp = sequence(nil, 'Arbitrary_Action_2'(_G24)),
Sp = do(['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob
    /tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1')], 1, s0) ;
Dp = sequence(nil, 'Arbitrary_Action_1'(_G22)),
Sp = do(['Arbitrary_Action_2'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob
    /tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')], 1, s0) ;
Dp = sequence(nil, 'Arbitrary_Action_1'(_G22)),
Sp = do(['Arbitrary_Action_2'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob
    /tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1')], 1, s0) ;
false.
```

Discussion of result:

As expected, a valid plan containing sequential execution is found and terminated. Furthermore, the additional test shows that multiple options can be found for the any-order construct.

Test A1.6.1
Test successful: yes.

Execution of test:

```
?- start_new_plan('test_A1_6_1', Action).
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(Action).
false.
```

Discussion of result:
As expected, a valid plan in which one action is chosen is found and terminated. The first program of the if-then-else construct is executed, because the condition of the if-statement is true.

Test A1.6.2
Test successful: yes.

Execution of test:

```
?- start_new_plan('test_A1_6_2', Action).
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
Action = ['Arbitrary_Action_2'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(Action).
false.
```

Discussion of result:
As expected, a valid plan in which one action is chosen is found and terminated. The second program of the if-then-else construct is executed, because the condition of the if-statement is false.

Test A1.7
Test successful: yes.

Execution of test:

```
?- start_new_plan('test_A1_7', Action).
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(Action).
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].
```

```
?- finishAction('Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/
    user/rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.
    owl#AMIGO_1')).
true.

?- get_action(Action).
false.
```

Discussion of result:
As expected, a valid plan is found and terminated only after the planner received the feedback that the blocking action of the while-loop is finished.

Test A1.8
Test successful: yes.

Execution of test:

```
?- start_new_plan('test_A1_8', Action).
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(Action).
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- finishAction('Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/
    user/rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.
    owl#AMIGO_1')).
true.

?- get_action(Action).
false.
```

Discussion of result:
As expected, a valid plan is found and terminated only after the planner received the feedback that the blocking action of the repeat-until-loop is finished.

Control Construct Conversion Predicate

This section displays the results of test A2.

Test A2
Test successful: yes.

Execution of test:

```
?- start_new_plan('test_A2', Action).
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 2
```

```
do [Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 3
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(Action).
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
do [Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 2
Action = ['Arbitrary_Action_2'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(Action).
do [Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
Action = ['Arbitrary_Action_3'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(Action).
false.
```

The following shows that the controlConstructConverter/2-predicate functions correctly:

```
?- ArbitraryConstruct = sequence('Arbitrary_Action_1'(_)),
    controlConstructConverter(ArbitraryConstruct, V), arg(1, V, List), length(
    List, Length), functor(ArbitraryConstruct, _, Length).
ArbitraryConstruct = sequence('Arbitrary_Action_1'(_G1)),
V = sequence(['Arbitrary_Action_1'(_G1)]),
List = ['Arbitrary_Action_1'(_G1)],
Length = 1 ;
false.

?- ArbitraryConstruct = sequence('Arbitrary_Action_1'(_), 'Arbitrary_Action_2
    '(_)), controlConstructConverter(ArbitraryConstruct, V), arg(1, V, List),
    length(List, Length), functor(ArbitraryConstruct, _, Length).
false.

?- ArbitraryConstruct = sequence('Arbitrary_Action_1'(_), 'Arbitrary_Action_2
    '(_), 'Arbitrary_Action_3'(_)), controlConstructConverter(
    ArbitraryConstruct, V), arg(1, V, List), length(List, Length), functor(
    ArbitraryConstruct, _, Length).
ArbitraryConstruct = sequence('Arbitrary_Action_1'(_G1), 'Arbitrary_Action_2
    '(_G2), 'Arbitrary_Action_3'(_G3)),
V = sequence(['Arbitrary_Action_1'(_G1), 'Arbitrary_Action_2'(_G2), '
    Arbitrary_Action_3'(_G3)]),
List = ['Arbitrary_Action_1'(_G1), 'Arbitrary_Action_2'(_G2), '
    Arbitrary_Action_3'(_G3)],
Length = 3 ;
false.
```

Discussion of result:
As expected, a valid plan containing a sequence with more than two programs is found and terminated.

The additional listing shows the conversion process in more detail. It shows that a sequence containing only one task is converted into a sequence containing a list with only one task. It also shows that a sequence containing two tasks is not converted, and that a sequence containing three tasks is converted into a sequence containing a list with all three tasks.

C.2.2 Temporal Planning

This section displays the results of tests B1 to B3.

Actions with Durations

This section displays the results of tests B1.1 to B1.2.

Test B1.1
Test successful: yes.

This test is essentially the same as test A1.1. Therefore, no test output is given here. Even so, it can be seen from the output of test A1.1 that the start time of Arbitrary_Action_2 indeed succeeds the start time of its predecessor Arbitrary_Action_1.

Test B1.2.1
Test successful: yes.

Execution of test:

```
?- start_new_plan('test_B1_2_1', Action).
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1),Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob
    /tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1),Arbitrary_Action_4(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob
    /tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 2
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1'), 'Arbitrary_Action_3'('http://roboticssrv.wtb.tue.nl/svn/ros/
    user/rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.
    owl#AMIGO_1')].

?- get_action(X).
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1),Arbitrary_Action_4(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob
    /tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
X = ['Arbitrary_Action_2'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1'), 'Arbitrary_Action_4'('http://roboticssrv.wtb.tue.nl/svn/ros/
    user/rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.
    owl#AMIGO_1')].

?- get_action(X).
false.
```

Discussion of result:
As expected, the planner produces a valid plan in which the actions are in the same order as given in the sequence constructs, the sequences are executed in parallel, and the start time of every subsequent action

succeeds the start time of its predecessor. The same robot is indeed used for every action.

Test B1.2.2
Test successful: yes.

Execution of test:

```
?- start_new_plan('test_B1_2_2', Action).
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1),Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob
    /tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1),Arbitrary_Action_4(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob
    /tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 2
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1'), 'Arbitrary_Action_3'('http://roboticssrv.wtb.tue.nl/svn/ros/
    user/rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.
    owl#AMIGO_1')].

?- get_action(X).
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1),Arbitrary_Action_4(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob
    /tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
X = ['Arbitrary_Action_2'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1'), 'Arbitrary_Action_4'('http://roboticssrv.wtb.tue.nl/svn/ros/
    user/rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.
    owl#AMIGO_1')].

?- get_action(X).
false.
```

Discussion of result:
As expected, the planner produces a valid plan in which the actions are in the same order as given in the sequence constructs, the sequences are executed in parallel, and the start time of every subsequent action succeeds the start time of its predecessor. The same robot is indeed used for every action.

Test B1.2.3
Test successful: yes.

Execution of test:

```
?- start_new_plan('test_B1_2_3', Action).
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1),Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob
    /tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)] at 1
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1),Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob
    /tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)] at 2
```

```
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1'), 'Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/
    user/rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.
    owl#PICO_1')].

?- get_action(X).
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1),Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob
    /tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)] at 1
X = ['Arbitrary_Action_2'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1'), 'Arbitrary_Action_2'('http://roboticssrv.wtb.tue.nl/svn/ros/
    user/rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.
    owl#PICO_1')].

?- get_action(X).
false.
```

Discussion of result:

As expected, the planner produces a valid plan in which the actions are in the same order as given in the sequence constructs, the sequences are executed in parallel, and the start time of every subsequent action succeeds the start time of its predecessor. The same robot is indeed used for every action.

Test B1.2.4

Test successful: yes.

Execution of test:

```
?- start_new_plan('test_B1_2_4', Action).
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1),Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob
    /tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 2
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 3
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(X).
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1),Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob
    /tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 2
X = ['Arbitrary_Action_2'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1'), 'Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/
    user/rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.
```

```
    owl#AMIGO_1') ].

?- get_action(X).
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
X = ['Arbitrary_Action_2'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1') ].

?- get_action(X).
false.
```

Discussion of result:

As expected, the planner produces a valid plan in which the actions are in the same order as given in the sequence constructs, the sequences are executed in parallel, and the start time of every subsequent action succeeds the start time of its predecessor. The same robot is indeed used for every action.

Test B1.2.5

Test successful: yes.

Execution of test:

```
?- start_new_plan('test_B1_2_5', Action).
do [Arbitrary_Action_5(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    ),Arbitrary_Action_7(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack
    /tue_multirobot_scheduling/owl/pico.owl#Actuator_Base)] at 1
do [Arbitrary_Action_6(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    ),Arbitrary_Action_8(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack
    /tue_multirobot_scheduling/owl/pico.owl#Actuator_Base)] at 2
Action = ['Arbitrary_Action_5'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    '), 'Arbitrary_Action_7'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/pico.owl#Actuator_Base
    ') ].

?- get_action(X).
do [Arbitrary_Action_6(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    ),Arbitrary_Action_8(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack
    /tue_multirobot_scheduling/owl/pico.owl#Actuator_Base)] at 1
```

```
X = ['Arbitrary_Action_6'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    '), 'Arbitrary_Action_8'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/pico.owl#Actuator_Base
    ')].

?- get_action(X).
false.
```

Discussion of result:

As expected, the planner produces a valid plan in which the actions are in the same order as given in the sequence constructs, the sequences are executed in parallel, and the start time of every subsequent action succeeds the start time of its predecessor. Also, the planner uses two different robots.

Test B1.2.6

Test successful: yes.

Execution of test:

```
?- start_new_plan('test_B1_2_6', Action).
do [Arbitrary_Action_7(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    )] at 1
do [Arbitrary_Action_8(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    )] at 2
do [Arbitrary_Action_5(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    )] at 3
do [Arbitrary_Action_6(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    )] at 4
Action = ['Arbitrary_Action_7'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    ')].

?- get_action(X).
do [Arbitrary_Action_8(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    )] at 1
do [Arbitrary_Action_5(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
```

```
    )] at 2
do [Arbitrary_Action_6(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    )] at 3
X = ['Arbitrary_Action_8'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    ')].

?- get_action(X).
do [Arbitrary_Action_5(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    )] at 1
do [Arbitrary_Action_6(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    )] at 2
X = ['Arbitrary_Action_5'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    ')].

?- get_action(X).
do [Arbitrary_Action_6(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    )] at 1
X = ['Arbitrary_Action_6'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    ')].

?- get_action(X).
false.
```

Discussion of result:

As expected, the planner produces a valid plan in which the actions are in the same order as given in the sequence constructs, the sequences are executed in sequence, and the start time of every subsequent action succeeds the start time of its predecessor. The same robot is indeed used for every action.

Test B1.2.7

Test successful: yes.

Execution of test:

```
?- start_new_plan('test_B1_2_7', Action).
do [Arbitrary_Action_5(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    ),Arbitrary_Action_5(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
```

```
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack
    /tue_multirobot_scheduling/owl/pico.owl#Actuator_Base)] at 1
do [Arbitrary_Action_6(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    ),Arbitrary_Action_6(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack
    /tue_multirobot_scheduling/owl/pico.owl#Actuator_Base)] at 2
Action = ['Arbitrary_Action_5'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    '), 'Arbitrary_Action_5'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/pico.owl#Actuator_Base
    ')].

?- get_action(X).
do [Arbitrary_Action_6(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    ),Arbitrary_Action_6(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack
    /tue_multirobot_scheduling/owl/pico.owl#Actuator_Base)] at 1
X = ['Arbitrary_Action_6'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    '), 'Arbitrary_Action_6'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/pico.owl#Actuator_Base
    ')].

?- get_action(X).
false.
```

Discussion of result:

As expected, the planner produces a valid plan in which the actions are in the same order as given in the sequence constructs, the sequences are executed in parallel, and the start time of every subsequent action succeeds the start time of its predecessor. Also, the planner uses two different robots.

Test B1.2.8

Test successful: yes.

Execution of test:

```
?- start_new_plan('test_B1_2_8', Action).
do [Arbitrary_Action_5(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    )] at 1
```

```
do [Arbitrary_Action_6(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    )] at 2
do [Arbitrary_Action_5(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    )] at 3
do [Arbitrary_Action_6(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    )] at 4
Action = ['Arbitrary_Action_5'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    ')].

?- get_action(X).
do [Arbitrary_Action_6(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    )] at 1
do [Arbitrary_Action_5(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    )] at 2
do [Arbitrary_Action_6(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    )] at 3
X = ['Arbitrary_Action_6'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    ')].

?- get_action(X).
do [Arbitrary_Action_5(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    )] at 1
do [Arbitrary_Action_6(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    )] at 2
X = ['Arbitrary_Action_5'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
```

```
    ')].

?- get_action(X).
do [Arbitrary_Action_6(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    )] at 1
X = ['Arbitrary_Action_6'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Base
    ')].

?- get_action(X).
false.
```

Discussion of result:

As expected, the planner produces a valid plan in which the actions are in the same order as given in the sequence constructs, the sequences are executed in sequence, and the start time of every subsequent action succeeds the start time of its predecessor. The same robot is indeed used for every action.

Planner Robustness

This section displays the results of tests B2.1 to B2.4.

Test B2.1

Test successful: yes.

Execution of test:

```
Starting executive

Start getting plan at time:
1392972981.89

Action(s) received from planner:
['Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)']

Starting action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392972981.9

Finished action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392972981.9
Duration: 0.000142812728882

Action(s) received from planner:
```

```
['Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)']

Starting action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392972981.9

Finished action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392972981.92
Duration: 0.0161719322205

Action(s) received from planner:
['Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)']

Starting action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392972981.94

Finished action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392972982.95
Duration: 1.0163769722

Finished executive at time:
1392972985.99

Total executing time:
4.09438896179 seconds

Total action steps needed: 3
```

Discussion of result:
As expected, the planner continues to function properly.

Test B2.2
Test successful: yes.

Execution of test:

```
Starting executive
```

```
Start getting plan at time:
1392973614.06

Action(s) received from planner:
['Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)']

Starting action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392973614.07

Finished action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392973644.09
Duration: 30.0237460136

Action(s) received from planner:
['Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)']

Starting action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392973644.19

Finished action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392973665.22
Duration: 21.0219209194

Action(s) received from planner:
['Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)']

Starting action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
```

```
at time:
1392973665.56

Finished action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392973692.58
Duration: 27.0249619484

Finished executive at time:
1392973695.64

Total executing time:
81.5723650455 seconds

Total action steps needed: 3
```

Discussion of result:
As expected, the planner continues to function properly.

Test B2.3
Test successful: yes.

Execution of test:

```
Starting executive

Start getting plan at time:
1392973878.73

Action(s) received from planner:
['Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)']

Starting action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392973878.73

Finished action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392973880.74
Duration: 2.0026550293

Action(s) received from planner:
['Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)']
```

```
Starting action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392973880.77

Finished action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392973881.77
Duration: 1.00125622749

Action(s) received from planner:
['Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)']

Starting action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392973881.92

Finished action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392973910.96
Duration: 29.0490620136

Finished executive at time:
1392973913.99

Total executing time:
35.2559821606 seconds

Total action steps needed: 3
```

Discussion of result:
As expected, the planner continues to function properly.

Test B2.4
Test successful: yes.

Execution of test:

```
Starting executive

Start getting plan at time:
1392973968.75
```

```
Action(s) received from planner:
['Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)', 'Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)']

Starting action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392973968.9

Finished action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392973968.9
Duration: 0.000159025192261

Starting action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392973968.9

Finished action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392973969.91
Duration: 1.00883102417

Action(s) received from planner:
['Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)', 'Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)']

Starting action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392973970.06

Starting action:
Arbitrary_Action_2
```

```
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392973970.06

Finished action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392973972.07
Duration: 2.00863003731

Finished action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392973972.09
Duration: 2.02477312088

Action(s) received from planner:
['Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)', 'Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)']

Starting action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392973972.11

Starting action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392973972.12

Finished action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392973972.11
Duration: 0.000110149383545

Finished action:
Arbitrary_Action_3
```

```
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392973973.13
Duration: 1.0126979351

Finished executive at time:
1392973976.13

Total executing time:
7.38043403625 seconds

Total action steps needed: 3
```

Discussion of result:
As expected, the planner continues to function properly.

Test B2.5
Test successful: yes.

Execution of test:

```
Starting executive

Start getting plan at time:
1392974044.96

Action(s) received from planner:
['Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)', 'Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)']

Starting action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974045.07

Starting action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974045.07

Finished action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974068.1
Duration: 23.0308279991
```

```
Finished action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974072.08
Duration: 27.0034720898

Action(s) received from planner:
['Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)', 'Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)']

Starting action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974072.39

Starting action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974072.39

Finished action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974092.43
Duration: 20.0350701809

Finished action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974095.41
Duration: 23.0130038261

Action(s) received from planner:
['Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)', 'Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)']

Starting action:
```

```
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974095.57

Starting action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974095.57

Finished action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974116.59
Duration: 21.0210790634

Finished action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974119.6
Duration: 24.0317890644

Finished executive at time:
1392974122.63

Total executing time:
77.6722021103 seconds

Total action steps needed: 3
```

Discussion of result:
As expected, the planner continues to function properly.

Test B2.6
Test successful: yes.

Execution of test:

```
Run 1:
Starting executive

Start getting plan at time:
1392974171.43

Action(s) received from planner:
['Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)', 'Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
```

```
    PICO_1)']

Starting action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974171.58

Finished action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974171.58
Duration: 0.000152826309204

Starting action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974171.58

Finished action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974173.59
Duration: 2.01473903656

Action(s) received from planner:
['Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)', 'Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)']

Starting action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974173.74

Starting action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974173.74
```

```
Finished action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974174.8
Duration: 1.0597858429

Finished action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974199.78
Duration: 26.0418508053

Action(s) received from planner:
['Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)', 'Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)']

Starting action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974199.81

Starting action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974199.81

Finished action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974200.84
Duration: 1.02532911301

Finished action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974200.84
```

```
Duration: 1.02519392967

Finished executive at time:
1392974203.84

Total executing time:
32.4063608646 seconds

Total action steps needed: 3


Run 2:
Starting executive

Start getting plan at time:
1392974334.79

Action(s) received from planner:
['Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)', 'Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)']

Starting action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974334.93

Starting action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974334.93

Finished action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974336.98
Duration: 2.05032610893

Finished action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974355.96
Duration: 21.0232350826
```

```
Action(s) received from planner:
['Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)', 'Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)']

Starting action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974356.15

Starting action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974356.15

Finished action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974357.15
Duration: 1.00124812126

Finished action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974379.17
Duration: 23.023100853

Action(s) received from planner:
['Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)', 'Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)']

Starting action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974379.2

Starting action:
Arbitrary_Action_3
```

```
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974379.2

Finished action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974381.2
Duration: 2.00533795357

Finished action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974402.22
Duration: 23.0195600986

Finished executive at time:
1392974405.22

Total executing time:
70.430505991 seconds

Total action steps needed: 3


Run 3:
Starting executive

Start getting plan at time:
1392974448.93

Action(s) received from planner:
['Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)', 'Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)']

Starting action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974449.07

Starting action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
```

```
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974449.07

Finished action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974470.1
Duration: 21.0234839916

Finished action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974479.09
Duration: 30.014316082

Action(s) received from planner:
['Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)', 'Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)']

Starting action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974479.21

Starting action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974479.21

Finished action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974480.21
Duration: 1.00130701065

Finished action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
```

```
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974504.24
Duration: 25.0331420898

Action(s) received from planner:
['Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)', 'Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)']

Starting action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974504.26

Starting action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974504.26

Finished action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974505.26
Duration: 1.00273299217

Finished action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#PICO_1']
at time:
1392974527.3
Duration: 23.0356099606

Finished executive at time:
1392974530.37

Total executing time:
81.4365859032 seconds

Total action steps needed: 3
```

Discussion of result:

As expected, the planner continues to function properly. This test is run multiple times to strengthen the proof that the planner continues to function properly.

The Cheapest Plan

This section displays the results of test B3.

Test B3
Test successful: yes.

Execution of test:

```
Run 1:
?- start_new_plan('test_B3', Action).
Times: [3,4,5,5,7,10,3,4,5,9,10,2,2,4,5,8,10,5,6,6,8,8]
do [Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
do [Arbitrary_Action_4(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 2
Action = ['Arbitrary_Action_3'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(X).
Times: [1,1]
do [Arbitrary_Action_4(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
X = ['Arbitrary_Action_4'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(X).
false.

Run 2:
?- start_new_plan('test_B3', Action).
Times: [5,6,9,9,10,2,4,5,7,8,10,2,4,6,8,8,2,3,4,5,6,7]
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)] at 1
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)] at 2
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1')].

?- get_action(X).
Times: [1,1]
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
X = ['Arbitrary_Action_2'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(X).
false.

Run 3:
```

```
?- start_new_plan('test_B3', Action).
Times: [3,4,5,6,6,8,4,5,6,7,10,2,3,3,6,8,9,4,8,8,9,10]
do [Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
do [Arbitrary_Action_4(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 2
Action = ['Arbitrary_Action_3'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(X).
Times: [1,1]
do [Arbitrary_Action_4(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
X = ['Arbitrary_Action_4'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(X).
false.
```

Run 4:

```
?- start_new_plan('test_B3', Action).
Times: [5,8,9,10,3,5,7,9,3,3,4,7,10,10,2,5,6,8,10]
do [Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)] at 1
do [Arbitrary_Action_4(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)] at 2
Action = ['Arbitrary_Action_3'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1')].

?- get_action(X).
Times: [1,1]
do [Arbitrary_Action_4(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
X = ['Arbitrary_Action_4'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(X).
false.
```

Run 5:

```
?- start_new_plan('test_B3', Action).
Times: [3,4,5,7,9,10,5,6,7,7,8,10,4,7,7,9,10,4,7,9,10,10]
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)] at 1
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 3
```

```
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1')].

?- get_action(X).
Times: [1,1]
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
X = ['Arbitrary_Action_2'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(X).
false.

Run 6:
?- start_new_plan('test_B3', Action).
Times: [5,5,8,9,10,4,4,6,8,9,9,2,3,4,6,9,10,3,4,10]
do [Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)] at 1
do [Arbitrary_Action_4(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 2
Action = ['Arbitrary_Action_3'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1')].

?- get_action(X).
Times: [1,1]
do [Arbitrary_Action_4(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
X = ['Arbitrary_Action_4'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(X).
false.
```

Discussion of result:

As expected, the planner always returns the cheapest plan from all valid plans found. The numbers after "Times:" display the total durations of all valid plans that are found.

C.2.3 Multi-Robot Control

This section displays the results of tests C1 to C3.

Too Many Robots

This section displays the results of tests C1.1 to C1.2.

Test C1.1

Test successful: yes.

Execution of test:

```
?- start_new_plan('test_C', Action).
```

```
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/ros_grounding.owl#Loc_1
    )] at 1
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/ros_grounding.owl#Loc_1
    ')].

?- get_action(X).
false.
```

Discussion of result:
As expected, the resulting plan is a plan for only one robot.

Test C1.2
Test successful: yes.

Execution of test:

```
?- start_new_plan('test_C', Action).
false.
```

Discussion of result:
As expected, the planner fails to find a valid plan.

Not Enough Robots

This section displays the results of tests C2.1 to C2.2.

Test C2.1
Test successful: yes.

Execution of test:

```
?- start_new_plan('test_C', Action).
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/ros_grounding.owl#Loc_1
    )] at 1
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/ros_grounding.owl#Loc_1
    ')].

?- get_action(X).
false.
```

Discussion of result:
As expected, the planner selected the only available robot to go to only one 'order location'.

Test C2.2
Test successful: yes.

Execution of test:

```
?- start_new_plan('test_C', Action).
false.
```

Discussion of result:
As expected, the planner fails to find a valid plan.

Robot Teamwork

This section displays the results of test C3.

Test C3
Test successful: yes.

Execution of test:

```
?- start_new_plan('test_C', Action).
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/ros_grounding.owl#Loc_1
    ),Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack
    /tue_multirobot_scheduling/owl/ros_grounding.owl#Loc_2)] at 1
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/ros_grounding.owl#Loc_1
    '), 'Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/ros_grounding.owl#Loc_2
    ')].

?- get_action(X).
false.
```

Discussion of result:
As expected, the planner selected all available robots and plans for concurrent execution. A valid plan is produced.

C.2.4 Robot Capabilities

This section displays the results of tests D1 to D2.

Making Choices based on Capabilities

This section displays the results of tests D1.1 to D1.3.

Test D1.1
Test successful: yes.

Execution of test:

```
?- start_new_plan('test_D1_1', Action).
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Arm)
    ] at 1
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
```

```
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Arm
    ')].

?- get_action(X).
false.
```

Discussion of result:
As expected, the planner selects the robot with arms to do the task where arms are needed. A valid plan is produced.

Test D1.2.1
Test successful: yes.

Execution of test:

```
?- start_new_plan('test_D1_2_1', Action).
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Arm)
    ,Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1)] at 1
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Arm
    '), 'Arbitrary_Action_3'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1')].

?- get_action(X).
false.
```

Discussion of result:
As expected, the planner selects the robot with arms to do the task where arms are needed, and the other robot for the other task. A valid plan is produced.

Test D1.2.2
Test successful: yes.

Execution of test:

```
?- start_new_plan('test_D1_2_2', Action).
do [Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1),Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Arm)
    ] at 1
Action = ['Arbitrary_Action_3'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    PICO_1'), 'Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user
    /rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Arm
    ')].

?- get_action(X).
false.
```

Discussion of result:

As expected, the planner selects the robot with arms to do the task where arms are needed, and the other robot for the other task. A valid plan is produced.

Test D1.3

Test successful: yes.

Execution of test:

```
?- start_new_plan('test_D1_3', Action).
false.
```

Discussion of result:

As expected, the planner fails to find a valid plan.

Conflicts

This section displays the results of test D2.

Test D2

Test successful: yes.

Execution of test:

```
?- start_new_plan('test_D2', Action).
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Arm)
    ] at 1
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Arm)
    ] at 2
Action = ['Arbitrary_Action_2'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Arm
    ')].

?- get_action(X).
do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1,http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Arm)
    ] at 1
X = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1', 'http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/amigo.owl#Actuator_Arm
    ')].

?- get_action(X).
false.
```

Discussion of result:

As expected, the planner produces a valid plan in which the two actions are executed in sequence instead of concurrently, even though the procedure contains a split-join construct. This is because the actions conflict with each other and cannot be executed in parallel.

C.2.5 Online Planning

This section displays the results of tests E1 to E2.

Feedback Loop

This section displays the results of test E1.

Test E1
Test successful: yes.

Execution of test:

```
Starting executive

Start getting plan at time:
1392974974.09

Action(s) received from planner:
['Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)']

Starting action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974974.1

Finished action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974975.1
Duration: 1.00122499466

Action(s) received from planner:
['Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)']

Starting action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974975.18

Finished action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392974976.18
```

```
Duration: 1.00275707245

Finished executive at time:
1392974979.27

Total executing time:
5.18195605278 seconds

Total action steps needed: 2
```

Discussion of result:
As expected, the following communication features all function as required:

- The executive queries for a new plan.
- The planner initiates a new plan and returns the first action to the executive.
- The executive asks for the next action.
- The planner returns the next action of a plan.

Breaking out of While-Loops

This section displays the results of test E2.

Test E2
Test successful: yes.

Execution of test (simulated only with Prolog):

```
?- start_new_plan('test_E2', Action).

do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 2
do [Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 3
Action = ['Arbitrary_Action_1'('http://roboticssrv.wtb.tue.nl/svn/ros/user/
    rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(X).

do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
do [Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 2
X = ['Arbitrary_Action_2'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(X).

do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
```

```
do [Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 2
X = ['Arbitrary_Action_2'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- finishAction('Arbitrary_Action_2'('http://roboticssrv.wtb.tue.nl/svn/ros/
    user/rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.
    owl#AMIGO_1')).
true.

?- get_action(X).

do [Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
X = ['Arbitrary_Action_3'('http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1')].

?- get_action(X).
false.
```

Execution of test (simulated with Python test application and PySWIP):

```
Application output on terminal:
[INFO] [WallTime: 1392976008.832159] Starting executive

do [Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 2
do [Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 3

Action list = ['Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/
    user/rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.
    owl#AMIGO_1)']
elapsed time: 1.00114512444 of action Arbitrary_Action_1
action result = True , action_name = Arbitrary_Action_1(http://roboticssrv.
    wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multirobot_scheduling
    /owl/action_domain.owl#AMIGO_1)

do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
do [Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 2

New Action list = ['Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros
    /user/rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain
    .owl#AMIGO_1)']

elapsed time: 1.0140709877 of action Arbitrary_Action_2
action result = False , action_name = Arbitrary_Action_2(http://roboticssrv
```

```
    .wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1)

do [Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1
do [Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 2

New Action list = ['Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros
    /user/rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain
    .owl#AMIGO_1)']

elapsed time: 1.05684804916 of action Arbitrary_Action_2
action result = True , action_name = Arbitrary_Action_2(http://roboticssrv.
    wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multirobot_scheduling
    /owl/action_domain.owl#AMIGO_1)

do [Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)] at 1

New Action list = ['Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros
    /user/rob/tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain
    .owl#AMIGO_1)']

elapsed time: 1.01441693306 of action Arbitrary_Action_3
action result = True , action_name = Arbitrary_Action_3(http://roboticssrv.
    wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/tue_multirobot_scheduling
    /owl/action_domain.owl#AMIGO_1)
No new plan is received. Plan is finished!

Total executing time: 7.3527238369 seconds

Total action steps needed: 4


Log file:
Starting executive

Start getting plan at time:
1392976009.06

Action(s) received from planner:
['Arbitrary_Action_1(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)']

Starting action:
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392976009.1

Finished action:
```

```
Arbitrary_Action_1
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392976010.1
Duration: 1.00114512444

Action(s) received from planner:
['Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)']

Starting action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392976010.11

Finished action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392976011.12
Duration: 1.0140709877

Action(s) received from planner:
['Arbitrary_Action_2(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)']

Starting action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392976011.2

Finished action:
Arbitrary_Action_2
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392976012.26
Duration: 1.05684804916

Action(s) received from planner:
['Arbitrary_Action_3(http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/
    tue_multirobot_stack/tue_multirobot_scheduling/owl/action_domain.owl#
    AMIGO_1)']

Starting action:
```

```
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392976012.3

Finished action:
Arbitrary_Action_3
with action inputs:
['http://roboticssrv.wtb.tue.nl/svn/ros/user/rob/tue_multirobot_stack/
    tue_multirobot_scheduling/owl/action_domain.owl#AMIGO_1']
at time:
1392976013.31
Duration: 1.01441693306

Finished executive at time:
1392976016.42

Total executing time:
7.3527238369 seconds

Total action steps needed: 4
```

Discussion of result:
As expected, the progress flow is indeed as follows:

1. The planner will always do the first action.
2. The first time the second action needs to be executed, the (simulated) executive should tell the planner that it failed.
3. The planner tells the executive to execute the second action again upon getting the feedback that the action failed.
4. The second time the second action needs to be executed, the (simulated) executive should tell the planner it succeeded.
5. The planner should return the third action now.