Doctoral Dissertation

Robots that Help Each Other: Self-Configuration of Distributed Robot Systems

Robert Lundh
Technology

Örebro Studies in Technology 33
Örebro 2009

Robots that Help Each Other:
Self-Configuration of Distributed Robot Systems

Örebro Studies in Technology 33

Robert Lundh

Robots that Help Each Other: Self-Configuration of Distributed Robot Systems

Title: Robots that Help Each Other:
Self-Configuration of Distributed Robot Systems

Publisher: Örebro University, 2009
www.publications.oru.se

Editor: Heinz Merten
heinz.merten@oru.se

Printer: intellecta infolog, V Frölunda 04/2009

ISSN 1650-8580
ISBN 978-91-7668-666-9

Abstract

Imagine the following situation. You give your favorite robot, named Pippi, the task to fetch a heavy parcel that just arrived at your front door. While pushing the parcel back to you, she must travel through a door. Unfortunately, the parcel she is pushing is blocking her camera, giving her a hard time to see the door. If she cannot see the door, she cannot safely push the parcel through it.

What would you as a human do in a similar situation? Most probably you would ask someone for help, someone to guide you through the door, as we ask for help when we need to park our car in a tight parking spot. Why not let the robots do the same? Why not let robots help each other? Luckily for Pippi, there is another robot, named Emil, vacuum cleaning the floor in the same room. Since Emil has a video camera and can view both Pippi and the door at the same time, he can estimate Pippi's position relative to the door and use this information to guide Pippi through the door by wireless communication. In that way he can enable Pippi to deliver the parcel to you. The goal of this thesis is to endow robots with the ability to help each other in a similar way.

More specifically, we consider *distributed robot systems* in which: (1) each robot includes modular functionalities for sensing, acting and/or processing; and (2) robots can help each other by offering those functionalities. A *functional configuration* of such a system is any way to allocate and connect functionalities among the robots. An interesting feature of a system of this type is the possibility to use different functional configurations to make the same set of robots perform different tasks, or to perform the same task under different conditions. In the above example, Emil is offering a perceptual functionality to Pippi. In a different situation, Emil could offer his motion functionality to help Pippi push a heavier parcel.

In this thesis, we propose an approach to automatically generate, at run time, a functional configuration of a distributed robot system to perform a given task in a given environment, and to dynamically change this configuration in response to failures. Our approach is based on artificial intelligence planning techniques, and it is provably sound, complete and optimal.

In order to handle tasks that require more than one step (i.e., one configuration) to be accomplished, we also show how methods for automatic configu-

ration can be integrated with methods for task planning to produce a complete *plan* were each step is a configuration. For the scenario above, generating a complete plan before the execution starts enables Pippi to know before hand if she will be able to get the parcel or not. We also propose an approach to merge configurations, which enables concurrent execution of configurations, thus reducing execution time.

We demonstrate the applicability of our approach on a specific type of distributed robot system, called PEIS-Ecology, and show experiments in which configurations and sequences of configurations are automatically generated and executed on real robots. Further, we give an experiment where merged configurations are created and executed on simulated robots.

Keywords: cooperative robotics, distributed robot systems, self-configuration, task planning.

Acknowledgments

In the work of making robots help other robots, I, Robert, got help from some other human beings that I would like to acknowledge.

First of all, I would like to thank my supervisors Prof. Alessandro Saffiotti and Dr. Lars Karlsson at the Center of Applied Autonomous Sensor Systems (AASS), Örebro University, Sweden. Together we have had many interesting and fruitful discussions, most often research related, but the side-tracks have been entertaining. Thank you for all the support and guidance.

The main part of the research described in this thesis have been conducted as part of the Peis-Ecology Project. I would like to thank everyone who have contributed to this project and in that way made my research possible. I would especially like to thank Dr. Mathias Broxvall for his helpfulness, his work on the Peis software and for taking part in many inspiring discussions. I would also like to thank Dr. Amy Loutfi, Dr. Rafael Muñoz-Salinas, Marco Gritti, Jayedur Rashid, and Jan Larsson for their valuable contributions to the Peis-Ecology Project. The Peis-Home apartment was built and is constantly modified by our excellent research engineers Bo-Lennart Silverdahl and Per Sporrong. Thank you for all the help. The Peis-Ecology Project was mainly founded by ETRI (Electronics and Telecommunications Research Institute, Korea) through the project "Embedded Component Technology and Standardization for URC (2004-2008)".

As part of my Ph.D. studies I have been a member of CUGS (the National Graduate School in Computer Science, Sweden) which is also the main funder of this work. Taking courses at CUGS is very special, meeting student and lecturers from many different Universities in Sweden. I really appreciated this experience and I would like to thank all Ph.D. students and lecturers involved in CUGS. I would especially like to thank Kevin LeBlanc who had to put up with me on all the trips back and forth to the different course sites. A big thanks to Anne Moe for her great job in taking care of everything.

Back in the good old days when I started my Ph.D. studies, AASS was an extraordinary workplace. We had ultra long coffee breaks, a KUF list, and all sorts of funny activities. We even had something called the AASS Olympics and I can proudly say that I once won this track and field event. AASS is still a very

special place, but the closer you get to the final date (the dissertation), the lesser time is left to do all those funny things. I would like to thank all the employees at AASS, past and present, that help make AASS a creative and friendly workplace. Extra acknowledgments to Federico Pecora for proof-reading this thesis and to Barbro Alvin for always fixing everything.

I would like to thank my parents for their encouragement and support. Finally, my greatest love and appreciation goes to my wife Thereze and my sons, Noa and Malte. Thank you for always being there for me and for teaching me what life is all about.

Örebro, March 26, 2009
Robert Lundh

Contents

List of Figures

List of Algorithms

Chapter 1
Introduction

Can you help me? Is it not remarkable how we can extend our own capabilities, just by asking this question? For example, by asking each other for help, we are able to address more advanced tasks than if we are alone, we can perform tasks easier, we can access more information, and we are able to obtain physical support. Especially interesting are two aspects of how we help and cooperate. First, we have altruism. Humans are almost unique in the altruism of our cooperation [Fehr and Fischbacher, 2003]. We help each other even when there is no direct benefit to us, and even to people not in our family or set of acquaintances. Examples of such human altruism are to help an old lady you do not know to cross the street or keeping a door open for a stranger. The second aspect is about how we can "lend" capabilities to each other. For example, when parking a car, the driver can get help from a person standing outside with estimating the distance to surrounding objects, and in that way better know when the car is in the right place. In a sense, the driver is "borrowing" distance measuring capabilities from the other person.

Is it possible for robots to do something similar to what humans do, and if so, in what situations is it useful for robots to help each other?

To answer the second question, consider the situation shown in Figure 1.1. This figure shows a mobile robot, named Pippi, who has the task to push a box through a door. In order to perform this task, Pippi needs to know the position and orientation of the door relative to herself at every time during execution. She can do so by using her sensors, e.g., a camera, to detect the edges of the door and measure their distance and bearing. While pushing the box, however, the camera view is obscured by the box. Pippi can still rely on the previously observed position, and update this position while she moves using odometry. Unfortunately, odometry will be especially unreliable during the push operation due to slippage of the wheels. There is, however, a third solution: a second robot, called Emil, could observe the scene from an external point of view in order to compute the relative position between Pippi and the door, and communicate this information to Pippi.

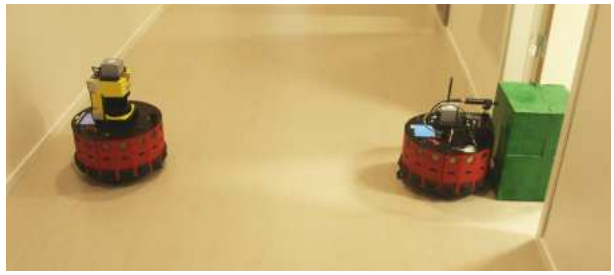


Figure 1.1: Can Emil help Pippi to push the box through the door?

The above scenario illustrates an instance of the general approach that we suggest in this thesis: to let robots help each other by borrowing functionalities from one another. In the above example, Pippi needs a functionality to measure the relative position and orientation of the door in order to perform her task: she has the options to either compute this information using her own sensors, or to borrow this functionality from Emil.

The scenario also illustrates that there are situations in when robots need to cooperate in an altruistic manner. Here Emil helps Pippi to complete her task even though there is no direct benefit to him.

1.1 Problem Definition

To define the objectives of this thesis, it is necessary to elaborate more on which type of systems we address, and the different components of the system.

1.1.1 Distributed Robot Systems

The type of system we are concerned with can be described in a way similar to the usual model with a robot and an environment found in, for example, Artificial Intelligence [Russell and Norvig, 2003, Chapter 2],[Rosenschein, 1987] and Cognitive Science [Newell, 1987, Chapter 2]. Figure 1.2 shows a typical model of a system with a robot that interacts with the environment through sensors and actuators. The robot *observes* the environment through its sensors and uses the actuators to *perform actions* in the environment. Both the robot and the environment have a state. The state of the environment specifies the actual physical properties of, and relations between, different entities in the environment, e.g., the position of a robot, or the color of a cup. The state of the robot specifies its internal physical and computational state, e.g., battery level, and what processes are executing. Such a system can be defined as $\Sigma = \langle X, Y, U, Z \rangle$. X denotes the states of the environment and Y the states of the robot. Z denotes the observations of the environment and U the actions in the environment.

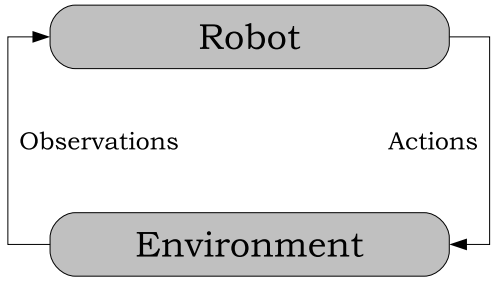


Figure 1.2: A robot interacts with the environment

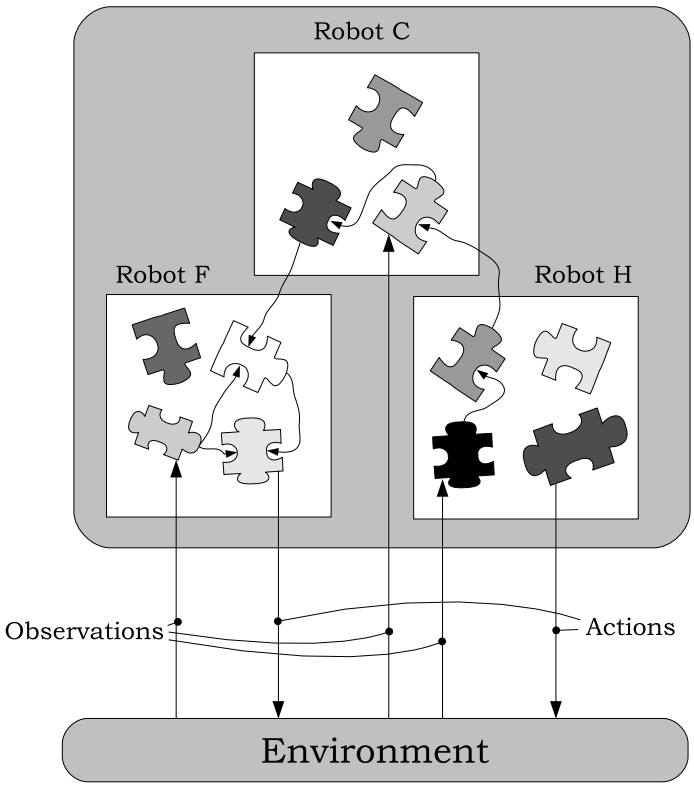


Figure 1.3: A distributed robot system where functionalities can interact with the environment and each other.

Figure 1.3 shows a distributed robot version of the traditional robot environment view which is suitable for the systems we consider in this thesis. There are still the same components as before. However here the distributed nature of the system is made explicit. In this distributed robot system Σ, there are several robots where, each robot is composed of several components (i.e., functionalities, represented as jigsaw puzzle pieces in the figure) that can interact with each other internally and across robot platforms, and some components can also interact directly with the environment. Further, we do not assume that the robots are homogeneous: they may have different sensing, acting, and reasoning capabilities, and some of them may be as simple as a fixed camera monitoring the environment. Functionalities that incorporate actuators can in general only be used for one purpose at a time. To encapsulate this type of restrictions we use resources, e.g., a motion functionality may require the resource that controls the actuator. In our scenario above, there are a number of different functionalities, e.g., Emil has a camera functionality and a functionality which computes the position of Pippi relative to the door from camera images, and Pippi has a functionality responsible for robot movements. The camera functionality *observes* the environment and the movement functionality performs *actions* in the environment. The latter functionality also requires the resource of the actuator that performs the action.

1.1.2 Configurations

The key point in the type of systems described above is that each robot in the system may use functionalities from other robots in order to compensate for the ones that it is lacking, or to improve its own, to perform a given task.

Functionalities can be connected to each other in different ways to address different tasks. We informally define *configuration* any way to allocate and connect the functionalities of a distributed robot system Σ. A formal definition shall be given in Chapter 3. Note that we are interested in functional software configurations, as opposed to the hardware configurations usually considered in the field of reconfigurable robotics (e.g., Fukuda and Nakagawa [1988], Mondada et al. [2004]). Even though the functionalities may interact directly with the environment through sensors and actuators, a configuration of a distributed robot system can be seen as single robot that interacts with the environment as described in the usual robot-environment view above.

Often, the same task can be performed by using different configurations. For example, in our scenario, Pippi can perform her door-crossing task by connecting her own door-crossing functionality to either (1) her own perception functionality, (2) a perception functionality borrowed from Emil, or (3) a perception functionality borrowed from a camera placed over the door. Having the possibility to use different configurations to perform the same task opens the way to improve the flexibility, reliability, and adaptivity of a distributed robot system.

1.1.3 Configuration Plans

In the above example, only one configuration is required to complete the task. However, the configuration in which Emil is guiding Pippi requires that Emil is located at a position where he can observe Pippi and the door. If Emil is not there, he must move to such a position before he can guide Pippi. Hence, the task actually requires several actions to be accomplished, and each action requires its own configuration. We informally call such a sequence of actions, where each action is a configuration, a *configuration plan*. The configurations in Figure 1.4 constitute a configuration plan that enables Pippi to cross the door with help from Emil, when Emil is not located at the desired observing position. The configuration to the left in Figure 1.4 illustrates the configuration which moves Emil to the correct position. This configuration changes the state of the environment such that the configuration in which Pippi crosses the door is applicable (the configuration to the right in Figure 1.4).

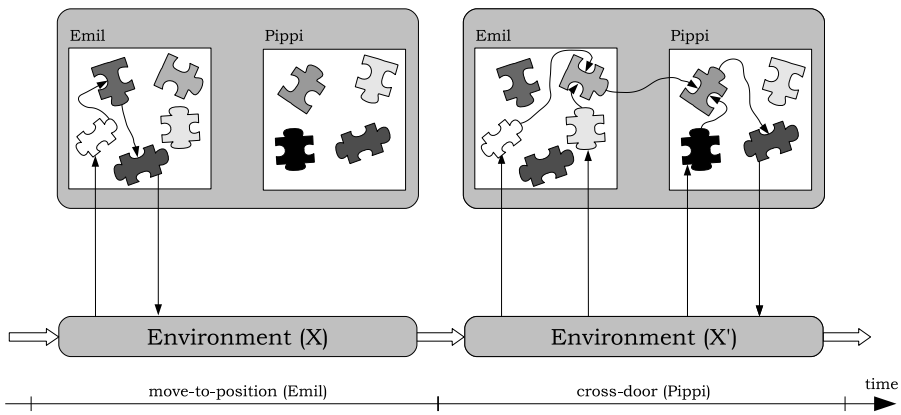


Figure 1.4: A configuration plan with two actions/configurations.

There are also cases in which it is desirable to execute several configurations concurrently. For instance, it can be desirable when several robots want to perform different tasks at the same time, but also when parallelizing a configuration plan in order to reduce execution time. In order to execute several configurations concurrently, it is necessary to validate that the configurations can safely share the available functionalities and resources without conflicts.

1.1.4 The Self-Configuration Problem

In a broad sense, the general problem addressed in this thesis is the study of configurable distributed robot systems of the type discussed above, and in partic-

ular the automatic creation and execution of configurations for these systems. More specific problem definitions will be given in the following chapters.

1.2 Objectives of this Thesis

The long term objective of our work is to enable robots to help each other in an altruistic manner for tasks that require robots to lend functionalities to one another.

To achieve this long term objective we focus on the problems of automatically generating configurations, or sequences of configurations, and how these configurations can be safely executed. In this context, the concrete research objectives of this thesis are:

1. To formally define the concept of functional *configuration* of a distributed robot system.
2. To study how to *automatically generate* a configuration of a distributed robot system for a given task, environment, and set of resources.
3. To study how to automatically generate *sequences of configurations* for tasks that require several steps.
4. To study how *several configurations* can be safely executed in *parallel*.
5. To study when and how to *change* the current configuration, or sequence of configurations, in response to changes in the environment, in the tasks, or in the available resources.

1.3 Methodology

In order to achieve the above objectives we use a methodology in which these objectives are addressed in increasing order while constantly:

- Developing and implementing representations and algorithms,
- Validating properties formally, and
- Testing ideas experimentally with the goal of having an approach that works on a real distributed robot system.

More precisely, we start by defining a concept of configuration which is adequate for the purpose of automatically reasoning about configurations, and showing how to use AI planning techniques to generate a configuration that solves a given task. Then, we describe different techniques to combine the configuration planning with action planning to solve tasks that requires several steps to be solved, and how configurations can be merged for concurrent execution. At each step of our development, we show the formal properties of

the techniques that we introduce, and we show how these techniques can be incorporated in a full distributed robot system. In particular, we describe how to reconfigure, and re-plan, in response to functionality failures. Finally, we describe an experimental system where these ideas are implemented, and show examples of it in which several robots and other devices help each other to perform different tasks.

1.4 Thesis Outline

The remaining parts of this thesis are organized as follows:

Chapter 2 is a survey of work related to the system and the problems that we address in this thesis. From the system point of view, a brief overview of work in multi-robot systems and closely related areas such as multi-agent systems and network robot system is given. With respect to the problem addressed in this thesis, we review selected work on configurations of software systems and work on how to coordinate physical interaction among robots.

Chapter 3 presents the suggested framework to define configurations in distributed robot systems. The main contribution of this chapter is a formal definition of the concept of configuration and its components.

Chapter 4 details our approach to the automatic generation of configurations. The main contribution of this chapter is a planning algorithm that given a goal, a domain, and a state generates an optimal configuration.

Chapter 5 discusses three different ways in which the configuration generation algorithm can be coupled with an action planner to generate sequences of configurations. An approach in which the action planner and configuration planner are loosely coupled is described in more detail since this is considered as the best tradeoff between efficiency and optimality.

Chapter 6 discusses when and why it is desirable to execute several configurations concurrently. The main contribution of this chapter is an approach for merging configurations. If two configurations can be merged into one configuration, then it is also possible to execute the configurations concurrently. We also describe how the merging can be used for more efficient execution of configuration plans.

Chapter 7 illustrates the applicability of the approach. Eight different experiments have been conducted to illustrate the different parts of our approach. Seven of these experiments were conducted on a real multi robot system and one experiment was conducted in a mid-fidelity 3D simulator.

Chapter 8 concludes this thesis with main contributions, limitations of the approach, and directions of future work.

1.5 Publications

Parts of the work reported in this thesis have been presented in a number of journal and conference papers. The publications in the list are available on-line at http://www.aass.oru.se/~rlh.

- R. Lundh, L. Karlsson, and A. Saffiotti. Autonomous functional configuration of a network robot system. *Robotics and Autonomous Systems*, 56(10):819–830, October 2008.
- A. Saffiotti, M. Broxvall, M. Gritti, K. LeBlanc, R. Lundh, J. Rashid, B.S. Seo, and Y.J. Cho. The PEIS-ecology project: vision and results. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2329–2335, Nice, France, September 2008.
- R. Lundh, L. Karlsson, and A. Saffiotti. Automatic configuration of multi-robot systems: Planning for multiple steps. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI)*, Patras, Greece, July 2008.
- R. Lundh, L. Karlsson, and A. Saffiotti. Dynamic self-configuration of an ecology of robots. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3403–3409, San Diego, CA, November 2007.
- R. Lundh, L. Karlsson, and A. Saffiotti. Plan-based configuration of an ecology of robots. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 64–70, Rome, Italy, April 2007.
- R. Lundh, L. Karlsson, and A. Saffiotti. Plan-based configuration of a group of robots. In *Proceedings of the 17th European Conference of Artificial Intelligence (ECAI)*, pages 683–687, Riva del Garda, Italy, August 2006.
- R. Lundh, L. Karlsson, and A. Saffiotti. Can Emil help Pippi? In *Proceedings of the ICRA-05 Workshop on Cooperative Robotics*, Barcelona, Spain, April 2005.
- R. Lundh, L. Karlsson, and A. Saffiotti. Dynamic configuration of a team of robots. In *Proceedings of the ECAI-04 Workshop on Agents in dynamic and real-time environments*, pages 57–62, Valencia, Spain, August 2004.

Chapter 2
Related Work

The work presented in this thesis is about robots that help each other to perform tasks. In this chapter we discuss research work that is related to the type of system that we consider, and related to the type of problem that we consider.

The *system* we consider is composed of multiple cooperating robotic agents residing in a physical environment. Research areas that are concerned with similar systems are: Multi-Robot Systems, Swarm Robotics, Network Robot Systems, and to some extent Multi-Agent Systems. (Section 2.1)

The *problem* we consider is how to automatically generate a configuration, or a sequence of configurations, for a distributed robot system. The general problem of self-configuration of a distributed system is addressed in several fields, including ambient intelligence, web service composition, distributed middleware, autonomic computing, coalition formation, network robot systems, and multi robot task performance. Work on automatic configuration has also been done in other research areas such as: program supervision, dynamic software architectures, and single robot task performance.

We divide the review of work on related problems into three parts: (1) configuration of software systems that not necessarily deal with physical robots (Section 2.2), (2) configuration of software in robotic systems (Section 2.3), and (3) different types of interaction or coordination in physical robot systems (Section 2.4).

We discuss research related to the specific techniques that we use in other chapters of the thesis (Chapters 3 – 6).

2.1 Distributed Robot Systems

In this section, we look at the target system for our approach, i.e., a system that is composed of multiple robots and/or robotic devices that are physically distributed in the environment.

The field of distributed robot systems and cooperation among robots is a relatively new research area. It was not until the late 1980's that researchers

started to gain interest in issues concerning multiple physical robots. Prior to this, cooperation between agents concerned software agents and research on robots mostly considered single robots. The first topics in focus were reconfigurable robots, swarms, motion planning, and architectures. As the research area has grown, certain aspects have been more investigated than others. Several taxonomies and summaries of the field have been proposed, e.g., [Parker, 2003a, 2008, Dudek et al., 2002]. A key component of a multi-robot system is the ability to coordinate and program the interaction between the robots. A number of different middlewares or languages have been presented that facilitate coordination, e.g., ROCI [Chaimowicz et al., 2003], TDL [Simmons and Apfelbaum, 1998], Dynamic Teams [Jennings and Kirkwood-Watts, 1998], and OpenPRS [Ingrand et al., 1996].

Many problems that are considered in multi-robots systems are often closely related to problems that have been, or are currently, addressed by other research areas. One source of *inspiration* is research on single robot systems. Multi-robot systems can address many single robot problems and in that way use the advantage of having several robots to solve the problems faster and more robust (e.g., multi-robot mapping and exploration [Burgard et al., 2005]). Another common source of inspiration is biology, where cooperation between animals (including humans) has been studied for a long time. Behaviors that have been studied are, for example, flocking [Hayes and Dormiani-Tabatabaei, 2002, Turgut et al., 2008], foraging [Balch, 1999, Shell and Mataric, 2006] and following trails [Vaughan et al., 2000]. Flocking, foraging and other insect inspired behaviors are typically considered in a research area called swarm robotics. In swarm robotics, the robots are usually simple, homogeneous (or at least not highly heterogeneous) and in large quantity. The aim is to achieve a collective emergent behavior from the local interactions among agents and between the agents and the environment. The approach presented in this thesis is mainly concerned with a system of relatively few robots that can be highly heterogeneous in sensing/actuation capabilities and computational complexity.

Multi-agent and distributed artificial intelligence (DAI) are other research areas that consider similar problems (although often not for physical robots). These areas have addressed many problems considering cooperation between agents. Early work in DAI considered distributed problem solving settings with a precedence order among sub-tasks [Durfee et al., 1988]. Later work has included the notion of coalitions between sub-groups of more closely interacting agents [Shehory and Kraus, 1998] [Lau and Zhang, 2003]. The work on coalition formation is particularly interesting. Coalition formation is concerned with the problem how to allocate tasks, that cannot be address by a single agent, to disjoint agent teams (coalitions). The work by Vig and Adams [2005] points out the difficulties of and a potential solution to how well-known coalition formation algorithms can be used for the multi-robot domain.

In the multi-agent systems community, team-work [Pynadath and Tambe, 2003], capability management [Timm and Woelk, 2003] and norms [Boella,

2003] have been used to account for the different forms of interactions between the sub-tasks performed by the agents in a team.

For a more detailed overview of research on multi-agent coordination, see the article by Pynadath and Tambe [2002]. For more general overviews in the area of agents and multi agent systems, see [Lesser, 1999, Nwana and Ndumu, 1999, Jennings et al., 1998, Sycara, 1998]. This thesis is concerned with interactions among physical robots and therefore works considering only software agents is of less interest.

The last area concerned with multiple cooperating robot agents that we mention is Network Robot Systems [Sanfeliu et al., 2008]. In network robot systems, the idea of having one extremely competent isolated robot acting in a passive environment is abandoned, in favor of a network of cooperating robotic devices embedded in the environment [Akimoto and Hagita, 2006, Lee and Hashimoto, 2002, Dressler, 2006, Kim et al., 2004, Rusu et al., 2008, Saffiotti and Broxvall, 2005]. In these systems, the term "robot" is taken in a wide sense, including both mobile robots, fixed sensors or actuators, and automated home appliances. One of the most interesting features of a system of this type is the possibility to use different functional configurations to make the same system perform different tasks, or perform the same task under different conditions. This capability provides a great potential for flexibility and robustness, by taking advantage of the diversities and the redundancies within the system. This potential, however, is not fully exploited today. Most existing network robot systems are configured by hand, or through hand-written scripts that can only account for a very limited set of contingencies [Chaimowicz et al., 2003, Lee et al., 2004, Broxvall et al., 2006]. The goal of this thesis is to start filling this gap. The approach proposed in this thesis can be used to automatically generate, on-line, a functional configuration of a network robot system, given information about the target task, the functionalities available in the system, and the current state of the environment.

2.2 Configuration of Software Systems

In this section we consider work in three particular areas: program supervision, automatic web service composition, and ambient intelligence. These works are related to ours since they all consider software components that are automatically connected to each other to create a solution. However, they do not consider mobile robots and the complexity they entail.

2.2.1 Program Supervision

Program supervision is concerned with the problem of automating the reuse of complex software [Thonnat and Moisan, 1995, 2000, Shekhar et al., 1998]. A typical program supervision system consist of:

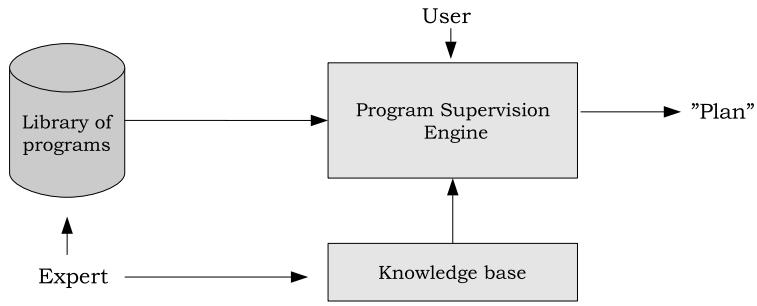


Figure 2.1: A simplified model of a program supervision system

- a database for organizing a library of programs,
- a knowledge base that describes the different programs in the database,
- a user interface that enables users to put up requests on the system and for an expert to modify the system, and
- a supervision engine that selects, plans and executes the programs based on the information in the knowledge base and the information given from the user.

In such a system (Figure 2.1), the user gives a request of the output data in interest, together with input data. The program supervision engine uses this data to generate a plan of programs that can produce the requested output data. In order to do this it uses the information in the knowledge base that describes the characteristics of the programs in the database. An expert of the domain, that the program supervision tool will address, must provide the system with the appropriate information. The most interesting part for us is the supervision engine that generate a plan of programs that produce the requested output. This is similar to the problem of generating configurations. In program supervision, hierarchical planning techniques have been extensively used with good results. Program supervision is mainly used in image processing, but similar concepts have been used in signal processing [Klassner et al., 1998], scientific computing [Rechenmann and Rousseau, 1992] and software engineering [Krueger, 1992]. Programs supervision has been used for applications such as detection of objects in road scenes [Thonnat et al., 1994] and automatic galaxy classification [Vincent et al., 1997].

Program supervision architectures were originally conceived to be centralized, however recent work explores distributed versions based on mobile agents [Khayati et al., 2005].

2.2.2 Automated Web Service Composition

A research area that recently has gained a lot of interest is the semantic web. "The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation" [Berners-Lee et al., 2001]. Here, the problem of how to automatically put together different web services (web service composition) to get new services is particularly interesting. Consider the following problem: you want to go to Rome for a weekend. At first, this does not really look like a problem, hence going to Rome is probably nice. However, to plan a trip with different means of transportation (car, train, flight, etc) and accommodations (Hotel, hostel) is difficult, even thought there are web services that handle each of them separately. The example above is a simplified version of the example given in [Koehler and Srivastava, 2003]. For this example, the web service composition problem would be to combine the different services so that they together form a service for booking trips. Automated web service composition is then used to automatically generate such a composition based on a user request. The article by Rao and Su [2004] presents a framework for automated web service composition that highlights the different parts that are necessary for a service composition system. Figure 2.2 is an illustration of the model. This model is similar to the model for program supervision. Both models have two types of users, one that sends requests to the system and one that provides the information with the appropriate knowledge services. Both models include databases that handle the services or programs and a module that automatically generates the configurations. However, the framework for automatic web service composition also includes modules for translation between specifications, an evaluator and an execution engine. The translator is used to translate between a more easy straight forward language used by users and the specification language used by the system. The evaluator is used to evaluate the solutions by the automatic generator so that the best solution is selected. The execution engine executes the composite service.

Approaches based on planning are widely used to automatically compose web services [Carman et al., 2003, Sirin et al., 2004, Pistore et al., 2005, Hoffmann et al., 2007]. The article by Peer [2005] gives a detailed description of different AI planning techniques used for the problem.

An interesting twist on the Semantic Web was presented at the Robot Semantic Web workshop [Henderson et al., 2007] in San Diego 2007. This workshop discussed the possibility to use similar techniques as in the semantic web for letting robots share knowledge with each other, i.e., to create a robot semantic web. RobotShare [Fan and Henderson, 2007] is a first step in this direction.

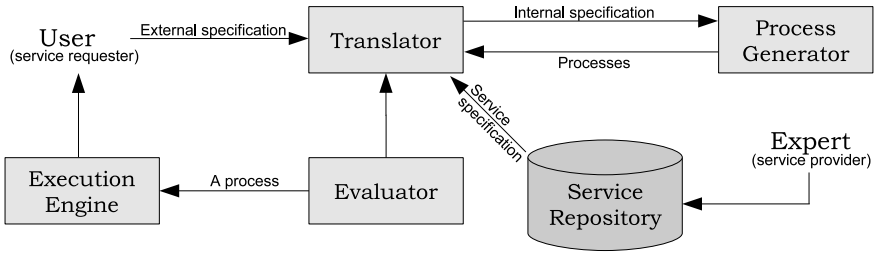


Figure 2.2: A framework for service composition systems [Rao and Su, 2004].

2.2.3 Ambient Intelligence

Ambient Intelligence represents a long-term objective for European research. "The vision for Ambient Intelligence arises from the convergence of three key technologies: Ubiquitous Computing, Ubiquitous Communication, and Intelligent User-Friendly Interfaces. It proposes a laid-back mode of dialogue with an integrated service infrastructure in which one's everyday surroundings become the Interface." [ISTAG, 1999]

In the area of ambient intelligence (not including robots), Heider and Kirste [2005] propose an approach that uses plan-based techniques to control an intelligent environment. The aim is to make interaction between the user and the environment more goal oriented, i.e., the user should not have to learn how to operate all functions on all devices, but rather just give the goal of the interaction (e.g., Find the media source containing media event "Terminator"). Planning is used to develop strategies on how different functions can be executed to reach the goal given by the user. The functions are executed in a sequence like actions of a plan. The work by Amigoni et al. [2005] discusses what type of planner to use for ambient intelligence applications. They propose a planner called distributed hierarchical task network (D-HTN) that generates centralized plans based on the distributed capabilities of the devices in the system. However, the work does not consider coordination among the devices for performing the tasks.

To automatically compose component-based applications is another interesting research problem in ambient intelligence (or pervasive computing). Several pervasive systems are concerned with this problem, e.g., PCOM [Becker et al., 2004], GAIA [Román et al., 2002], and AURA [Garlan et al., 2002]. Handte et al. [2005] (PCOM) presents an approach that uses distributed constraint satisfaction techniques to find possible solutions for how the components can be connected. An example of an application can be a control of a power point presentation (ppt-control). The configuration mechanism is trig-

gered when an application anchor is started (the ppt-control anchor). The dependencies (input and output) of the anchor must then be solved, e.g., a file-system must be connected to the input of the ppt-control anchor. Then providers of input and output must in their turn resolve their own dependencies. When all dependencies are resolved the application is ready to be used.

2.3 Configuration of Software in Robotic Systems

In this section we consider works on automatic configuration in single robot task performance and works on self-configuration in network robot systems.

2.3.1 Single Robot Task Performance

ROBEL [Morisset et al., 2004, Morisset and Ghallab, 2008] is an approach for single robot task performance in which a robot is able to learn how to perform high level tasks. The system generates *skills* (also referred to as modalities), using a hierarchical planner, that consist of a combination of sensory-motor functions (this is done offline). A skill represents one way to perform a desired task. By having several skills for each task, there are several alternatives to solve a task, and thus the robustness of the system improves. A Markov Decision Process is used to determine which skill is appropriate for a particular situation.

Kim et al. [2006] presents SHAGE; a framework for self-managed robot software. SHAGE aims to give a robot the ability to adapt its behavior to the current situation by dynamically changing its software architecture. When an adaption is requested, the current situation is assessed, and the approach tries to find an existing architecture description in a repository that is suitable. If no exact match is found, the most similar architecture is adapted by adding, removing or replacing components to obtain new architectures. The new architectures are tested until a successful architecture is found (trial-and-error). During the test, the system learns the applicability of the different architectures. The next time the same situation occurs, the system can adapt without trial-and-error.

The above works are related to ours since they consider software components and how they can be (re)configured to solve tasks in different ways. The focus in both works is rather on the learning of the right configuration, than the actual configuration generation. In contrast to our work, only the robots own software components can be used to find a solution.

2.3.2 Network Robot Systems

A few works that address problems close to ours can be found in the areas of network robot systems and robots in intelligent environments. Intelligent Spaces [Lee et al., 2004] deal with the problem of how an intelligent environment can actively provide humans and robots with information. In these spaces,

distributed intelligent networked devices (DINDs) act as providers of information. In contrast to the work in this thesis, cooperation in Intelligent Spaces is hard-coded, and it can only handle situations that do not require concurrent operations. Ha et al. [2006] present an approach for automated integration of networked robots into intelligent environments. They use a hierarchical planner to generate sequences of services for a given task. As in traditional web service composition, services are executed in a sequence like actions of a plan. In contrast, functionalities in our approach are executed in parallel and exchange continuous streams of data, which allows us to address tasks that require tight coordination.

Baker et al. [2004] and Gritti et al. [2007][Gritti and Saffiotti, 2008] both consider configuration problems similar to the one addressed in this thesis. In Baker et al., tasks are given to the system in the form of task modules. A suitable configuration for a task is generated by finding the (sensor, effector or computational) components in the system that comply with the constraints of the corresponding task module. Gritti et al. proposes a framework inspired by ideas from the field of semantic web service composition. Configurations are created by instantiating generic templates that express abstract interaction patterns for classes of tasks. In both cases, the search for a suitable configuration is done in a reactive way: the component instances that are found first are selected, and the search horizon is limited to one-step lookahead. This is different from the configuration algorithm proposed in this thesis, which performs a plan-based search that is sound, complete and optimal. A reactive approach might cope better with large and/or highly dynamic environments, but it cannot guarantee that a sound configuration is found, and that it has the lowest cost.

It should be noted that all of the above works only consider cases in which a single configuration is enough to solve the task, i.e., a sequence of configurations is not required. The work in this thesis, by contrast, also considers sequences of configurations.

2.4 Physical Interaction in Distributed Robot Systems

In this section we review works that are concerned with coordination among robots that are able to simultaneously perceive and manipulate the world. The automatic configuration approach presented in this thesis can be used to setup the communication channels for tasks that require tight coordination among robots to be performed. Therefore we review works concerned with coordination of multi-robot systems. We distinguish two branches: loose coordination and tight coordination. *Loose coordination* concerns tasks that can be divided into independent subtasks. The robots may interact extensively to divide and distribute the subtasks among each other. During the execution of the subtasks, the robots interact very little since the subtasks are to be considered indepen-

dent of each other. *Tight coordination* concerns tasks that cannot be easily divided into independent subtasks. The robots may interact extensively to decide who should do what part, and they also have to interact extensively during the actual execution.

2.4.1 Loose Coordination

Since the primitives in loose coordination are single robot tasks, the main problem is how to allocate the tasks, rather than how to perform them. This problem is usually referred to as task allocation, or role assignment when the primitives are roles.

Task allocation is concerned with the problem of how to allocate a number of tasks to a number of agents taking into account that different agents may be differently adequate for different tasks. The simplest case, when a task can only be assigned to one agent and an agent can only be assigned one task, is also known as an Optimal Assignment Problem [Gale, 1960] and has been studied in Game Theory and in Operations Research since 1960.

In the research area of cooperating robots, multi-robot task allocation is one of the more mature topics, and a great amount of approaches have been proposed. Here, the problem can be formulated as follows:

> There are a number of robots, each looking for a task, and a number of tasks, each requiring one robot. Tasks can be of different importance, meaning that tasks get priorities according to how important it is that the task is accomplished. The robots have different capabilities in terms of accomplishing different tasks, i.e., each robot estimates its capability to perform each potential task. The main problem is to maximize the overall expected performance for the assignment of tasks to robots, taking into account the priority of tasks and the different capabilities of the robots. [Gerkey and Matarić, 2003]

The most common type of approaches are based on the Contract Net Protocol (CNP) [Davis and Smith, 1983]. CNP uses auctions in order to assign tasks, i.e., robots make bids on available tasks using their task-specific performance estimation. The robot that is best suited to perform the task will get a contract for the task, since his/her performance estimate will win the bid. The contract allows the robot to execute the task. Some examples of approaches that use some variant of the CNP are M+ [Botelho and Alami, 1999], Murdoch [Gerkey and Matarić, 2002], TraderBots [Dias and Stentz, 2001], GOFER [Caloud et al., 1990].

Another well-known architecture, not based on the CNP, is the behavior-based ALLIANCE architecture [Parker, 1998]. This architecture uses a greedy algorithm to find the best task allocation. The algorithm consist of four, very simple steps: (1) find the pair of task and robot that gives highest utility, (2)

allocate this task to that robot, (3) remove this task-robot pair from the list, and (4) if the list is empty, stop, otherwise go to step (1). ALLIANCE also has an interesting solution to the problem of when to reassign tasks. The architecture uses something called motivational behaviors that are based on two internal models for impatience and acquiescence. Impatience allows robots to take over tasks from other robots in the team. If robot A gets the impression that robot B is not able to accomplish its assigned task, robot A gets **impatient** and can take over the task from robot B. Acquiescence works in a similar way by allowing a robot to give up its current task if the progress is not sufficient. The use of internal models makes the approach very efficient in terms of communication overload, compared to other approaches where robots broadcast their utilities (e.g. [Werger and Matarić, 2000]).

The work by Zlot and Stentz [2005, 2006] is based on TraderBots and focuses on task allocation for complex tasks. They define complex and simple tasks as follows. *Complex Tasks* are tasks that may have many potential solution strategies; finding a plan to a complex task often implies solving an *NP*-hard problem. *Simple Tasks* can be executed by a robot in a straightforward, prescriptive manner. In the definition of complex tasks, complex should be interpreted as its true meaning, i.e., consisting of interconnected parts. The complexity in the task lies in the relation between subtasks. The relations can be in terms of boolean logical associations (e.g. **or** represents alternative solutions) or precedence constraints. This does not necessarily imply tight coordination between robots, since the relation between tasks may only require that one task should be finished before another task starts, and such a constraint can be fulfilled using loose coordination. As mentioned earlier, the approach presented for this problem is market based. In market based approaches for traditional Multi-Robot Task Allocation, robots make bids on tasks that are put up for auction. The robot that is best suited, or rather, believe that it is best suited will win the bid and the right to perform the task. For complex tasks, the auction is different. A complex task cannot be neatly divided between robots, hence to put up a single task for auction does not make any sense. Instead, task trees are put up for auctions. A task tree is a way to represent the different relations between subtasks with the abstract tasks as nodes and primitive tasks as leaves. Relations are represented as different types of edges. By having task tree markets, the approach enables robots to express their valuations for both plans and tasks. This is not possible for other approaches that assume that complex tasks are decomposed into primitive subtasks before the allocation step.

For a more detailed overview and analysis of current research in multi-robot task allocation see the articles by Gerkey and Matarić [2003, 2004] and Dias et al. [2006].

As mentioned above, role assignment considers the same problem as task allocation but with roles as primitives. A role is usually defined as a set of available actions, e.g., a football player with the role `DEFENDER` has the available actions to pass the own goalie, make a sliding tackle, etc, but the action to

catch ball with hands is not available. Examples of different approaches that consider role assignment are [Vail and Veloso, 2003, Stone and Veloso, 1999, Iocchi et al., 2003, Frias-Martinez et al., 2004].

The mentioned approaches to multi-robot task allocation assume that the tasks to allocate are primitives, i.e., tasks that can be executed by a single robot with the right capabilities. They do not consider tasks that require tight coordination between robots.

In the gray area in between loose and tight coordination we find work on how to share common resources and how to resolve resource conflicts (and causal conflicts). In the work on plan-merging [Alami et al., 1995] these issues are considered when a robot needs to execute a plan that uses shared resources. By performing a plan merging operation, the plans with actions that use the shared resources are synchronized by setting timing constraints. In [Botelho and Alami, 2000] this approach, combined with M+ [Botelho and Alami, 1999] mentioned above, is extended with social rules that let robots further synchronize their plans. For instance, social rules can be used to avoid destructive or conflicting actions. If we have two plans that both start with the robots opens a door D1 and end with closing door D1 a social rule can be used such that the merged plan only contains one open and close action. This is still not tight coordination since coordination is used to avoid conflicts in the execution of individual tasks rather than to solve a cooperative task.

2.4.2 Tight Coordination

In contrast to loose coordination, the focus for research on tight coordination tasks has mainly been on domain specific approaches, and not on how to perform or allocate tasks in a more generic sense. The obvious reason for this is that the primitives are not single robot tasks, but rather tasks with several interacting robots involved. Compared to loose coordination where the coordination is only in the initial phase, when the task is decomposed and allocated, tight coordination also requires that the robots coordinate (extensively) throughout the entire duration of the task. This means that a general approach for tight coordination in addition to the question "Who should do which task?" needs to answer the question "How should we jointly perform the task?".

These questions are not easy to answer since the tasks often require real-time coordinated control between robots for execution and the robots must act in a highly coordinated fashion in order to complete them. Thus, most current answers address the tight coordination problem only in a specific domain or task. An example of such a domain is formation control [Balch and Arkin, 1998, Saffiotti et al., 2000], which concerns the problem of keeping and changing formations of robots. For this domain, a mechanism for multiple objectives is of great importance. Here, a robot has at least two objectives that need to be considered: the team objective (to keep the formation) and the individual objective (to avoid obstacles). Object transportation and cooperative

manipulation[Rus et al., 1995, Stroupe et al., 2005] are other domains that typically require tight coordination among robots. A great variety of approaches that consider these problems have been proposed. Apart from the common approaches with robots pushing[1] or carrying boxes, there exist more peculiar approaches with robots transporting boxes using ropes [Donald et al., 2000]. Even though these tasks require tight coordination between robots, the amount of planning for such a task is rather low. For example, while carrying or pushing objects, robots can work in a leader-follower manner and in a rather simple way adopt a tightly coordinated behavior.

In the above approaches the cooperation pattern and the relative roles of each robot within this pattern are predefined and hand-coded. As these domain specific approaches are reaching an acceptable level, attempts to more general approaches for tight coordination are proposed, i.e., approaches that address the problem of how to perform a task and/or who should do which part of the task. That is, the works we will present below tries to automatically determine the cooperation pattern and/or the relative roles of each robot within this pattern. For the question of who should do which part of a task, there are some works in progress to extend traditional multi-robot task allocation to incorporate tight coordination tasks.

On the work on roles, Chaimowicz et al. [2004], presents an approach using dynamic role assignment. In this work, they define a role as "a function that one or more robots perform during the execution of a cooperative task". Such a role determines how the robots act in terms of actions and interaction with each other. The basic concept of the architecture is that at all time there is at least one leader and one follower. The leader broadcasts its own estimated position and velocity to all the followers. The planner on the leader and the trajectory controllers on the followers send set points to the controller. Each robot possesses a cooperation module that is responsible for the role assignment (leader/follower) and for other decisions that directly affect the planner and trajectory controllers. It is important that the best suited robot (in terms of sensor power, manipulation capabilities) leads the group. The leadership is changed either by the leader relinquishing it to another robot or by a leadership request from one of the followers. The experimental validation shows three different experiments with robots that carry a box. The approach is limited to transportation or formation tasks.

The Robotics Institute at Carnegie Mellon University performs research in several different subareas of cooperative robotics. Three of their research lines are particularly interesting to us. Common for all three lines is that they are based on the market based approach called TraderBots [Dias and Stentz, 2001] that uses the Contract Net Protocol mentioned in Section 2.4.1.

[1] Some researches argue that box-pushing tasks are not to be considered as tight coordination tasks since a single robot can push one end at a time. This is true for some cases, but as we stated before, this depends on the involved robot capabilities

Kalra et al. [2005, 2007] present an approach called Hoplits that is focused on market-based task allocation that incorporates tasks that require tight coordination. The Hoplites framework especially addresses tasks that require extensive planning of future interactions between robots in a team. An example of such a task is gallery monitoring. A gallery with several wings must be kept "secure". The level of security is different for different wings which mean that some wings are constantly observed and some wings are watched more periodically. The proposed framework uses a market based approach where each robot is rewarded when a task is completed. The size of the reward is based on how much closer it brings the group to accomplishing the team goal. The market incorporates both passive and active coordination. Passive coordination is used for easier problems in which robots quickly react to each other's actions and influence each other implicitly. Active coordination is used to address harder problems and coordination is achieved explicitly by selling and bidding on complex plans on the market. In the architecture, robots use passive coordination as long it is profitable, i.e., until they discover that active coordination would result in a more profitable solution. The framework does not include a specific planner for generating team plans. The planner to use is chosen depending on the domain in which the robots should operate. Efforts have been made [Stentz et al., 2004] to include the complex task allocation approach [Zlot and Stentz, 2005, 2006] reviewed earlier and the Hoplits framework into the TraderBot architecture.

The work by Jones et al. [2006] considers a problem that they define as the Pickup Team Challenge: to "dynamically form heterogeneous teams of robots given very little a priori information". The idea is that the teams can be created with very short notice (for instance in case of a disaster) and the robots can be highly heterogeneous (different manufacturers, sensors, software). The approach uses a combination of plays [Bowling et al., 2004] and TraderBots [Dias and Stentz, 2001] to form the teams. A play defines a set of robot roles and a sequence of actions for each role. In general, a task can be addressed by different plays and are hand-written in advance. The TraderBot architecture is used to match roles with robot capabilities. The Plays framework takes care of the robot execution and monitoring of tightly coordinated tasks.

In the last work of interest from CMU, Simmons et al. [2000, 2002] presents a three-layered architecture for coordination of heterogeneous robots. The robots coordinate by allowing the three layers to interact with their similar layers located at different robots. For example, at the planning layer, tasks are allocated using TraderBots [Dias and Stentz, 2001]. The work considers a task involving a heterogeneous team of robots — a crane, a robot with a manipulator, and a robot with stereo cameras — solving a construction task where a beam is placed on top of a stanchion. This task requires tight coordination between the robots involved; the robot with the stereo camera tracks the scene and sends information about the position differences to the foreman that controls the movement of the crane and the robot with the manipulator. In the real

experiment, the configuration of the team, including the setup of information flow, is hand-coded. However, in their motivating example, it appears that an objective of this work is to develop techniques that enable robots to assist each other with information or capabilities in tightly-coupled tasks automatically.

None of the above approaches concern the question of how to automatically generate a joint configuration that specifies the information flow between robots required to solve the task.

2.4.3 Combined loose and tight coordination

The work by Tang and Parker [2007] considers both loose and tight coordination tasks. The approach combines market-based task allocation with a robot coalition formation algorithm called ASyMTRe-D [Tang and Parker, 2005c, Parker and Tang, 2006]. The approach assumes a partially ordered plan of tasks that are put up for auctions based on ordering constraints. The robots that receive tasks start to negotiate how they can solve them. Either they try to solve them individually or by forming coalitions with other robots (with constraints on: the number of robots in each coalition, and the number of coalitions to generate). The best coalitions for each robot are placed as bids in the auction and the winner coalitions are then used to perform the tasks. The approach is similar in many respects to the approach that we present in this thesis. The coalition formation algorithm ASyMTRe-D exists in a centralized version (ASyMTRe [Parker et al., 2005, Tang and Parker, 2005a,b]) that works in a similar way as our configuration generation algorithm. ASyMTRe is inspired by Information Invariants [Donald, 1995] and Schema Theory [Arkin, 1987]. As in Schema Theory, each robot is controlled by a number of building blocks or schemas that can be categorized into the following groups: environmental sensors, perceptual schemas, motor schemas, and communication schemas. By connecting the different schemas to each other in the correct way, the information required by a task can be retrieved through an information flow that reaches from environmental sensors to motor schemas. The principle with ASyMTRe is to connect the different schemas in such a way that a robot team is able to solve tightly-coupled tasks by information sharing. This is done automatically using a greedy algorithm that starts with handling the information needs for the less capable robots and continues in order to the most capable robots. For each robot that is handled, the algorithm tries to find the robots with least sensing capabilities and maximal utility to provide information.

The work on combining task allocation with coalition formation [Tang and Parker, 2007] and our approach to combine action planning and configuration planning also have parts in common, but focus is on different problems. Their approach assumes an already existing partial ordered plan and based on that they generate "configurations" at run-time (similar to what we call independent action and configuration planning, Section 5.3). Our approach generates a total ordered plan and considers configurations for the individual actions at

planning time (i.e., a *configuration plan* is generated) to assure that the task has a solution. To enable several actions to be executed concurrently, we analyze the configuration plan at execution time. On the other hand, our approach does not include task allocation to assign the tasks to the robots. We believe that the main reason for these differences is the different objectives of the two approaches. The ASyMTRe approach has its origin in an ambitious DARPA project with the aim to build a sensor network with a large number of robots (originally 100+, later 70+) [Parker, 2003b, Parker et al., 2004]. Some tasks in building the sensor network require tight coordination (e.g., leading and guiding simple robots to their sensing locations). In the actual project, the tight coordination was set up manually, but as a continuation of the project, ASyMTRe was presented for autonomous sensor-sharing for tightly-coupled cooperative tasks [Parker et al., 2005]. The aim is to generate an optimal configuration for a team of robot. i.e., to find a configuration that resolves the information requirements of all robots in an optimal way. It can be viewed as the configuration is generated at a level above the individual robots; a higher level that is telling who should help whom and how. The goal of our approach is not to fulfill all the needs of all robots in one configuration. Our approach is more at the individual robot level. When a robot is assigned a task, it will generate a configuration, or a sequence of configurations, that helps it to perform the task. This configuration may include other robots that provide valuable information but it may also only include the robot that got the task. In this way, each robot that gets a task is responsible on its own to generate a configuration that gathers the help it needs.

There are other works that are concerned with the problem of coalition formation for multi-robot systems. Vig and Adams [2006, 2007] present an approach that adapts well-known techniques from the multi-agent domain (i.e., [Shehory and Kraus, 1998]) to the multi-robot domain. However, this work is more concerned on how to find the right team of robots for a task than how to automatically setup the information flow between robots required to solve the task.

Chapter 3
Functional Configurations

The first goal in our research program is to develop a definition of configuration that is adequate for the objectives presented in the Introduction.

In this chapter, the concepts of configuration, functionality, channel and state are given a formal specification.

3.1 States

We assume that the system (i.e., the robots and environment) can be in a number of different states. We do not keep the robot state Y and the environment state X separate, i.e., the set of all states is denoted $S = Y \times X$. A state consists of a number of state variables. Each state variable is assigned a value from a value domain. A value domain can for instance be boolean (true and false), or another sets of object (e.g., rooms, robots etc). The state also specifies different types of resources that can be used. The set of resource types TN constitutes a special class of state variables. For each type of resource $tn \in TN$ the state specifies the total number of resources tn_{max} and the number of currently available resources tn_{avail}. There is a number of robots $r_1, \ldots, r_n$. The properties of the robots, such as what sensors they are equipped with and their current positions, are considered to be part of the current state s_0.

3.2 Functionalities

A **functionality** is a program/process that may use information to produce additional information. A functionality is specified as

$$f = \langle Id, I, O, \Phi, Pr, Po, Re, Cost \rangle . \tag{3.1}$$

Each instance of a functionality has an unique identifier Id i.e., a term of the form $\alpha(c_1, \ldots, c_n)$ that specifies its type, on which specific robot it is located, and additional parameters used to control a functionality, e.g., a navigation

functionality needs a goal location as a parameter. The remaining elements of the functionality tuple represent:

- $I = \{i_1, i_2, \ldots, i_k\}$ is a specification of *inputs*, where $i_j = \langle desc, dom \rangle$. The descriptor (desc) gives the state variable of the input data, the domain (dom) specifies to which set the data belongs. We let $dom(I)$ denote $\times_{j=1}^{k} dom(i_j)$.
- $O = \{o_1, o_2, \ldots, o_m\}$ is a specification of *outputs*, where $o_j = \langle desc, dom \rangle$ as above.
- $\Phi : dom(I) \rightarrow dom(O)$ specifies the *transfer function* from inputs to outputs. There are special cases of functionalities that have $I = \emptyset$, i.e., the functionality encapsulates a sensor. There are also special cases of functionalities that have $O = \emptyset$, i.e., the functionality encapsulates an actuator.
- $Pr : S \rightarrow \{T, F\}$ specifies in terms of state variables the *causal preconditions* of the functionality, i.e., Pr specifies in what states $s \in S$ the functionality can be used. For example, a functionality may have a precondition that specifies that the light must be on. If the light is not on, it is not possible to use the functionality.
- $Po : S \rightarrow S$ specifies the *causal postconditions*. Po is defined in terms of which state variables change and to what values. Po transforms the state s before the functionality was executed into the state s' after the functionality has been executed.
- $Re : Tn \rightarrow \mathbb{N}$ specifies the resources of different types required to execute the functionality. In order to execute the functionality in state s, the number of resources for each type tn defined by Re must be available in s. The resources considered here are *reusable*. That is, while executing a functionality, the functionality needs resources of different types to perform the execution. When it has finished, the resources are released and free to use for some other functionality. Note, that two functionalities can only share a resource of a certain type tn_1 if the total number of required resources of type tn_1 is lower or equal to the number of available resources of type tn_1 in state s. Compare with preconditions where several functionalities can have the same preconditions and still be executed at the same time.
- $Cost_i \in \mathbb{R}$ specifies how expensive the functionality is, e.g., in terms of time, processor utilization, energy, etc. In the remainder of this thesis, cost is a single scalar that summarizes these factors, i.e., $Cost \in \mathbb{R}$.

Functionalities can incorporate sensors and/or actuators: e.g., a camera can be encapsulated in a camera functionality. Functionalities that incorporate actuators generally require the resource that controls the actuator. This is to ensure that only one functionality controls an actuator at a time. Functionalities usually run continuously while active, but some may terminate after some condition is met — e.g., a navigation functionality terminates when the goal is reached. We call a functionality of this type a *terminating* functionality.

Note that we view functionalities as black boxes that interact in a simple way trough their inputs and outputs. If we wanted to capture functionalities that can interact in more complex ways, a model describing the internal workings of each functionality may be needed. Since functionalities are communicating processes, they could be formally specified using some form of process algebra such as the π-calculus [Milner et al., 1992] or Petri Nets [Petri, 1962]. The simpler model adopted here, however, suffices for the goals of this thesis.

3.3 Channels

A **channel** $ch = \langle f_{send}, o, f_{rec}, i \rangle$ transfers data from an output o of a functionality f_{send} to an input i of another functionality f_{rec}.

3.4 Configurations

A **configuration** c is a pair $\langle F, Ch \rangle$, where F is a set of functionalities and Ch is a set of channels. Each channel connects the output of one functionality in F to the input of another functionality in F. There must be at least one terminating functionality in a configuration that determines when the configuration has completed its task.

From the functionalities and channels of a configuration it is possible to define other important attributes of a configuration c such as [1]:

- $Pr_c = \bigwedge_{f \in F} Pr_f$, i.e., the conjunction of all preconditions for the functionalities in F.
- $Po_c = \circ_{f \in F} Po_f$, i.e., the composition of all postconditions for the functionalities in F, where $\circ_{f \in F} Po_f(s) = Po_{f_n}(Po_{f_{n-1}}(\ldots Po_{f_1}(s)))$.
- $Re_c(tn) = \sum_{f \in F} Re_f(tn)\ \forall tn \in Tn_c$ where Tn_c represents the set of all different types of resources in c. That is, the resources required by a configuration are categorized in different types, and for each type there is a certain number required.
- $Cost_c$ is based on the cost of the involved components. The formula for configuration cost is given later in this section.

[1] We use subscripts to refer to specific elements in a functionality tuple, e.g., I_f refers to the field I in the tuple of f.

- $R_c = \{r \mid r \text{ is the robot of functionality f, and } f \in F\}$ Thus, R_c is the set of all robots used in the configuration.

An example of how the configuration cost $Cost_c$ can be computed is by aggregating the costs of the involved functionalities combined with a fixed cost number for each channel and robot used. The cost function used in our experiments is computed according to the following formula:

$$Cost_c = |R_c| \times 10 + |Ch_c| + \sum_{f \in F} Cost_f \tag{3.2}$$

The cost of the functionalities in the configuration is directly retrieved from the cost specified by the functionality instance itself. This means that functionalities are able to dynamically set their own costs, which opens up for a more market-based approach of deciding which functionality to use. For instance, it is possible to have two different laser range finders L1 and L2, where L1 is consuming less energy than L2 and thus has a lower cost. In addition to the cost of the functionalities there is also a fixed cost for using robots and channels. $|R_c|$ gives the number of robots used by the configuration. To use a robot has an additional cost of 10 units for each robot. $|Ch_c|$ gives the number of channels and for each channel used the cost is increased by one unit. In summary, this example function for calculating the cost favor configurations with few robots, channels and functionalities of low cost.

Even though the cost function that we use is simple, it enables a selection of configurations, and it is sufficient in many cases. However, it does not specifically incorporate any reliability or accuracy measure of configurations or functionalities. This could of course be incorporated in the cost functions specified above. In practice though, accuracy and cost often grow together. For instance, to use a laser range finder is more costly to use than a sonar array, but on the other hand the laser gives higher accuracy than the sonars. Complementary methods can also have different costs and reliability (e.g. in the door-crossing example, a method that requires two robots is more reliable, even though the cost is higher, than a method using only one robot). The question to answer is then "which is most important: cost or accuracy?". This is not an easy question to answer since this often varies from case to case. It is still an important question since it changes which functionality/configuration is most suitable for a situation. A possible solution could be to learn costs and the reliability of functionalities from experience. However in this thesis we use only a cost function for selection specified above, since to also incorporate a reliability/accuracy measure and a learning mechanism of these parameters is beyond the scope of our work.

3.5 Admissibility

In the context of a specific state s, a configuration $\langle F, Ch \rangle$ is *admissible* if and only if the following three conditions are satisfied:

Information admissibility Each input of each functionality is connected to an output of another functionality with a compatible specification:

$$\forall f \in F\ \forall i \in I_f\ \exists ch \in Ch, f_{send} \in F, o \in O_{f_{send}} : \\ ch = (f_{send}, o, f, i) \text{ and } desc(o) = desc(i) \text{ and } dom(o) = dom(i) \tag{3.3}$$

Causal admissibility All preconditions in Pr_c hold in s, and all postconditions in Po_c are causally compatible:

$$Pr_c(s) = T \text{ and } \neg\exists po_i, po_j \in Components(Po_c) : Conflict(po_i, po_j, s) \tag{3.4}$$

where $Conflict(po_i, po_j, s)$ means that $po_i(po_j(s))$ and $po_j(po_i(s))$ would not result in the same state. Conflict implies that po_i and po_j assign different values to the same state variable.

Resource admissibility For each type of resources in Re_c, the number of resources required for this type is less or equal to the amount of resource available of this type in state s:

$$\forall tn \in Tn_c : Re_c(tn) \leqslant tn^s_{avail} \tag{3.5}$$

where tn^s_{avail} is the value of tn_{avail} in s.

3.6 Examples

To illustrate the above concepts, we consider an example inspired by the scenario in the introduction. A robot A is assigned the action of pushing a box through a door. The "cross-door" functionality requires the position and orientation of the door with respect to A. There is a second robot B, and both A and B are equipped with a camera and a compass. The door to cross has a camera on its upper frame. Let's take a closer look at three of the functionalities we will use in this example. Table 3.1 shows the details of the camera functionality. The Id for this functionality consists of Camera, the robot it is located on, and the instance id. The camera functionality describes a camera sensor and therefor lacks any input. The camera-funct (ϕ) produces an image as output: $\langle$Image taken by A, Image$\rangle$. The first field of the output is the descriptor and the second field is the domain. The precondition specifies that there must be light to use the camera. The cost of the camera is 10 units.

Table 3.2 shows a functionality for measuring the position and orientation of the door. The Id of this functionality also has a parameter Door D that specifies which door to measure the position and orientation for. The cost of this functionality is 5 units. The input of this functionality is an image, and the output is the position of the door with respect to robot A. The precondition is that the door is in the field of view of robot A. Φ is a function called Calculate-pos-and-orient-funct that takes an image as input. In this image, the function

Camera	
Id	Camera, Robot A, ID32
I	-
O	$o_1 = \langle$Image taken by A, Image$\rangle$
Φ	Camera-funct
Pr	light-on
Po	-
Re	-
Cost	10

Table 3.1: The camera functionality

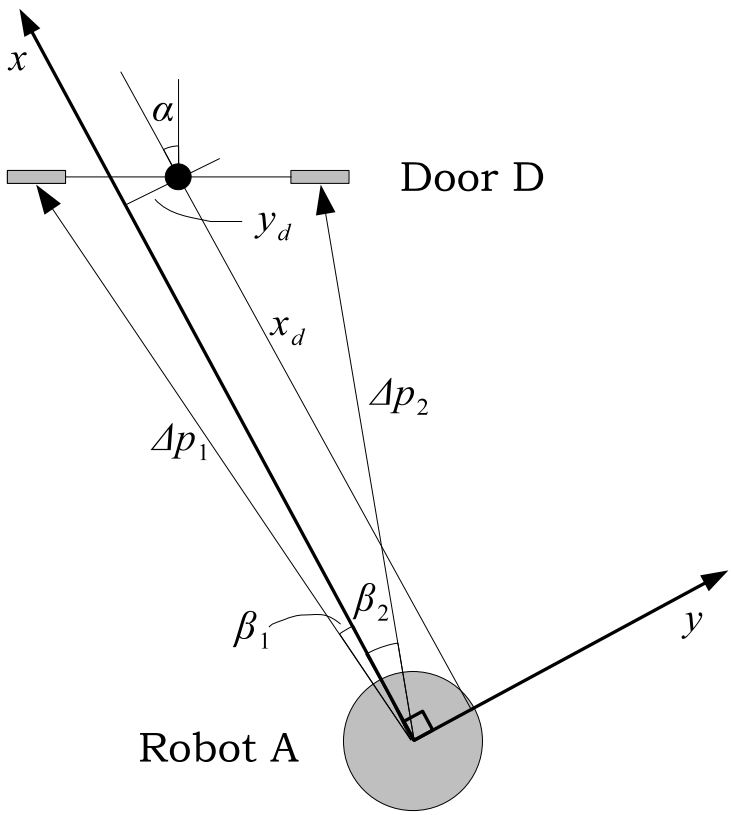


Figure 3.1: A sketch of the calculations for the measure door functionality

Measure-pos-and-orientation-of-door	
Id	Measure-pos-and-orientation-of-door, Robot A, ID20, Door D
I	$i_1 = \langle$Image taken by A, Image$\rangle$
O	$o_1 = \langle$Position of Door wrt A, real-coordinates$\rangle$ $o_2 = \langle$Orientation of Door wrt A, Radians$\rangle$
Φ	Calculate-pos-and-orient-funct
Pr	Visible Door D
Po	-
Re	-
Cost	5

Table 3.2: The measure-pos-and-orientation-of-door functionality

searches for features that can represent the door posts. When two candidates with high enough likelihood to be door post of the Door D are found, the function measures the distances $(\Delta p_1, \Delta p_2)$[2] and the bearings (β_1, β_2) to the posts. Figure 3.1 shows a sketch of how the position and orientation of the door is calculated. x_d and y_d is the requested position and α is the orientation. With the measurement $\Delta p_1, \Delta p_2, \beta_1, \beta_2$, the function calculates the x, y position of the door as the point in between the two posts. The orientation of the door is calculated as the rotation relative to the x-axis in the local coordinate system of robot A.

Table 3.3 shows a cross door functionality. Worth to note for this functionality is that it has preconditions that there must be two rooms r1 and r2 that are connected with a door D. The robot A must be in room r1 and after the execution of the cross door functionality, the postconditions of the functionality state that robot A will be in room r2. To specify the pre- and postconditions of the functionality we use a STRIPS-like language [Fikes and Nilsson, 1971] in which only relevant state variables are given. That is, only the state variables that need to have a certain value are given in the preconditions. For the postconditions only the state variables that change after execution are given, the values of the other state variables are left as they were. This functionality requires one resource of type $\langle$Cross-door, RobotA, Id45$\rangle$. The cost of the functionality is 20. In addition to these three functionalities there are other functionalities for coordinate transformations, compass uses and so on.

Figure 3.2 shows four different (admissible) configurations for the action "cross-door", using the available functionalities. Each configuration $\langle F, Ch\rangle$ is

[2]The distance is obtained by measuring the elevation angle to the base of the posts, assuming that the posts are "standing" on the floor

Cross-door	
Id	Cross-door, Robot A, ID45, Door D
I	$i_1 = \langle$Position of Door wrt A, real-coordinates$\rangle$ $i_2 = \langle$Orientation of Door wrt A, Radians$\rangle$
O	-
Φ	Move-to-cross-door-funct
Pr	There are two rooms r1 and r2 r1 is connected to r2 with door D Robot A is in room r1
Po	Robot is in room r2
Re	$Re(tn_1) = 1$, $tn_1 = \langle$Cross-door, RobotA, Id45$\rangle$
Cost	20

Table 3.3: The cross-door functionality

represented by a directed graph, in which nodes represent functionalities in F and arrows represent channels in Ch.

The first configuration (a) involves only A, the robot performing the action, assuming that its camera can view the door's edges while pushing the box (e.g., if the camera is mounted on a tall stand). The camera delivers information to a functionality that measures the pose of the door relative to the robot.

In the second configuration (b) in Figure 3.2, all information is provided by the camera on the door. This is connected to a functionality that measures the pose of A relative to the door. This pose is transformed to the pose of the door relative to A, and is delivered to A.

In the third configuration (c), robot B provides the needed information to A. The camera on B is connected to a functionality that measures the pose of the door, and to another one that measures the position of A. These measurements are relative to B. In order to compute the pose of the door relative to A, we use a coordinate transformation, for which we need the position and orientation of robot A relative to B. The latter is obtained by comparing the absolute orientations of the two robots measured by their on-board compasses.

The fourth configuration (d) is similar to (c), except that the orientation of A relative to B is obtained in a different way. Both robots have cameras and have a functionality to measure the bearing to an object. When the robots look at each other, each robot can measure the bearing to the other one. By comparing these measurements, we obtain the orientation of A relative to B.

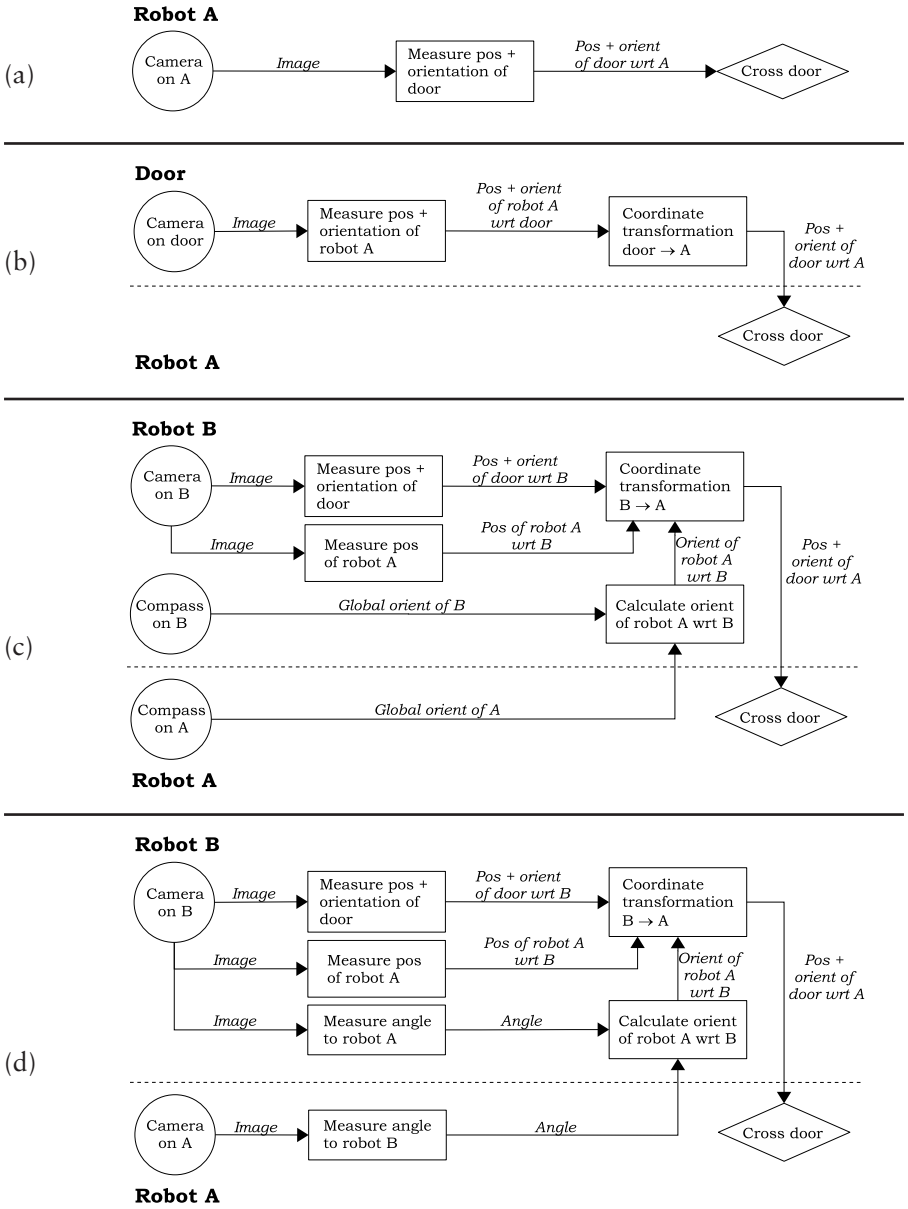


Figure 3.2: Four configurations that provide the position and orientation of a door to a "cross door" functionality. Boxes represent functionalities. Circles and diamonds represent functionalities that have embedded sensing and actuation capabilities, respectively. Arrows represent channels.

Chapter 4
Configuration Generation

This chapter describes the problem of automatic generation of configurations, and how it can be addressed using planning techniques. This includes a problem statement (Section 4.1), a comparison with traditional action planning and how to select a planning technique to use (Section 4.2). Sections 4.3-4.8 describe our approach to the problem and its different parts:

- The representation of the different elements of a functional configuration (Section 4.3).
- A formal definition of the configuration problem (Section 4.4).
- The algorithm used to generate configurations automatically (Section 4.5) and a search strategy to use for the algorithm (Section 4.6).
- A detailed example of how a configuration can be generated given the door-crossing domain (Section 4.7).
- The formal properties of the algorithm (Section 4.8).

Finally, in order to use the configuration algorithm in a real distributed robot system, it needs to be a part of a larger process. In Section 4.9 this top-level process is detailed.

4.1 Problem Statement

Let Σ be a distributed robot system (as described in Section 1.1.1), and let D be a domain describing, in some formalism, all the functionalities that exist in Σ. D implicitly defines the set $\mathcal{C}(D)$ of all the configurations that can be built in Σ (both admissible and not admissible ones). Let then A denote an action, and s denote the current state.

A *configuration problem*[1] $\langle A, D, s\rangle$ for Σ is the problem of finding a configuration $C \in \mathcal{C}(D)$ to perform A, which is admissible in state s.

[1] A more specific definition is given in Section 4.4

The definition of the configuration problem we have chosen is similar to the definition of an artificial intelligence planning problem [Nau et al., 2004]. This is not a coincidence, since we believe that an approach similar to traditional action planning would be suitable to generate configurations automatically. In the next section we discuss the choice of a planning approach for the automatic configuration problem.

4.2 Select a Planning Approach

In order to select a planning approach suitable for generating configurations we need to consider how configuration planning and action planning relate to each other. Action planning (or simply planning) usually refers to the task of creating a sequence of actions that will achieve a goal [Nau et al., 2004]. Each action in the plan has certain conditions that need to hold *before* the action (preconditions), and certain conditions that will hold *after* the action (postconditions). These conditions create causal dependencies between the actions in a plan; a causal flow determines which action should come after another. The planning process can be performed in several different ways. The two basic approaches are progression planning and regression planning. In progression planning, we a start with an initial state and search for different sequences of actions, until we find a sequence that reaches the goal. This is a quite straightforward method, but in its purest, unguided form, it is considered impractical and inefficient. The reason for this is that progression planning considers all actions, even actions that are irrelevant. This means that at each state there can be a large amount of applicable actions that must be considered, i.e., the tree to search becomes huge even for small problems. However, with heuristic techniques the search space can be significantly reduced. Regression planning works in the opposite way, the search is done from the goal to the initial state. In this way only actions relevant for achieving the current goal or subgoals are considered, resulting in a smaller branching factor. Both planning techniques have been widely studied in the literature and their applicability and performance depends on the characteristics of the planning problem, the domain, and so on.

Configuration planning differs from action planning in that the functionalities, unlike the actions in a plan, are not temporally and causally related to each other, but related by the information flow as defined by the channels. However, a complete configuration may be seen as a (rather complex) action, with preconditions and postconditions that can be temporally and causally related to other configurations. Thus, a plan for a group of robots corresponds to a set of temporally ordered configurations. This means that in a plan, the actions are executed in sequence. When an action is completed, the causal state will change, and the next action in the plan will become applicable and executed. Functionalities are executed in parallel, and a functionality becomes active when the input data is present. In order to establish the data flow between

functionalities, a mechanism that creates channels between functionalities is required. Such connections are not required for action planning, and obviously, no existing planning technique incorporates such a mechanism.

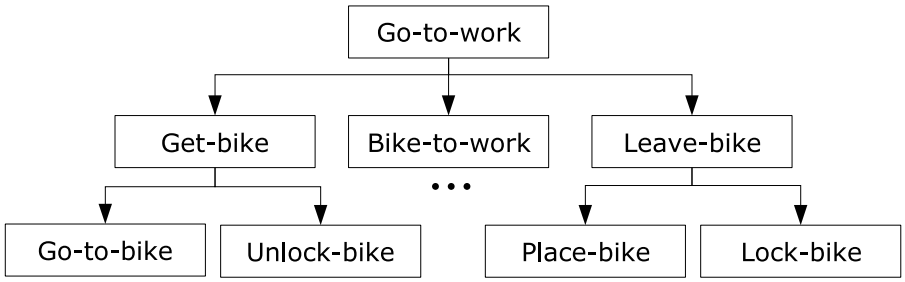


Figure 4.1: An example of an hierarchical structure for action planning

Both progression and regression planning traditionally works to find a plan that achieves a given goal state, i.e., the search is performed in the space of states. There are also planning techniques, such as hierarchical planning, that search for a plan in a different way. In hierarchical planning, the objective is not to achieve a goal state, but rather to perform a set of tasks, i.e., the search is performed in the task space instead in the state space. The idea with hierarchical planning is to take advantage of hierarchical structures of tasks. Hierarchies, in general, have a small amount of activities at each level. This is beneficial since the number of ways to combine the activities at each level is small, and thus of less computational cost compared to non hierarchical methods. The activities at each level in the hierarchical structure can be decomposed to a small number of activities at the level below. This means that in the different levels of the hierarchy the actions have different granularity, i.e., lower levels are more decomposed. For example (see Figure 4.1), the action **Go-to-work** can be decomposed into the actions **Get-bike**, **Bike-to-work**, and **Leave-bike**. Each of these actions can be further decomposed, e.g, **Get-bike** can be decomposed to **Go-to-bike** and **Unlock-bike**. When no further decomposition is possible, the primitive actions are reached, i.e., the actions that can be executed. The other actions used to combine other actions are referred to as methods. There is also a possibility to give several methods to have alternative solutions. For example, the method **Go-to-work** above could exist in several versions, considering actions with a car, a train, or similar instead of a bike. Algorithm 4.1 gives the basic steps for a hierarchical planner.

In order to combine functionalities to form admissible configurations that solve specific tasks, we have chosen to use techniques inspired by hierarchical

Algorithm 4.1 An algorithm for hierarchical planning.

Input: A planning problem $\langle G, D, s \rangle$
Output: An action plan $P = \langle a_1, \ldots a_k \rangle$

1. Take the first element α of G.
2. If α matches one or more action operators, non-deterministically select one operator o and instantiate it to action a. Check that the preconditions of action a holds. If the preconditions holds, add a to the end of P. Else, report failure and backtrack.
3. Else if α matches one or more method schemas, non-deterministically select one expansion m of the one of the matching methods. (This is a backtrack point). Add the methods of the expansion m to the top of G. If there is none, report failure and backtrack.
4. If G is empty, return P. Otherwise go back to 1.

planning, in particular the SHOP planner [Nau et al., 1999]. For the configuration generation problem, hierarchies can be used to specify how different functionalities can be combined on different levels into partial configurations, which in their turn can be combined into complete configurations. This is a very straightforward way to convert a planning technique to be used for automatically generating configurations. We have also successfully tested both progression and regression planning techniques to generate configurations automatically. However, based on the implementation details of the different planners and our previous experience, we selected the approach based on hierarchical planning for the purpose of this thesis. This is also the approach used in all the experiments. In Section 4.3.2 we describe how the different functionalities are combined with hierarchies of methods to form configurations. In the illustrative example of this chapter (Section 4.7), a hierarchy of methods is shown in Figure 4.9.

It should be noted that techniques different from AI planning could potentially be used to automatically generate configurations. For instance, since a traditional planning problem can be formulated as a constraint satisfaction system, the configuration generation problem can also be formulated in similar terms [Kumar, 1992]. However, in this thesis we will not only consider single configurations, but also sequences of configurations (See Chapter 5). The sequence of configuration problem can be seen as a planning problem on two levels: first the planning problem for generating the sequence, and then as the planning problem to generate configurations for each step in the sequence. To use planning techniques for both levels gives a solution that shares many con-

cepts throughout the different levels in the implementation, for instance the same state can be used for both planners.

4.3 Representation

In order to generate configurations automatically, we need a declarative representation of the different concepts in Chapter 3 such that the planning algorithm can use them. That is, the planner needs to know which functionalities are available (the domain), the state for which the configuration should be generated, and the goal of the configuration. Since we use a hierarchical planner the domain also contains methods for how functionalities can be combined. In this section, we detail the representation of functionalities, methods and state, but also what the planning algorithm outputs, i.e., the representation of a configuration.

4.3.1 Functionality Operator Schemas

The description of functionalities is realized using functionality operator schemas (or simply *operators*) similar to those of AI action planners. These operators specify the input and output information for each functionality, and may also include causal pre- and postconditions. The structure of the operators is shown in Figure 4.2.

```
(functionality
  :name      (functionality-name robot id parameters)
  :input     ((descriptor domain) ...)
  :output    ((descriptor domain) ...)
  :precond   condition(s)
  :postcond  condition(s)
  :resource  resource(s)
  :cost      default cost
  :term      terminating functionality
)
```

Figure 4.2: The structure of a functionality operator schema.

With respect to the formal elements of a functionality (Equation 3.1 in Section 3.2), the meaning of the fields in this operator is as follows. The name (*functionality-name*) together with the parameters (*robot, id, parameters*) is the Id of the functionality. The fields `input` and `output` correspond to I and O, and represent the data flow associated with the functionality. The input field specifies the type of data required by the functionality in terms of descriptor (desc) and domain (dom). The output field is on the same format as the input

field, but specifies the type of data the functionality is producing. The two optional fields `precond` and `postcond` correspond to Pr and Po, and represent the causal aspects of the operator. As mentioned in Chapter 3, the pre- and postconditions of the functionality are given in a STRIPS-like language [Fikes and Nilsson, 1971], i.e., only relevant state variables are given. The cost field of the operator corresponds to the Cost of the functionality. The term field specifies whether the operator is for a terminating functionality or not. Note that the Φ transfer function is not part of the operator; it corresponds to the actual code for executing the functionality.

Figure 4.3 shows three operators from the example in the previous chapter: `camera`, `measure-door`, and `cross-door`.

The variables of the operator are marked by a `?`. When a functionality operator is instantiated, the variables are bound. The output from `camera` is an image taken by the camera located on robot `?r`. Since `camera` is a sensor, no input is specified. In `measure-door` we have an image taken by camera `?r` as input and from that we are able to compute the position and orientation of the door `?d` relative to `?r` as output. The terminating functionality, `cross-door`, takes as input the position and orientation of the door, in coordinates relative to the robot crossing the door. Notice that the output of `camera` matches the input of `measure-door` and that the output of `measure-door` matches the input of `cross-door`. Intuitively, this means that channels between these functionalities can be established so that we have an information flow `camera` → `measure-door` → `cross-door`. This combination is equivalent to the configuration in Figure 3.2.a.

On the first row of the preconditions for `camera` the variable `?r` is to be bound to elements of the domain `robot`, i.e., those elements *e* for which (`robot` *e*) holds. On the second row the variable `?rm` is to be bound to elements of the domain `room`, and in addition it is required that the robot `?r` is in room `?rm` and that there is light in `?rm`. For `measure-door` the preconditions is that the door `?d` is fully visible in the input image, and for `cross-door` that the door is open. The variables are bound by evaluating the formulas in a state.

4.3.2 Methods

In order to combine functionalities to form admissible configurations, the configuration planner uses methods that describe alternative ways to combine functionalities (or other methods) for specific purposes. This is a technique inspired by hierarchical planning, in particular the SHOP planner Nau et al. [1999] as described in Section 4.2.

A *method* $\mathtt{m}$ has the form $\mathtt{m} = \langle \mathtt{Id}, \mathtt{Pr}, \mathtt{I}, \mathtt{O}, \mathtt{B}, \mathtt{Ch}, \mathtt{Iconn}, \mathtt{Oconn} \rangle$ where:

- Id is the method id, i.e., a term of the form $\alpha(\mathtt{c}_1, \ldots, \mathtt{c}_n)$;
- Pr is the set of preconditions;

```
(functionality
  :name     (camera ?r ?id)
  :input    -
  :output   (((image ?r) image))
  :precond  (((?r) (robot ?r))
             ((?rm) (room ?rm) (and (in ?r = ?rm) (light ?rm = t)))
  :postcond -
  :cost     3 )

(functionality
  :name     (measure-door ?r ?id ?d)
  :input    (((image ?r) image))
  :output   (((pos ?r ?d) real-coordinates)
             ((orient ?r ?d) radians))
  :precond  (((?r) (robot ?r))
             ((?d) (door ?d) (visible ?r ?d = t)))
  :postcond -
  :cost     2 )

(functionality
  :name     (cross-door ?r ?id ?d)
  :input    (((pos ?r ?d) real-coordinates)
             ((orient ?r ?d) radians))
  :output   -
  :precond  (((?r) (robot ?r))
             ((?rm) (room ?rm) (in ?r = ?rm))
             ((?rm2) (room ?rm2) (not (?rm = ?rm2)))
             ((?d) (connects ?d ?rm ?rm2)))
  :postcond ((in ?r = ?rm2))
  :resource ((re cross-door ?r ?id))
  :cost     3
  :term     T )
```

Figure 4.3: Three different functionality operators.

- I and O are specifications of inputs and outputs, given as sets of $\langle \mathrm{desc}, \mathrm{dom} \rangle$ pairs;
- $B = \{m_1, \ldots, m_k\}$ is the body of the method, where each m_i is either a functionality or method id;
- Ch is a set of internal channels; and
- Iconn and Oconn are sets of $\langle \langle \mathrm{desc}, \mathrm{dom} \rangle, m' \rangle$ that connect inputs in I and outputs in O, respectively, to methods and functionalities in the body B.

Each internal channel $ch \in Ch$ has the form $\langle m'_1, o_1, m'_2, i_2 \rangle$, where $m'_1, m'_2 \in B, o_1 \in O_{m'_1}$ and $i_2 \in I_{m'_2}$. Intuitively, $\langle I \cup B \cup O, Ch \cup Iconn \cup Oconn \rangle$ can be seen as a directed graph, with input nodes I and output nodes O.

Method Schema

A *method schema* represents a class of methods, and has the same form as a method, but contains variables. The Id component contains parameter variables, and Pr may contain additional (free) variables, which also need to be bound. For simplicity, we use the term method to refer to both methods and method schemas.

The structure of a method schema is shown in Figure 4.4. The structure

```
(config-method
  :name      (method-name robot parameters)
  :input     f#: (descriptor domain)
             ⋮
  :output    f#: (descriptor domain)
             ⋮
  :precond   condition(s)
  :postcond  condition(s)
  :channels  (f# f# (descriptor domain))
             ⋮
  :body      f#: (functionality/method-name robot id parameters)
             ⋮
)
```

Figure 4.4: The structure of a method schema.

is similar to the structure of the operator for a functionality. The important differences are that a method contains the fields “channels” and “body”, and

does not have the fields resource, cost, and term. The Id of the method is specified only by the method name. The precond field corresponds to Pr, in to I and Iconn, and out to O and Oconn The body lists the functionalities and/or methods used by the method. Each functionality/method is marked with a label (?f1, ?f2, . . . , ?fn). The channels field, which corresponds to Ch, lists the channels that should be created for the method. Each channel specifies which functionalities/methods it connects (using their labels) and the type of data it transfers. The input of a schema corresponds to Iconn of the method and is a list of *(descriptor domain)* that are labeled with the functionality/method in the body it should be connected to. In a similar way, the output (Oconn) of a method specifies the output data and which functionality/method in the body that is producing it. It is required that connected functionalities/methods have an output/input of the same type.

```
(config-method
  :name      (get-door-info ?r ?d)
  :precond   (((?r) (robot ?r))
              ((?d) (door ?d))
              ((?cid) (funct ?r camera ?cid))
              ((?mid) (funct ?r measure-door ?mid)))
  :in        -
  :out       ((?f2 ((pos ?r ?d) real-coordinates))
              (?f2 ((orient ?r ?d) radians)))
  :channels  ((?f1 ?f2 ((image ?r) image)))
  :body      ((?f1 (camera ?r ?cid))
              (?f2 (measure-door ?r ?mid ?d))) )
```

Figure 4.5: A method for combining a camera and a measure-door functionality.

Figure 4.5 shows an example of a method schema with Id = (get-door--info ?r ?d) that connects a camera functionality and a measure-door functionality on the same robot in order to obtain position and orientation of a given door. The preconditions field of the method decides when the method is applicable, and it also binds the free variables. For the method above, the ids of the functionalities used in the body (?cid and ?mid) must be bound. There may of course be several matching ids for a method yielding several possible expansions of the method. In channels a channel is specified that connects two functionalities in the body (labeled ?f1 and ?f2). This channel links the output (O) of functionality (?f1 (camera ?r, ?cid)) to the input (I) of functionality (?f2 (measure-door ?r ?mid ?d)), and has desc = (image ?r) and dom = image. The output of ?f2, ((pos ?r ?d) real-coordinates)) and ((orient ?r ?d) radians), is in the out field, so this is the output of the entire method. Thereby, any channel in a method higher up in the expansion hier-

archy which is connected to the output of `get-door-info` will be re-connected to the output of `measure-door`.

Expansion of Method

A method m with $Id = \alpha(v_1, \ldots, v_n)$ is expanded given a method schema with *Id* $\alpha(c_1, \ldots, c_n)$ as follows. Let σ be the substitution that unifies the two terms. The preconditions $Pr\sigma$ of the method schema are tested, yielding a new set of substitutions $\{\sigma_1, \ldots \sigma_h\}$ also including bindings for the free variables in Pr; if the preconditions do not hold, that set is empty. The result of the expansion is a set of methods $\{m\sigma_1, \ldots, m\sigma_h\}$ where $m\sigma_i = \langle Id\sigma_i, Pr\sigma_i, I\sigma_i, O\sigma_i, B\sigma_i, Ch\sigma_i, Iconn\sigma_i, Oconn\sigma_i \rangle$. Figure 4.6 shows a possible expansion of method `get-door-info` (Figure 4.5) with `?r = pippi` and `?d = door1`. There can be several methods with the same *Id*. Their total expansion is then the union of their individual expansions. Note that all methods with the same Id must have the same I and O (although the elements in I and O can be connected differently.) Having several methods with the same Id is a way to have alternative ways to obtain the same O given the same I. For instance, an alternative method for Id `get-door-info` could use a laser instead of a camera to measure the door position.

```
(config-method
  :name      (get-door-info pippi door1)
  :precond   ((robot pippi)
              (door door1)
              (funct pippi camera id204)
              (funct pippi measure-door id43))
  :in        -
  :out       ((F22 ((pos pippi door1) real-coordinates))
              (F22 ((orient pippi door1) radians)))
  :channels  ((F21 F22 ((image pippi) image)))
  :body      ((F21 (camera pippi c204))
              (F22 (measure-door pippi id43 door1))) )
```

Figure 4.6: A method expansion with pippi as the robot and door1 as the door.

A functionality operator o is expanded in a similar way, yielding a functionality $o\sigma$.

We call *configuration domain* a pair $D = \langle F, M \rangle$ where F is a set of functionality operators, and M is a set of methods.

It is possible to define methods that are recursive, i.e, a method that includes itself (directly or indirectly) as a sub method. However, to avoid infinite expansion of methods, it is necessary to forbid methods that are *front recursive*. A front recursive method is defined in a way such that methods are recursively

```
(config-method                  (config-method
  :name  (a x)                    :name  (a x)
  :body  ((F1 (a x))              :body  ((F1 (b x)) ...))
          (F2 ...)
              ...)                (config-method
)                                   :name  (b x)
                                    :body  ((F1 (a x)) ...))
```

Figure 4.7: Front recursive methods

expanded without ever adding any functionalities. Figure 4.7 shows the principle of front recursive methods. Either the method includes itself as first sub method (left), or there is a cycle of methods that includes each other as the first submethod (right). In the remaining part of this thesis we assume that all methods used are not front recursive.

4.3.3 State

In addition to the domain, the planner needs a state in order to generate a configuration description. A state is a way to represent the conditions that hold in the system for a given moment. As described in Section 3.1, these conditions are described in terms of state variables. Each state variable is assigned a value from a value domain. A value domain can for instance be boolean (true and false), or other sets of object (e.g., rooms, robots etc).

The state s used for planning configurations consists of two parts: robot state Y and environment state X.

The *robot state* contains information relative to the system itself, i.e., which functionalities are currently available, which functionalities are on/off, their current cost, and the available resources of different types. These facts are represented with different state variables, e.g.:

`(robot pippi)`	–	pippi is a robot.
`(funct pippi camera c1)`	–	pippi has a camera functionality with id c1.
`(cost pippi c1 100 on)`	–	c1 costs 100 units to use, and it is currently active, i.e., it does not need to be initialized before it can be used.
`(re camera pippi c1 1 3)`	–	pippi has a resource camera with id c1 that is currently available to use for 1 functionality ($\mathrm{tn}_{avail} = 1$) and there can be up to 3 users in total ($\mathrm{tn}_{max} = 3$).

The *environment state* is a representation of the facts that currently hold in the environment, e.g., information about rooms and places, how they are connected, etc. These facts are represented in a similar way as for the system state, e.g.:

`(room kitchen)`	–	kitchen is a room.
`(door d1)`	–	d1 is a door.
`(connected d1 kitchen living-room)`	–	kitchen and living-room are connected with d1.
`(in pippi = kitchen)`	–	Pippi is in the kitchen.

When the planner iterates over the methods and functionalities in the domain, it must verify that the functionalities can operate in the current state. This is done by comparing the state with the preconditions of the method or the functionality.

4.3.4 Configuration

Given a domain and a state, the planner generates a configuration description. The format of the configuration description is presented in Figure 4.8.

```
(configuration
  Functionalities:
    (f# (functionality-name robot id))
    ⋮
  Channels:
    (f# f# (descriptor domain))
    ⋮
  Preconds:  Conditions
  Postconds: Conditions
  Resources: Resources
  Terminating-functionalities:  f# ...
  Cost:      Total cost
)
```

Figure 4.8: The structure of a configuration-description.

The configuration description gives the information required to create and use a configuration. It consists of seven fields:

Functionalities: Corresponds to F of the configuration definition in Section 3.4. The functionalities are labeled with automatically generated identifiers (`f#`).

Channels: Corresponds to Ch. For each channel, the description gives the identifiers for the functionalities it connects and where these functionalities are located.

Preconds: Corresponds to Pr_c. Specifies all preconditions that must hold in the state.

Postconds: Corresponds to Po_c. Specifies how the configuration effects the state it is executed in.

Resources: Corresponds to Re_c. Specifies the resources required to use the configuration.

Terminating-functionalities: Lists the identifiers (f#) of the functionalities with terminating conditions.

Cost: Corresponds to $Cost_c$. The cost is calculated according to Equation 3.2 in Page 28.

The Preconditions and Resources of the configurations can be used to verify if the configuration is causal and resource admissible in a state different from the state in which it was generated. This important when sequences of configurations are generated and executed as described in Chapter 5.

4.4 Configuration Problem

Section 4.1 stated the configuration problem in general terms. We now define this problem in more precise terms, for the specific formal framework adopted here.

A *configuration problem* is a tuple (G, D, s) where G is a list of goals (i.e., functionality or method *Ids*), D is the domain specifying the functionalities and methods, and s is a state. A solution to the configuration problem is a configuration $\langle F, Ch' \rangle$. Other elements of a configuration (e.g., preconditions, cost, etc) can be directly derived from F and Ch. If $P = (G, D, s)$, then $\Pi(P, Ch)$, the set of all admissible configurations for G in s for D, is defined recursively as follows. (Ch is the set of channels connected to the goal elements in G. Ch is initially empty.)

(1) If G is empty, then $\Pi(G, D, s, Ch)$ contains exactly one configuration, namely the empty configuration $\langle \emptyset, \emptyset \rangle$. Otherwise, let g be the first goal in G and G' be the remaining goals, and continue as follows.

(2) If g is a functionality *Id* which expands to f, if its preconditions Pr_f hold in s, and the resources of f are available in s, then

$$\Pi(G, D, s, Ch) = \{\langle \{f\} \cup F', Ch' \rangle | \langle F', Ch' \rangle \in \Pi(G', D, s', Ch) \text{ and } \neg \mathtt{PostcondConflict}(f, F', s)\}$$

where s' is the new state after the resources for f have been subtracted.

(3) If g is a functionality *Id*, if its preconditions does not hold in s, or if the resources of f are not available in s, then $\Pi(G, D, s, Ch) = \emptyset$.

(4) If g is a method *Id* which expands to the set M according to the domain D, and the preconditions Pr_m holds in state s, then

$$\Pi(G, D, s, Ch) = \{\Pi(append(B_{m'}, G'), D, s, Ch') | \\ m' \in M \text{ and } Ch' = (Ch \text{ with channels to/from } g \\ \text{reconnected according to } Iconn_m/Oconn_m) \cup Ch_{m'}\}$$

In (2), $\neg PostcondConflict(f, F', s)$ is defined as

$$\neg\exists f' \in F' : \exists po_i \in Components(Po_f), po_j \in Components(Po_{f'}) : \\ Conflict(po_i, po_j, s)$$

The definition of Conflict can be found in Section 3.4.

In (4), the reconnection of channels in Ch for a given m' works as follows. If $\langle m', o, m'', i\rangle \in Ch$ where $\langle o, m_1\rangle \in Oconn_{m'}$, then this is changed to $\langle m_1, o, m'', i\rangle$. If $\langle m'', o, m', i\rangle \in Ch$ where $\langle i, m_1\rangle \in Oconn_{m'}$, then this is changed to $\langle m'', o, m_1, i\rangle$.

We state that this recursive definition of $\Pi(G, D, s, Ch)$ defines all and only the admissible configurations defined by the domain provided that the methods are admissible (defined in Section 4.8). The definition includes *all* admissible configurations according to the domain since in clause (4), it considers all possible expansions of a method. The definition includes *only* admissible configurations since configurations that are not causally and/or resource admissible are excluded according to clauses (2) and (3). In Section 4.8 we provide more details about the formal properties.

4.5 The Algorithm

The configuration planner takes as input a configuration problem $\langle G, D, s\rangle$ where G is a robot goal stack with initially one unexpanded method instance (corresponding to an action), D is a domain, and s is a state. It maintains an initially empty configuration description $c = \langle F, Ch, Pr, Po, Re, Fterm, Cost\rangle$ (see Section 4.3.4) that is also the output of the configuration planner. Algorithm 4.2 shows how it works.

To illustrate how the planner functions, let us assume it is expanding `(l5 (get-door-info pippi door4))` (step 1). It chooses to use the method in Fig. 4.5. First, it replaces `?r` with `pippi` and `?d` with `door4` everywhere in the method. It then looks in s to verify that `pippi` is a robot and that it has a camera functionality (`(f ?r camera ?cid)`; ?cid needs to be bound to an id of an actual camera functionality), and a measure-door functionality (`(f ?r measure-door ?mid)`; ?mid also needs to be bound) (step 3). New labels, say `l7` and `l8`, replace `?f1` and `?f2` in the body of the method. The channel is added

Algorithm 4.2 An algorithm for generating a configuration

Input: A configuration problem $\langle G, D, s \rangle$
Output: A configuration description $\langle F, Ch, Pr, Po, Re, Fterm, Cost \rangle$

1. Take the top element α of G.
2. If α matches the Id of an operator o, first check if there is an instance f' in F with the same Id.
   (a) If there is an instance f', then redirect all channels with f_{send} or f_{rec} equal to α to f' according to the definition of Π above.
   (b) If there is no instance f' in F with the same Id, instantiate o to obtain f and check the following: (I) that f's preconditions hold in s, (II) that f's resources are available in s, and (III) that f's postconditions are not incompatible with the postconditions of the current configuration description. If these criteria hold:
      i. add the resulting functionality f to F_c in the configuration description, add Pr_f to Pr_c, add Po_f to Po_c, and add the resources consumption of f to Re_c.
      ii. If f is a terminating functionality, add the functionality Id to $Fterm$. Calculate the cost of that functionality for that specific instantiation, first by querying the system state, and if that fails by taking the default value from the operator.
      iii. Create a new state s' by subtracting the resources required by f from s.
      iv. Go back to 1.

      If any of the criteria (I – III) does not hold, report failure and backtrack.
3. If α matches the Id of one or more methods in D, non-deterministically select one expansion m of one of the matching methods. (This is a backtrack point). If there is none, report failure and backtrack.
4. Add the expansion m as follows:
   (a) Add the channels Ch_m to the current configuration c.
   (b) Add the functionality and method *Ids* in B_m to the top of G.
   (c) Use the in and out connection fields $Iconn_m$ and $Oconn_m$ to reconnect any channels in c as described in the definition of Π (Section 4.4).
5. If G is empty, return $\langle F, Ch, Pr, Po, Re, Fterm, Cost \rangle$. Otherwise go back to 1.

to the current configuration (step 4a) and the two functionalities `(l7 (camera pippi ca1))` and `(l8 (measure-door pippi md1 door4))` are added to the top of G (step 4b). Finally, we go through the channels already in the configuration, and any channel we find with label `(l5 (get-door-info pippi, door4))` for its out connection is reconnected to `(l8 (measure-door pippi md1 door4))` (step 4c). We proceed to step 5, and then return to step 1. There we find `(l7 (camera pippi c1))` at the top of the stack, which matches a functionality operator and is added to the current configuration (step 2). And so on. Note that a different state can result in a different set of admissible configurations. In the cross-door example, if one robot does not have a compass the third configuration (c) in Figure 3.2 above is not generated.

4.6 Search Strategy

The configuration algorithm has been given in a non-deterministic form and must be complemented with a search strategy. We use best first with branch-and-bound [Lawler and Wood, 1966], using the cost of the partial configuration as a lower bound and pruning when this exceeds the lowest cost among the complete configurations generated so far. The search terminates when all partial configurations on the stack have a higher cost.

4.6.1 Best First Search

The basic idea with best first search is to guide the search by selecting the most promising node for further expansion, e.g., in our case to select the partial configuration with lowest cost. Initially there is of course only one node, but as soon as methods are expanded and instantiated, new nodes are created. In algorithm 4.2 above, new nodes are created when a α matches a method in D (step 3). For each method that is found, a new node is created. A method is found if the preconditions in the schema hold. Since there can be several methods with the same Id, a new node is created for each of the alternatives as long as their preconditions hold. The preconditions also have another role, to bind free variables. There can be several bindings that are valid for the preconditions, and thus a node for each binding (each instantiation of the schema) is created. In our algorithm, each node is a partial configuration with its own cost. The selection of which node to continue the search for is done based on the cost, i.e., the node with lowest cost is selected first. Each time a node is selected, the first element in G for that node is expanded. This results in a new cost for that particular node. When the expansion is done for that node, it is time to select which node to expand next. This selection is again based on which of the current nodes that has the lowest cost. In this way the algorithm takes turns on nodes, always selecting the one with lowest cost for expansion. In practice, this selection can be achieved by having a sorted list of nodes and always picking the first node in this list for expansion.

4.6.2 Branch and Bound

Branch and bound is a way to further control the search and to decide when to terminate the search, and still guarantee that there cannot be any configurations left to expand with lower costs. This is done by maintaining a lower bound. The method can be explained with the following points:

- Expand the partial configurations according to the best first search method above. When the first configuration is fully expanded, keep the cost of this configuration as a lower bound.
- Exclude all partial configurations that have a higher cost than the lower bound and continue expanding the other nodes.
- As soon as a partial configuration has higher cost than the lower bound it is excluded from expansion.
- If a new fully expanded configuration has a lower cost, use this cost as the new lower bound.
- If there are no more partial configurations, return a sorted list of the fully expanded configurations.

It is the possibility to exclude partial configurations with a cost higher than the lower bound that makes this search strategy effective. Hence, there is no possibility that their total cost can be lower than the already fully expanded configuration and it is no use to spend time on expanding these nodes (This is of course based on the fact that the cost cannot be decreased during expansion).

4.7 Explanatory Example

The configuration planner has been implemented and used in a real distributed robot system (See Chapter 7). However, in this Chapter we settle for a simpler example to illustrate how our configuration planner works. In this example, we show how a configuration equivalent to the one in Figure 3.2.c. can be automatically generated. The input to the configuration planner is a goal, a state, and a domain that contains a set of functionality operators and a set of methods.

The set of functional operators defines two functionalities that incorporate sensors (`camera` and `compass`), one that describe actuators (`cross-door`), and four ordinary functionalities (`measure-robot-pos`, `measure-robot-orient--compass`, `measure-door`, and `transform-info`).

The main part of the functionalities above must run on the robot that is observing the door and the door-crossing robot from a distance. We use rh for parameters that are intended to represent the helping robot, and r for the door-crossing robot. The following is an example of a method that connects `camera` with `measure-robot-orient` on the helping robot (rh):

```
(config-method
  :name     (get-robot-orient ?r ?rh)
  :precond  (((?r) (robot ?r))
             ((?rh) (robot ?rh) (not (= ?r ?rh)))
             ((?cid1) (f ?r compass ?cid1))
             ((?cid2) (f ?rh compass ?cid2))
             ((?rid) (f ?r measure-robot-orient ?rid)))
  :out      ((?f3 ((orient ?r ?rh) radians)))
  :channels ((?f1 ?f3 ((orient ?r) radians))
             (?f2 ?f3 ((orient ?rh) radians)))
  :body     ((?f1 (compass ?r ?cid1))
             (?f2 (compass ?rh ?cid2))
             (?f3 (measure-robot-orient ?r ?rid)))
)
```

In a similar way, the methods (get-robot-pos ?r ?rh), (get-door-info ?r ?d), and the top method (do-cross-door ?r ?d) are constructed to make the hierarchical structure shown in Figure 4.9.

Note that we have two alternative versions of (get-door-info) that uses different number of robots. The version detailed in Figure 4.5 requires only a single robot and the other requires two robots. Below, we detail this version together with the top method, do-cross-door.

```
(config-method
  :name     (get-door-info ?r ?d)
  :precond  (((?r) (robot ?r))
             ((?rh) (robot ?rh) (not (=?r ?rh)))
             ((?tid) (f ?r transform-info ?tid)))
  :out      ((?f4 ((pos ?r ?d) real-coordinates))
             (?f4 ((orient ?r ?d) radians)))
  :channels ((?f1 ?f4 ((pos ?rh ?d) real-coordinates))
             (?f1 ?f4 ((orient ?rh ?d) radians))
             (?f2 ?f4 ((orient ?rh ?r) radians))
             (?f3 ?f4 ((pos ?rh ?r) real-coordinates)))
  :body     ((?f1 (get-door-info-direct ?rh ?d))
             (?f2 (get-robot-orient ?rh ?r))
             (?f3 (get-robot-pos ?rh ?r))
             (?f4 (transform-info ?rh ?tid ?r ?d)))
)

(config-method
  :name     (do-cross-door ?r ?d)
  :precond  (((?r) (robot ?r))
             ((?d) (door ?d))
             ((?cid) (f ?r cross-door ?cid)))
  :channels ((?f1 ?f2 ((pos ?r ?d) real-coordinates))
             (?f1 ?f2 ((orient ?r ?d) radians)))
  :body     ((?f1 (get-door-info ?r ?d))
```

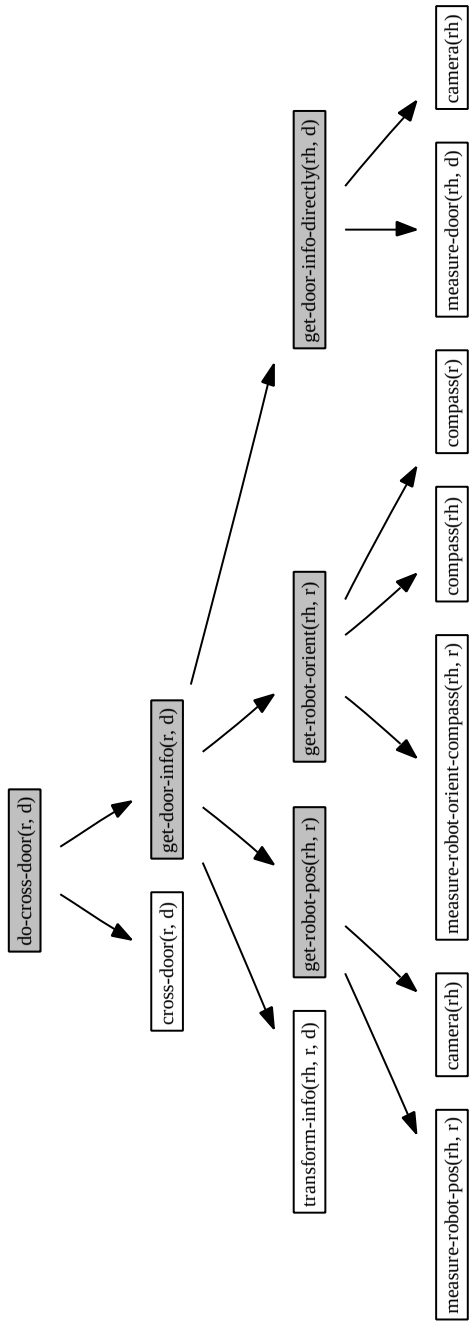


Figure 4.9: The hierarchy of methods (gray) and functionalities (white) for `cross-door`.

```
            (?f2 (cross-door ?r ?cid ?d))
)
```

Since the version with two robots uses the version with one robots, it is necessary to rename the one robot version to (`get-door-info-direct`) and create a wrapper method called (`get-door-info`) that directly calls this version.

The configuration planner needs a state to decide if a method/functionality is applicable. In this example we use the following state that is based on the map in Figure 4.10 and the sensors available:

```
(state
  (robot Pippi) (robot Emil) (door Door1)
  (in Pippi = Room1) (in Emil = Room1) (open Door1)
  (connects Door1 Room1 Room2)
  (funct Emil camera ca1) (funct Pippi camera ca1)
  (funct Pippi compass co1) (funct Emil compass co1)
  (funct Pippi transform-info ti1)
  (funct Pippi cross-door cd1)
  (funct Emil measure-robot-pos mrp6)
  (funct Emil measure-robot-orient-compass mroc3)
  (funct Emil measure-door md1)
  (cost Emil ca1 3 on) (cost Pippi ca1 3 on)
  (cost Pippi co1 3 on) (cost Emil co1 3 on)
  (cost Pippi ti1 2 on) (cost Pippi cd1 5 on)
  (cost Emil mrp6 2 on) (cost Emil mroc3 2 on)
  (cost Emil md1 2 on)
  (re cross-door Pippi cd1 0 1)
  (re compass Pippi co4 0 3)
  (re transform-infoEmil ti1 0 5)
  (re measure-robot-pos Emil mrp6 0 1)
  (re compass Emil co2 0 3)
  (re measure-robot-orient-compass Emil mroc3 0 10)
  (re measure-door Emil md1 0 1)
  (re camera Emil ca1 0 3)
)
```

In this state we have two robots (`Emil` and `Pippi`) equipped with cameras and compasses, a door (`Door1`) that is open, and that all these objects are located in room `Room1`. `Door1` connects `Room1` with a second room (`Room2`). There is also a declaration of the cost of the functionalities.

In addition to the domain and state described above, the configuration planner is also given a goal. The goal G contains a top-method, in this example: (`do-cross-door Pippi Door1`). (`Pippi`) is the robot that should cross the door (`Door1`).

The first step of the planner is to take this method instance at the top of the stack G. This gives us the stack:

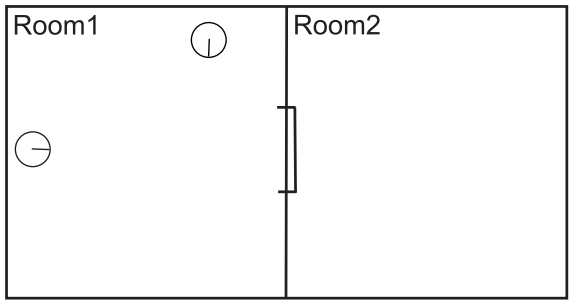


Figure 4.10: A map of two rooms connected with a door. Robots are displayed as circles with a line giving their heading. In room1, Emil is to the left and Pippi to the right.

```
Node 1
(stack
  (MAIN (do-cross-door Pippi Door1))
)
```

The second step is to find an instantiated version of this method that is applicable in the current state. If we instantiate the `do-cross-door` method above, `?r` should be replaced with `Pippi` and `?d` with `Door1`. The preconditions of this method require that `Pippi` is a robot and that `Door1` is a door. There are also preconditions to bind the `?cid` to a functionality with name cross-door. The preconditions of `do-cross-door` are satisfied by the current state and `cd1` can be bound to `?cid`. This gives the bindings: `?r = Pippi`, `?d = Door1`, and `?cid = cd1`.

In the third step, to expand the method, the methods/functionalities of the method are assigned unique labels, instead of unbound labels (?f1, ?f2, ...), and are put on the stack. The channels of the method are added to the current configuration. After the expansion of this first method, the stack and configuration have the following format:

```
Node 2
(stack
  (F108 (get-door-info Pippi Door1))
  (F109 (cross-door Pippi Door1)
)

(configuration
  Channels:
     ((F108 F109 ((pos Pippi Door1) real-coordinates))
      (F108 F109 ((orient Pippi Door1) radians))))
  Cost: 12
)
```

At this stage, the cost of the configuration is 12 (i.e. $1 \cdot 10$ for the robots involved, plus $2 \cdot 1$ for the channels). Next, the planner goes back to the first step and takes the method label at the top of the stack. This label actually matches two different methods schemas: get-door-info in Figure 4.5 with one robot and get-door-info specified in the beginning of this section which uses two robots. This means that we will have two different partial configurations to expand (Node 3 and 4). The algorithm selects to first instantiate ((F108 get-door-info Pippi Door1)) according to the method schema on in Figure 4.5, using the bindings: ?r = Pippi, ?d = Door1, ?cid = ca1, and ?mid = md1[2]. The channels of the method are added to the current configuration. With the new channels added, the cost of the configuration is now 14 (i.e. 1 extra for each channel). The stack and configuration after the expansion of (F108 get-door-info) are:

```
Node 3
(stack
(F110 (camera Pippi ca1))
  (F111 (measure-door Pippi md1 Door1))
  (F109 (cross-door Pippi cd1 Door1))
)

(configuration
  Channels:
     (F110 F111 ((pos Pippi Door1) real-coordinates))
     (F110 F111 ((orient Pippi Door1) radians))
     (F111 F109 ((pos Pippi Door1) real-coordinates))
     (F111 F109 ((orient Pippi Door1) radians)))
  Cost: 14
)
```

The configuration in node 4 still has cost 12, which is lower than 14 for configuration for node 3. According to best first search, node 4 is selected for further expansion. The algorithm instantiate ((F108 get-door-info Pippi Door1)) according to the method schema in the beginning of this section, using the bindings: ?r = Pippi, ?d = Door1 ?rh = Emil, and ?tid = ti1. The channels of the method are added to the current configuration. With the new channels added, the cost of the configuration is now 26 (i.e. 10 extra for a new robot and 1 for each channel). The four methods/functionalities of (F108 get-door-info) are labeled F112-F115 and put on the stack. The out field of the method specifies that the output from (F115 transform-info) is the output from the entire method. This information is used to update the existing channels in the current configuration, i.e., the channels are now connecting (F115 transform-info) with (F109 cross-door). The stack and configuration after the expansion of (F108 get-door-info) are:

[2]This is actually done in two steps, first instantiating the wrapper method get-door-info that in its turn instantiate get-door-info-direct.

```
Node 4
(stack
  (F112 (get-door-info-directly Emil Door1))
  (F113 (get-robot-orient Emil Pippi))
  (F114 (get-robot-pos Emil Pippi))
  (F115 (transform-info Emil ti1 Emil Pippi Door1))
  (F109 (cross-door Pippi cd1 Door1))
)

(configuration
  Channels:
    ((F112 F115 ((pos Emil Door1) real-coordinates))
     (F112 F115 ((orient Emil Door1) radians))
     (F113 F115 ((orient Emil Pippi) radians))
     (F114 F115 ((pos Emil Pippi) real-coordinates))
     (F115 F109 ((pos Pippi Door1) real-coordinates))
     (F115 F109 ((orient Pippi Door1) radians)))
  Cost: 26
)
```

If we look at the costs of the configurations for Node 3 and Node 4, we see that Node 3 has a lower cost, and therefore our best first strategy selects Node 3 for further expansion. During the expansion of the Node 3, the method `(F111 (measure-door Pippi md1 Door1))` in the stack is considered. This method is actually a functionality and cannot be further expanded. Instead it is considered to be included in the current configuration. However, the preconditions for this functionality require that the door is visible for Pippi. This is not true, therefore this node contains an invalid configuration. The algorithm backtracks and continues the search by expanding Node 4 instead.

The next method to expand in Node 4 is `(F112 (get-door-info--directly))`. After the expansion according to the schema (in Figure 4.5), the first method/functionality in the stack for Node 5 will be `(camera Emil ca1)`. This functionality is instantiated and added to the functionality list of the configuration Node 5. After that the measure-door functionality operator is instantiated. In this partial configuration, the camera and measure-door are performed by Emil. Therefore, the door is visible and the measure-door functionality can successfully be added to the configuration. This gives us the following stack, configuration, and state:

```
Node 5
(stack
  (F113 (get-robot-orient Emil Pippi))
  (F114 (get-robot-pos Emil Pippi))
  (F115 (transform-info Emil ti1 Emil Pippi Door1))
  (F109 (cross-door Pippi cd1 Door1))
)
```

```
(configuration
  Functionalities:
    ((F117 (measure-door Emil md1 Door1))
     (F116 (camera Emil ca1)))
  Channels:
   ((F116 F117 ((image Emil) image))
     (F117 F115 ((pos Emil Door1) real-coordinates))
     (F117 F115 ((orient Emil Door1) radians))
     (F113 F115 ((orient Emil Pippi) radians))
     (F114 F115 ((pos Emil Pippi) real-coordinates))
     (F115 F109 ((pos Pippi Door1) real-coordinates))
     (F115 F109 ((orient Pippi Door1) radians)))
  Preconds:
    ((robot Emil) (room Room1)
     (in Emil = Room1) (light Room1 = t)
     (door Door1) (visible Emil Door1 = t))
  Resources:
    ((re camera Emil ca1)
     (re measure-door Emil md1))
  Cost: 32
)
(state
  (re camera Emil ca1 9 10)
  (re measure-door Emil md1 2 3)
  ...
)
```

Notice that the labels in the existing channels are updated. The cost after expanding `F112: get-door-info` is 32 (i.e., 26 plus 3 for camera, 2 for measure-door, and 1 for the channel). Here we also show that the local state changes for Node 5. The resource used by the camera is subtracted in the state, i.e., the 6th parameter that gives the number of resources available is decreased, the 7th parameter gives the total number of resources that can be used at the same time. For resource camera ca1 on Emil above, there are 9 available slots of a total of 10 slots. This does not mean that there are ten cameras ca1, rather that ca1 can handle 10 users at a time.

The planner continues expanding methods and adding functionalities and channels to the configuration until the stack is empty. The final configuration description given by the planner is shown in Figure 4.11. In this description, the functionalities in the list are connected with each other by channels. The description also shows which preconditions must hold in order to execute the configuration, which resources are required, and how the execution would affect the state (the postconditions). Figure 4.12 shows the graph equivalent to this description.

The cost is calculated, as mentioned before, by summering the cost of the functionality instances together with general costs for channels and robots used.

```
(configuration
  Functionalities:
    ((F109 (cross-door Pippi cd1 Door1))
     (F115 (transform-info Emil ti1 Emil Pippi Door1))
     (F131 (measure-robot-pos Emil mrp6 Pippi))
     (F120 (measure-robot-orient-compass Emil mroc3 Pippi))
     (F119 (compass Emil co2))
     (F118 (compass Pippi co4))
     (F117 (measure-door Emil md1 Door1))
     (F116 (camera Emil ca7)))
  Channels:
    ((F116 F131 ((image Emil) image))
     (F116 F117 ((image Emil) image))
     (F117 F115 ((pos Emil Door1) real-coordinates))
     (F117 F115 ((orient Emil Door1) radians))
     (F120 F115 ((orient Emil Pippi) radians))
     (F131 F115 ((pos Emil Pippi) real-coordinates))
     (F115 F109 ((pos Pippi Door1) real-coordinates))
     (F115 F109 ((orient Pippi Door1) radians))
     (F118 F120 ((orient Pippi) radians))
     (F119 F120 ((orient Emil) radians)))
  Preconds:
    ((robot Emil) (robot Pippi) (room Room1)
     (room Room2) (connected Door1 Room1 Room2)
     (in Emil = Room1) (light Room1 = t)
     (door Door1) (visible Emil Door1 = t)
     (visible Emil Pippi = t) (not (?rm = ?rm2)))
  Postconds: ((in Pippi = Room2))
  Resources:
    ((re cross-door Pippi cd1)
     (re transform-info Emil ti1)
     (re measure-robot-pos Emil mrp6)
     (re measure-robot-orient-compass Emil mroc3)
     (re compass Emil co2)
     (re compass Pippi co4)
     (re measure-door Emil md1)
     (re camera Emil ca1))
  Terminating-functionalities:
     (F109)
  Cost:  52
)
```

Figure 4.11: The final configuration description

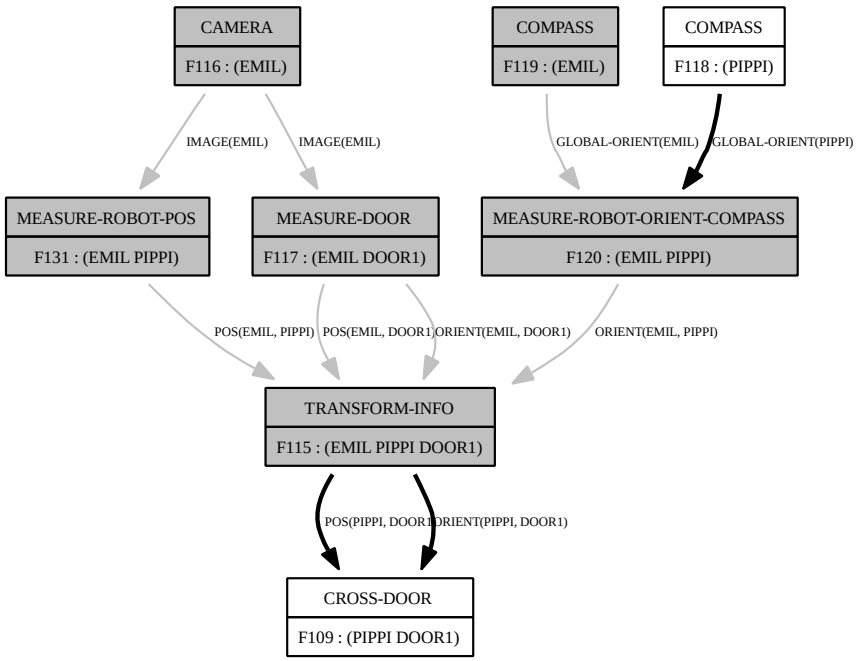


Figure 4.12: The configuration generated in the example. The colors of the boxes show where the functionalities are located; gray is Emil and white is Pippi. Channels are represented with arrows.

The configuration includes two robots, 10 channels, and eight functionalities (of various cost).

$$\mathrm{cost}(c) = 2 \cdot 10 + 10 \cdot 1 + \sum \mathrm{f_{cost}} = 52$$

The cost for the configuration is 52.

4.8 Formal Properties

As expected with a centralized plan-based approach, the configuration planner has good formal properties. We state the following:

Theorem 4.1 (Soundness). *If* D *only contains information admissible methods (defined below), then the configuration algorithm generates only admissible configurations.*

Theorem 4.2 (Completeness). *The configuration algorithm eventually generates any admissible configuration that is defined by the functionalities and methods in* D.

Theorem 4.3 (Optimality). *If the configuration algorithm uses best-first search with branch-and-bound, it returns the admissible configuration with the lowest cost that is defined by the functionalities and methods in* D.

In order to give proofs for Theorems 4.1- 4.3, it is important to define what it means for a method m to be well specified, i.e., *information admissible*. Information admissibility for methods means that all the information used by the sub methods/functionalities in the body of the method, or provided as output by the method, can be obtained from the outputs of the other sub methods/functionalities in the body or via the input of entire method. More formally we define information admissible methods as follows:

Let m be a method with output $\mathrm{O_m}$, body $\mathrm{B_m}$, and internal channels $\mathrm{Ch_m}$. m also has the specifications $\mathrm{Iconn_m}$ and $\mathrm{Oconn_m}$ of how the methods in the body $(\mathrm{m}', \mathrm{m}'', \ldots)$ are connected to the input $\mathrm{I_m}$ and output $\mathrm{O_m}$ of the method. We say that an output $\mathrm{o} \in \mathrm{O_m}$ of the entire method is *bound* if its data are produced inside the method, i.e., there is some $\mathrm{m}' \in \mathrm{B_m}$ such that $\mathrm{o} \in \mathrm{O_{m'}}$ and $\langle \mathrm{o}, \mathrm{m}' \rangle \in \mathrm{Oconn_m}$. For each submethod $\mathrm{m}' \in \mathrm{B_m}$ and each input $\mathrm{i} \in \mathrm{I_{m'}}$ of that submethod we say that i is *bound* if its data are either produced in the method, i.e., there is a channel $\langle \mathrm{m}'', \mathrm{o}, \mathrm{m}', \mathrm{i} \rangle$ such that $\mathrm{dom}(\mathrm{i}) = \mathrm{dom}(\mathrm{o})$, $\mathrm{desc}(\mathrm{i}) = \mathrm{desc}(\mathrm{o})$ and $\mathrm{m}'' \in \mathrm{B_m}$ and $\mathrm{o} \in \mathrm{O_{m''}}$, or supplied as input to the method m, i.e., $\mathrm{i} \in \mathrm{I_m}$ and $\langle \mathrm{i}, \mathrm{m}' \rangle \in \mathrm{Iconn_m}$. Then, we say that a method m is information admissible if: (1) all $\mathrm{o} \in \mathrm{O}$ are bound, and (2) for all $\mathrm{m}' \in \mathrm{B_m}$ and all $\mathrm{i} \in \mathrm{I_{m'}}$, i is bound.

Lemma 4.4 (Information admissibility). *All the configurations in* $\Pi(\mathrm{G}, \mathrm{D}, \mathrm{s}, \mathrm{Ch})$ *are information admissible if: the domain* D *only contains information admissible methods,* $\mathrm{G} = \{\mathrm{m_0}, \ldots, \mathrm{m_k}\}$ *(a set of methods), and* $\forall \mathrm{m} \in \mathrm{G} : \mathrm{I_m} = \emptyset$, *that is each method* m *in* G *does not need any external input.*

Proof. From the definition of information admissibility in Equation 3.3, we have that a configuration is information admissible if and only if each input of each functionality is connected via an adequate channel ($ch \in Ch, ch = (f_{send}, o, f, i)$) to an output of another functionality f_{send} with a compatible specification ($desc(o) = desc(i)$ and $dom(o) = dom(i)$).

From the definition of information admissible methods we have that each input i of a functionality/method in the body B_m of the method m is provided either from a functionality/method m_{send} in B_m via an internal channel (m_{send}, o, f, i) or from I_m. In the first case, if m_{send} is a functionality, the condition is satisfied. If m_{send} is a method, then that method must also be information admissible and its output o must be connected to a submethod or functionality in the body of m_{send}. Going down through methods, eventually we will reach a functionality f_{send}.

In the second case, that is the input i comes from I_m, then m must be enclosed in some other method m' (recall that the top methods m in G had $I_m = \emptyset$). As we assumed that all methods are information admissible, m' must also be an information admissible method, with either some method $m_{send} \in B_{m'}$ with an output connected to $i \in I_m$ (case 1) or with i connected to some $i \in I_{m'}$ (case 2). As there is a finite sequence of methods enclosing f, if the second case keeps repeating itself, one of the top methods in G will eventually be reached. For a top-method m in G, the input of m_{send} will be connected to a method in B_m. □

Lemma 4.5. *All the configurations in* $\Pi(G, D, s, Ch)$ *are causally admissible.*

Proof. Assume the contrary, that there exist a configuration in $\Pi(G, D, Ch, s)$ that is causally inadmissible. We argue that this case can never occur.

From the definition of causal admissibility in Equation 3.4, we have that a configuration is causally admissible if and only if all functionalities in the configuration are applicable in the state, and their effects are causally compatible. For a configuration to be causally inadmissible, it must include functionalities f for which the preconditions $Pr_f(s)$ do not hold, and/or include functionalities with postconditions $Po_f(s)$ that are incompatible with the postconditions of other functionalities of the configuration. Since the preconditions and the postconditions of all functionalities are known, and since in clause (2) of the definition of Π, the preconditions of f is verified with state s and the postconditions of f is compared for compatibility, there cannot exist any causally inadmissible configurations in $\Pi(G, D, s, Ch)$. Thus, all the configurations in $\Pi(G, D, s, Ch)$ are causally admissible. □

Lemma 4.6. *All the configurations in* $\Pi(G, D, s, Ch)$ *are resource admissible.*

Proof. Assume the contrary, that there exist a configuration in $\Pi(G, D, Ch, s)$ that is resource inadmissible. We argue that this case can never occur.

From the definition of resource admissibility in Equation 3.5 we have that for each type of resources in Re_c, the number of resources required for this

type is less or equal to the amount of resource available of this type in state s. In order to have a configuration resource inadmissible, it must be possible to include a functionality which requires more resources than what is available in s. Since the resources requirements of all functionalities are known, since the state s is updated with resources usage, and since the resources of different types are verified with the state, before a functionality is included in a configuration, there cannot exist any resource inadmissible configurations in $\Pi(G, D, s, Ch)$. Thus, all the configurations in $\Pi(G, D, s, Ch)$ are resource admissible. □

We are now in a position to prove soundness, completeness and optimality for our configuration generation algorithm. We assume that the domain D is finite, and G is a set of methods, and that each method in G does not need any external input. For convenience, we repeat the theorems.

Theorem 4.1 (Soundness). *If* D *only contains information admissible methods, then the configuration algorithm generates only admissible configurations.*

Proof. For a configuration to be admissible, it must be information, causal, and resource admissible. For any given configuration problem P, Lemma 4.4, Lemma 4.5, and Lemma 4.6 guarantee that all configurations in $\Pi(P, Ch)$ are information, causal, and resource admissible if D only contains information admissible methods. Since the configuration algorithm is a straightforward implementation of the definition of $\Pi(P, Ch)$, it follows that it generates only admissible configurations. □

Theorem 4.2 (Completeness). *The configuration algorithm eventually generates any admissible configuration that is defined by the functionalities and methods in* D.

Proof. By definition, the set of all configurations for a given configuration problem P is given by $\Pi(P, Ch)$. The configuration algorithm is a straightforward implementation of the definition of $\Pi(P, Ch)$, and it performs a full search since D is finite. Hence, it eventually generates any configuration in $\Pi(P, Ch)$. By Lemma 1, this configuration is admissible. Since this it true for any P, we have the thesis. □

Theorem 4.3 (Optimality). *If the configuration algorithm uses best-first search with branch-and-bound, it returns the admissible configuration with the lowest cost that is defined by the functionalities and methods in* D.

Proof. Follows directly from the completeness of the algorithm in its non-deterministic form, and from the use of branch-and-bound. □

4.9 Top-Level Process

In order to use functional configurations in real distributed robot system, our approach to configuration generation must be embedded in a larger process. In particular, the following aspects should be considered.

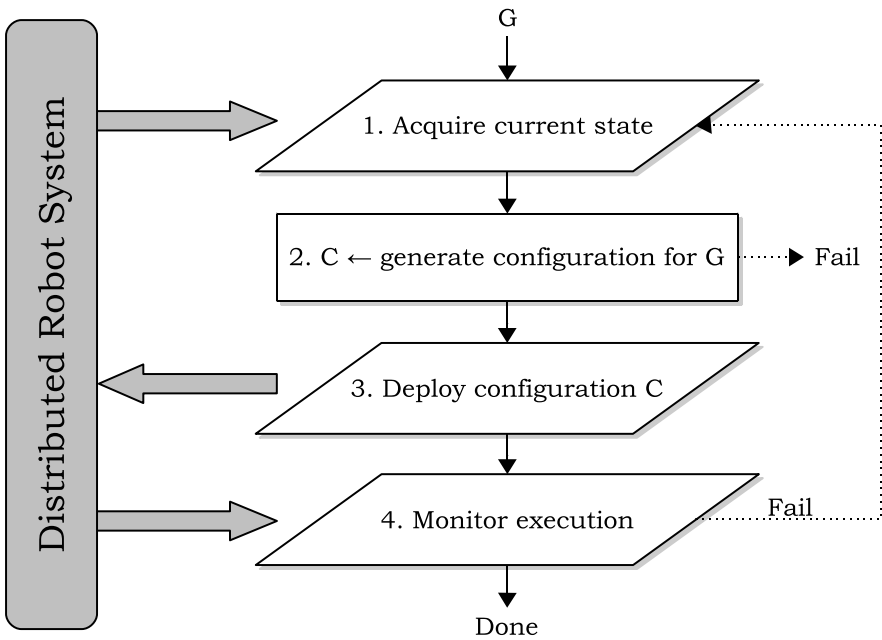


Figure 4.13: Flow chart of the top-level process.

First, configuration planning depends on the current state, which includes both the state of the distributed robot system and the one of the environment. Since we do not assume that the system and the environment are static, this state should be dynamically acquired before the search for an admissible configuration is started. Second, the generated configuration should be instantiated in the actual distributed robot system. Third, execution should be monitored in order to decide when the configuration has finished, and to detect failures.

Figure 4.13 gives an overall view of the top-level process that includes all the aspects above. The top-level process is run by one single robot that configures the distributed robot system to help it solving its task.

4.9.1 State Acquisition

State information is used to ensure that configurations are admissible. As described in Section 4.3.3, our state consists of two parts: robot state and environment state. The *robot state* contains information relative to the internal state of the multiple robots, i.e., which functionalities are currently available, which robotic devices are on/off, and what are their current costs. The *environment state* is a representation of the facts that currently hold in the environment, e.g., information about rooms and places, how they are connected, etc. Algorithm 4.2 assumes that the entire state is acquired before planning starts (step 1 in Figure 4.13). For large distributed robot systems, this could be a time consuming process. In practice the algorithm can be easily modified to acquire the state on demand only for those parts of the system that are currently of interest, and that are not static.

How the state information is acquired depends on the underlying infrastructure of the specific distributed robot system. In Section 7.2.1 we give an example of how state acquisition is performed in the PEIS-Ecology.

4.9.2 Configuration Generation

The second step in Figure 4.13 is to generate a configuration description given the acquired state, a domain, and a goal G. If there is no configuration, the top-level process fails. To generate configurations we use the approach described previously in this chapter.

4.9.3 Deployment of a Configuration

Once a configuration description is generated, it must be deployed on the distributed robot system (step 3 Figure 4.13). This involves three steps: first, to allocate all resources required by the configuration; second, to set up the channels between the functionalities; third, to subscribe to the appropriate signals from the functionalities in the configuration, which announce termination or

failure conditions. Like for state acquisition, how deployment is done in practice depends on the infrastructure of the specific distributed robot system. An example is given in Section 7.2.1.

4.9.4 Configuration Execution and Monitoring

After a configuration has been deployed, execution continues until its main functionality (a terminating one) signals termination. For instance, a "move to" functionality signals termination when its destination is reached. When this happens (step 4 in Fig. 4.13), the execution of the configuration is done.

The normal flow of execution above is marked by the solid arrows in Figure 4.13. Our framework also accounts for possible failures, indicated by the dotted lines in Figure 4.13. If a functionality in the current configuration fails it sends a `fail` signal, and the top-level process tries to generate an alternative configuration for G. If this fails, the top-level process reports failure and ends. This way to monitor configurations assumes that functionalities are able to detect their own failure. Smarter monitoring techniques could be used [Pettersson, 2005, Kaminka et al., 2002, Li and Parker, 2007] but this is beyond the scope of this thesis.

Chapter 5
Sequences of Configurations

The configuration problem described in Chapter 4 is concerned only with finding a configuration for one action. However, in practice most tasks require more than one action to be completed. For instance, if the robot wants to wake up a person, the robot must first reach the bedroom and then move close to the bed, *before* it can wake the person up. Different configurations may be required for each of these actions, thus creating a sequence of configurations that can complete the task. This chapter discusses sequences of configurations, how these can be automatically generated in different ways, and how they can be used in practice. Section 5.1 describes the concept of configuration plans and Section 5.2 discusses their relation to action plans. Section 5.3 discusses three different approaches to the configuration plan problem. One of these approaches is considered to be better than the other two and is given a more detailed description in Section 5.4. The approach to generate sequences of configurations also needs to be part of a top-level process in order to be used in real distributed robot system (Section 5.5).

5.1 Configuration Plans

Let Σ be a distributed robot system, D be a domain for it, and $\mathcal{C}(D)$ the set of configurations in Σ, as defined in Section 4.1. As already stated before, many tasks require a sequence of configurations to be completed rather than a single configuration. We call such a plan, where each step is a configuration, a *configuration plan*.

Definition 5.1 (Configuration Plan). *A configuration plan is a sequence of configurations* $CP = \langle c_1, \ldots, c_k \rangle$, *where* $k \geqslant 0$.

The applicability of each configuration is decided according to the preconditions of its functionalities and in a similar way each configuration can change the state according to the postconditions of its functionalities. Thus, a domain D describing available functionalities can be considered to define a

state-transition system $\langle S, C, \varphi, \gamma \rangle$ with states S, configurations $C = \mathcal{C}(D)$, an applicability function $\varphi : S \times C \rightarrow \{T, F\}$ defined according to the preconditions of the individual configurations, and a transition function $\gamma : S \times C \rightarrow S$ defined according to the postconditions of the configurations (see Chapter 3 for definitions of state). Given a configuration plan $\langle c_1, \ldots, c_k \rangle$, the state-transition function γ defines the states $\langle s_0, \ldots, s_k \rangle$ in which configurations are executed, where s_0 is the initial state.

Definition 5.2 (Admissible configuration plan)**.** *A configuration plan is* admissible *if and only if each* c_i *is admissible in the state* s_{i-1} *in which it will be executed.*

That is each configuration in the configuration plan must be information, causal, and resources admissible in the state in which it will be executed (See Section 3.5).

The domain D implicitly defines the set $\mathcal{CP}(D)$ of all the configuration plans (both admissible and not admissible) that can be built in system Σ. Let then G denote a task (or first order logical formula), and s_0 denote the initial state.

Definition 5.3 (Configuration Plan Problem)**.** *A configuration plan problem* $\langle G, D, s_0 \rangle$ *is the problem of finding a configuration plan* $CP \in \mathcal{CP}(D)$ *to achieve* G*, which is admissible from starting state* s_0*. To achieve* G *means that* G *should hold in state* s_n*.*

In the remaining part we detail and discuss solutions to the configuration plan problem.

5.2 Action Plans

In the standard AI literature [Russell and Norvig, 2003, Nau et al., 2004], an action plan can be defined as a sequence of actions $P = \langle a_1, \ldots a_k \rangle$, where $k \geqslant 0$. An action is defined by its preconditions and postconditions (effects). The domain D describing all available actions can be considered to define a state-transition system $\langle S, A, \varphi, \gamma \rangle$ with states S, actions A, an applicability function $\varphi : S \times A \rightarrow \{T, F\}$ defined according to the preconditions of the individual actions, and a transition function $\gamma : S \times A \rightarrow S$ defined according the postconditions of the actions.

This definition is very similar to the definition of a configuration plan in the previous section. An action can be seen as an abstraction of a configuration only concerned with the causal aspects of the configuration — the configuration specifies how to implement or execute an action. In general, an action can be implemented by several configurations, i.e., for each action there is a set of configurations. The different configurations for the same action can have different pre- and postconditions, however, they all share the same base of preconditions and postconditions from the action. In this way an action can be seen as a generalization of the configurations it represents.

Note that from now on we reserve the term *task* to denote the top-level task, and use the term *action* to denote each individual sub-task performed by each configuration.

5.3 Integrated Action and Configuration Planning

From the definitions above we can see that combining an action planner with a configuration planner would let the robots deal with tasks that require more than one configuration/action to be performed.

There are several ways the combination of an action planner and a configuration planner could be done. These ways can be described with the following variables: (1) If the decisions about what actions to perform (i.e., the action planning) should be taken at planning time or execution time. (2) If the actions should be expanded into configurations (i.e., the configuration planning) at planning time or at execution time. (3) If the action and configuration planning should be done independently of each other or not. We here present three different settings for the variables above.

5.3.1 Independent Action and Configuration Planning

The first setting works by first calling the action planner to find an action plan $\langle a_1, \ldots, a_k \rangle$ for solving a particular task. That is, the decisions about which actions to perform (1) are done at planning time. This plan is then executed action by action. For each action a_i that is performed, a suitable configuration c_i is generated by the configuration planner at the time when the action must be executed. Thus, for (2) the decision about when to expand actions into configurations is taken at execution time. For (3) the action planning decisions and the configuration planning decisions are taken independently of each other.

5.3.2 Fully Integrated Action and Configuration Planning

The second way is to have the planners fully integrated. Both the decisions about what actions to perform (1) and the expansion of the actions into configurations (2) are taken at planning time. The decisions for 1 and 2 are fully interdependent, i.e., the configuration planner is called immediately to generate configurations for each action that is considered during search, so the system is working directly with configuration plans $\langle c_1, \ldots, c_k \rangle$. In this way it is possible to cut parts of the search space based on the availability of configurations and to only create admissible configuration plans.

5.3.3 Loosely Coupled Action and Configuration Planning

Loosely coupled action and configuration planning is based on the idea to generate an action plan and configurations for this plan *before* we start to execute

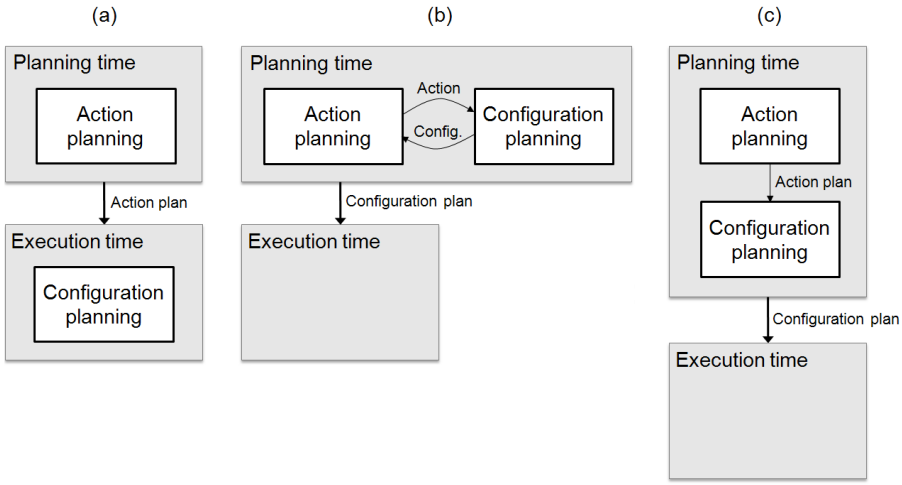


Figure 5.1: Different ways to combine action and configuration planning. (**a**) Independent. (**b**) Fully integrated. (**c**) Loosely coupled.

it. First a complete action plan $\langle a_1, \ldots, a_k \rangle$ is generated, and then for that plan, a configuration is generated for each action: $\langle c_1, \ldots, c_k \rangle$. That is, both the decision on actions to perform (1) and the expansion of actions into configurations (2) are done at planning time as in the fully integrated approach above. However, configuration generation is only done when a complete action plan has been found, in order to validate that plan.

If the action plan is not valid (i.e., there are not configurations for all actions), control returns to action planning to generate an alternative action plan, taking into account information about the failed action and its state, and so on. In this way, it is possible know if there is an admissible configuration plan for the generated action plan.

5.3.4 Conceptual Comparison

In Figure 5.1, the three different approaches are shown side by side for comparison. The independent approach (Figure 5.1a) assumes that the two planning problems can be addressed independently of each other. This approach has problems when an action cannot be expanded into a configuration at execution time. If this happens, a new action plan must be generated that fulfills the goal. Since some actions may be irreversible, there may be situations in which this solution would not be able to complete the task. Even if a new plan can be found, the fact that the actions in the failed plan were executed leads

to a suboptimal performance. For example, consider the task to cross a door mentioned in the introduction. A plan of two actions is generated: (move-to-position Emil pos1), (cross-door Pippi door1). The independent approach then generates a configuration for the first action (move-to-position Emil pos1), executes this configuration, then takes the next action (cross-door Pippi door1) and generates a configuration for this action.

The fully integrated approach (Figure 5.1b) considers both planning problems simultaneously. For the cross door task, this means that while considering the (move-to-position Emil pos1) action in the job of generating the action plan, the configurations for this action are also generated. The same thing happens for (cross-door Pippi door1) and any other action that is considered. In this way, it is possible to guarantee that the generated configuration plans are admissible and optimal. However, since configurations are generated for all actions in the search space (even the actions that do not lead to the goal), the complexity of the problem makes it unusable in most practical cases.

The loosely coupled approach (Figure 5.1c) can, like the fully integrated approach, guarantee that the generated configuration plan is admissible. It avoids the complexity problems of the integrated approach by only trying to generate configurations for actions that are on a path to the goal. That is, first the action plan with the actions (e.g., (move-to-position Emil pos1) and (cross-door Pippi door1)) is created and then a configuration for each one of these actions are generated to verify that the action plan is executable. The price to pay is that global optimality of the configuration plan cannot be guaranteed in general. Compared to the independent approach, the loosely coupled approach can reject bad action plans before they are actually executed, and find better alternatives.

5.3.5 Empirical Comparison

	Loosely coupled		Fully integrated	
	time	configurations	time	configurations
Experiment 3	0.3 s	6	1.0 s	1021
Experiment 4	0.06 s	5	0.2 s	175
Experiment 7	3.4 s	9	14.0 s	14097
Experiment 8	2.4 s	8	9.1 s	8625
Experiment 7'	4.5 s	467	686.8 s	414188

Table 5.1: Results

All three approaches have been implemented and tested on several tasks. Table 5.1 presents the result of a comparison between the loosely coupled approach and the fully integrated approach performed on four different tasks. The

descriptions of the tasks can be found in Chapter 7 on real robot experiments. In this comparison, we measured the time it took to generate a configuration plan for each task and how many configurations that was generated for each task. The test computer was a standard PC with an Intel Pentium 3.4 GHz dual core CPU and 2 GB RAM. It should be noted that the configuration problems for these experiments were relatively simple, and for the loosely coupled approach most of time was spent on finding an action plan. For the task in experiment 7, we also tried with more alternatives to solve each action (i.e. more available functionalities). The results for this test is given in Table 5.1 as Experiment 7'. As expected the results from this comparison shows that the loosely coupled approach is more efficient than the fully integrated approach.

The independent approach and the loosely coupled approach have also been implemented and tested on a real distributed robot system. The independent approach was used in [Lundh et al., 2007a,b, Saffiotti et al., 2008, Lundh et al., 2008b]. However in more recent work [Lundh et al., 2008a], the loosely coupled approach is used. In the remaining part of this chapter we focus on the loosely coupled approach since this gives the best tradeoff between effectiveness and optimality. In Section 5.4.5, we give an example how the loosely coupled approach address a task and compare this with the independent action and configuration approach.

5.3.6 Bibliographical Notes

In the literature there are some works about integrating task planning with more detailed types of reasoning, such as the aSyMov planner [Cambon et al., 2004] which combines symbolic and geometric reasoning. Bouguerra and Karlsson [2005] present an approach for combining several planners for generating actions plans. The approach works by first using one planner for generating a plan consisting of primitive actions and/or abstract actions. Each abstract action is then used as a goal/task for another planner that in its turn generates a plan of primitive actions and/or abstract actions, and so on, until only primitive actions remain. This way of combining planners is very similar to our loosely coupled integration of action and configuration planning (see Section 5.3.3).

5.4 Loosely Coupled Action and Configuration Planning

Figure 5.2 shows a more detailed schema of how the two different planners interact and how the action and configuration planning loop works in the selected loosely coupled approach. In order to generate action plans (step 1), we employ a state of the art action planner called PTLplan [Karlsson, 2001]. An action plan consists of actions like (goto Pippi bedroom), (dockto Pippi bed), and (wakeup Pippi Johanna). This plan is given to the configuration planner

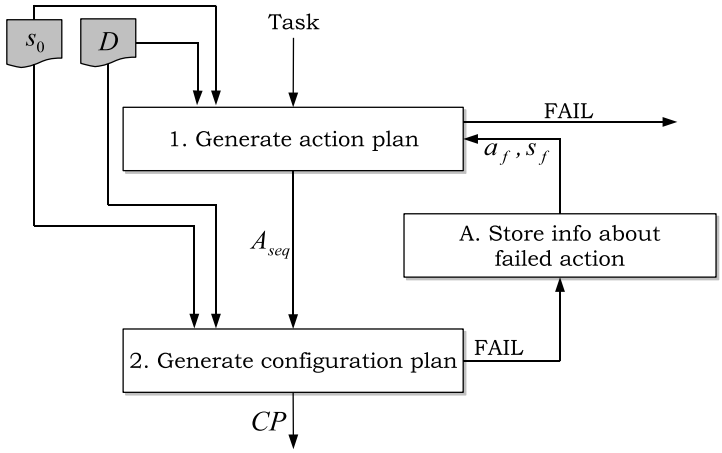


Figure 5.2: An algorithm that implements the loosely coupled action and configuration planning approach.

(step 2 in Fig. 5.2). In this step, a configuration is generated for each action in the action plan, thus creating a configuration plan (and its own sequence of predicted states). If there is a problem of finding a configuration for an action, information about that action and its position is stored (step A in Fig. 5.2). The action planner is then called again, and it removes that particular action in that particular position and then tries to find an alternative sequence of actions that can achieve the task. If an alternative action plan is found, it is given to the configuration generator which again tries to turn it into a configuration plan. These are the steps (1, 2, A) that constitute the action and configuration planning loop. This loop continues until an admissible configuration plan that achieves the task (goal) is found, or until the action planner is unable to find an action plan. In the remaining part of this section we will describe the different steps of the action and configuration planning loop for the loosely coupled action and configuration planning approach. This approach has been used in and tested on a real distributed robot system. Section 5.5 gives details about configuration plan execution, and Section 7.4 describes experiments conducted for this approach.

5.4.1 Representation

Both planners require as input a domain, a state, and a goal. From the previous chapter we know that for the configuration planner, the domain describes the potentially available functionalities and the methods that describe how the

former effectively can be connected. The state describes the currently available functionalities and other properties that currently hold in the environment (e.g., robot positions, rooms, locations, doors, etc). The goal for the configuration planner is the top-method to start the expansion from. The domain for the action planner describes the potentially available actions; the state description is identical to the one of the configuration planner. The representation of the goal for the action planner is different from the goal representation for the configuration planner. For the action planner, the goal is specified using a logical formula that should hold in the states that we like to reach.

For pragmatic reasons, the domain description given in Section 4.3 is extended with operators for actions instead of using separate domains for the two planners. Figure 5.3 shows the structure for an action operator.

```
(ptl-action
:name      (action-type variables)
:precond   (preconditions)
:results   (postconditions)
:execute   (top-method-name variables)
)
```

Figure 5.3: Action operator

The name of the action describes which type of action it is and the variables it uses. The preconditions and the results (i.e., postconditions) are on the same form as for the functionality operators and methods described in Section 4.3.1. The execute field of the operator gives the name of the top-method that should be used when generating configurations for the action.

Figure 5.4 shows an action operator for the action "goto". The preconditions may need some further explanation. The first row puts a condition on **?r** that it must be a robot. The second row requires that **?r** is in a room **?y**, and the third row that **?x** (the room to go to) must be connected to **?y**, and that **?r** is not already in room **?x**. The results for the action is that after executing it, **?r** is in room **?x**. The execute field specifies that the top-method (move-to-room ?r ?x) should be used to generate configurations for the action.

The output of the action planner is a sequence of actions $A_{seq} = \langle a_1,\ldots,a_p\rangle$ where the actions are ordered based on the causal relationship between actions induced by the preconditions and effects of the individual actions: $a_i \preceq a_{i+1}$. A_{seq} is given as input to the configuration planner. The execute field of the action operator for each individual action in A_{seq} is used as a goal for the configuration planner.

The representation of a configuration is slightly modified when it is generated for a specific action. In Section 3.4 on Page 27 the preconditions of a configuration is defined as the conjunction of the preconditions of the func-

```
(ptl-action
 :name    (goto ?r ?x)
 :precond ( ( (?r) (robot ?r))
            ( (?y) (room ?y) (in ?r = ?y) )
            ( (?x) (room ?x) (and (conn ?x ?y) (not (in ?r = ?x)))) )
 :results (in ?r = ?x)
 :execute ( (move-to-room ?r ?x) ) )
```

Figure 5.4: Operator for action goto

tionalities it includes, and the postconditions of a configuration is defined as the composition of the postconditions of the functionalities it includes. To simplify the making of a domain, it is not necessary to duplicate all the pre- and postconditions of an action operator in the functionality operators that implements it. Instead the representation of a configuration also includes the pre and postconditions of the action for which it was generated.

The output of the entire action to configuration plan algorithm is a sequence of configurations (i.e., a configuration plan) where the configurations are ordered in the same way as actions in the action plan it is based on.

5.4.2 Generate Action Plan

The action planner we use is a sensor-based probabilistic action planner, called PTLplan [Karlsson, 2001]. PTLplan is a progression planner that starts from an initial state s_0 and works its way to a goal state s_G, i.e., a state in which the goal formula is true[1]. Figure 5.5 shows an example of how the planner works. Nodes are states and the edges are actions. In state s_0 (step 0), there are three applicable actions (a_1, a_3, a_6). Each one of these actions is considered, resulting in three new states (s_1, s_2, s_3) in step 1. For each of these states there are different (or the same) actions applicable that in their turn result in new states in step 2. The planner can also cut search from a state if a domain-specific control formula is violated [Karlsson, 2001]. The planner explores the search space in this way until a goal state is found (step 3). When a goal state is reached, the planner follows the path back to the initial state in order to create the action plan. For the example in Figure 5.5 we can see that one possible sequence of actions that reaches the goal state is $\langle a_1, a_6, a_7 \rangle$. In the transition graph (Figure 5.5) we can also see that it is possible to have several actions that result in the same state (s_7 step 2). For this particular graph we see that it does not matter in which order actions a_3 and a_6 are applied in state s_0, they result

[1]PTLplan actually works with belief states to account for partial observability and stochastic actions (i.e., a probability distribution over states), however for this application it is enough to just consider states

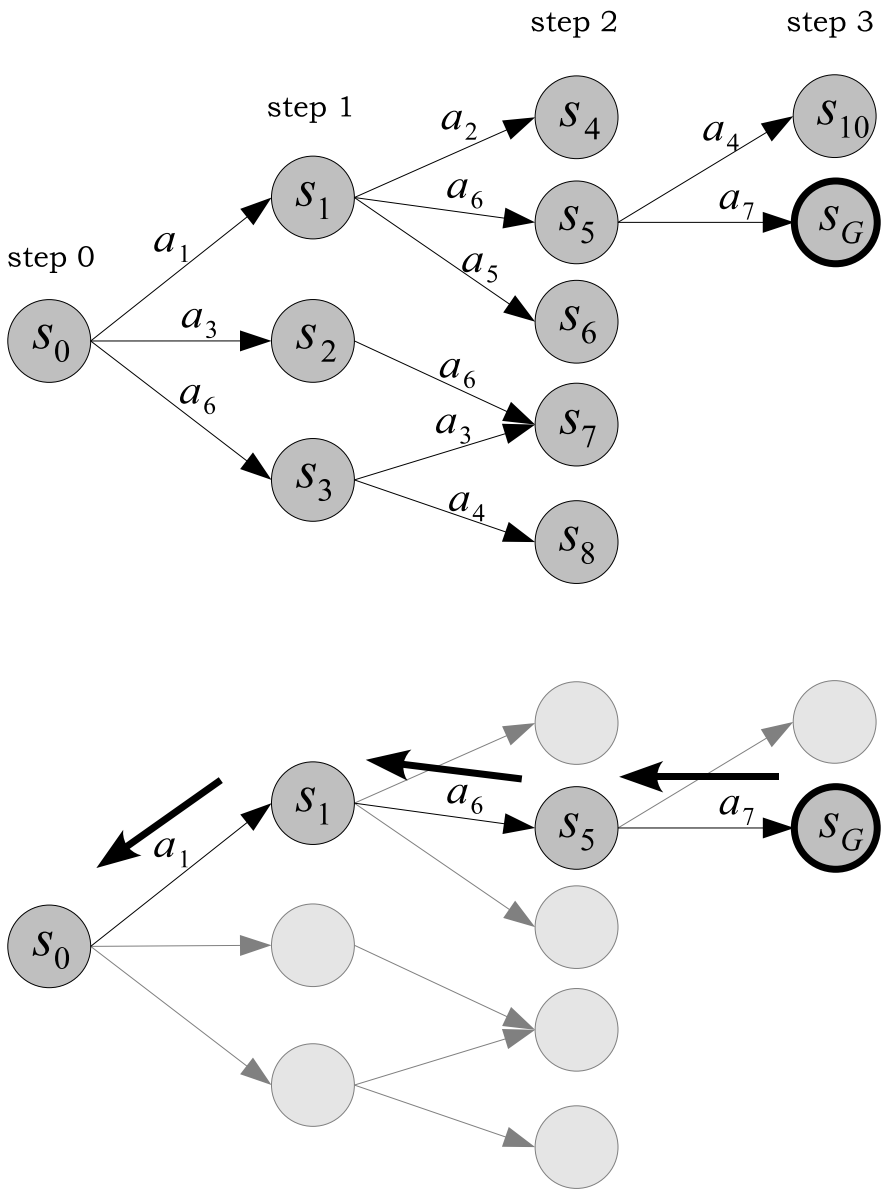


Figure 5.5: Nodes represent states, edges represent actions. **Top:** A transition graph created by PTLplan. **Bottom:** Follow the path back to the initial state to create the sequence of actions that constitutes the plan

in s_7 anyway. However, these are observations outside the sequence of actions that constitutes the action plan. More about the causal relationship between actions (and configurations) can be found in Section 6.4.1.

5.4.3 Generate Configuration Plan

As stated before, the job of the action planner is to find a sequence of atomic actions $\langle a_1, \ldots, a_n \rangle$ that achieve a task G. With an action plan, the job of the configuration planner is to find a configuration c_i for each action a_i in the action plan and thus creating a configuration plan. This problem can be described as an action plan to configuration plan problem.

Action Plan to Configuration Plan Problem and Solution

An *action plan to configuration plan problem* is a tuple (A_{seq}, D, s, G) where $A_{seq} = \langle a_1, \ldots, a_n \rangle$ is a sequence of actions (i.e., an action plan), where each action a_i has an associated configuration goal G_{a_i}, D is the domain specifying the functionalities, methods and actions, s is the initial state that A_{seq} is generated for, and G is the goal of the entire task. Given an action plan to configuration problem $P = (A_{seq}, D, s, G)$, then $CP = \langle c_1, \ldots, c_n \rangle$ is a solution for P if for each i, $1 \leqslant i \leqslant n$, c_i is a configuration for a_i that solves G_{a_i} given by the execute field of a_i, the goal of the entire task G is achieved, and CP is admissible according to Definition 5.2 on Page 68. The set of all solutions to a problem P is denoted $\mathcal{ACP}(P)$.

The Algorithm

Algorithm 5.1 presents an algorithm for the action plan to configuration plan problem. The input to the algorithm is a sequence of actions A_{seq}, a goal G for the entire configuration plan, an initial state s and a domain D. The output of the algorithm is an admissible configuration plan CP, or failure.

The algorithm is given in a non-deterministic form and needs to be complemented with a search strategy. The search strategy needs to define: the properties of the set of configurations generated in step 2 of the algorithm, and which configuration $c \in C$ to select in step 3. First we need to clarify why it is necessary to consider a *set* of configurations for each action and not just one configuration. Let's assume that we get a sequence of actions $\langle a_1, a_2, a_3 \rangle$ to generate a configuration plan for. The algorithm takes the first action (a_1) in this sequence and generates a configuration c_{11} for this action and uses its post-conditions to update the state. Configuration c_{11} is added to the configuration plan. In the next step, when it is time to find a configuration for action a_2, the configuration generation algorithm fails to find any admissible configuration. If there is no configuration for an action, the action plan must be reconsidered. However, we cannot be sure that there is no configuration for action a_2 since

Algorithm 5.1 An algorithm for generating a configuration plan from an action plan

Input: A sequence of actions A_{seq}, a goal G, a state s, and domain D.
Output: A configuration plan CP.

1. Take the first action a in the action plan A_{seq}.
2. Use the configuration algorithm to generate a set of configurations $C = \{c_0, c_1, \ldots, c_l\}$ for the goal G_a, domain D, and state s.
3. Non-deterministically select a configuration $c \in C$, add c to the tail of CP, and use its postconditions to update the state s. (This is a backtrack point). If there is no c, backtrack to previous action.
4. If A_{seq} is empty, verify that goal G is achieved in s and return CP. If the G is not achieved, backtrack to step 3. If A_{seq} is not empty, go back to 1.

we have only tried to generate a configuration when the configuration for the preceding action is c_{11}. There might be other configurations for a_1 with different postconditions. Those postconditions might lead to another state such that there is a configuration for action a_2. Recall that the postconditions of a configuration includes both postconditions of the action and additional postconditions specific for the configuration. By selecting a different configuration for a_1 it may be possible to find a configuration plan for the action plan, without having to regenerating the action plan. This of course assumes that there are configurations that have different effects. If all configurations have the same effects the action plan must be reconsidered.

In the general case, the only way to be sure that a configuration plan is found for an action plan, if there is one, is to define the search strategy such that the complete search space is covered. That is, in step 2 of the algorithm, the set of all admissible configurations must be generated for an action, and in step 3, each configuration in the set must be considered for the configuration plan. However, if we search for all configurations of an action, termination of the configuration planner cannot be guaranteed. Since we allow recursive methods, the configuration planner might get stuck on one configuration problem. In order to guarantee termination of the configuration planner, an upper limit on the number of functionalities in a configuration can be set. When a partial configuration is considered and the upper limit of functionalities is reached, the partial configuration is excluded from further expansion. In this way, the configuration will stop when all partial configurations are expanded to the upper limit and only the fully expanded configurations will be returned. If no configurations are found within this limit, it is possible to backtrack to the previous action. By incrementally increasing the upper limit, we can obtain complete-

ness. In Section 5.4.6, we will prove the completeness and soundness of the algorithm.

In our experiments (described in Chapter 7) however, we use an incomplete search strategy that generates a subset of configurations for each action. The configuration planner generates a set of alternative configurations $C_i^k = \{c_{i1}, \ldots, c_{ik} | k \leqslant n, n = |C_i|\}$ for each action a_i where $C_i^k \subseteq C_i$ and k denotes the k first configurations in C_i. This set is ordered according to the cost of the individual configuration. That is:

$$\begin{aligned} &Cost(c_{i1}) \leqslant Cost(c_{i2}) \leqslant \ldots \leqslant Cost(c_{ik}) \text{ and} \\ &\forall c \in C_i^k \, \forall c' \in C_i \setminus C_i^k : Cost(c) \leqslant Cost(c') \end{aligned}$$

Algorithm 5.2 shows in detail the procedure used in practice for the generation of a configuration plan, given a sequence of actions, an initial state and a domain. The output of the algorithm is a configuration plan CP, provided it is possible to find one. If it fails, the output is the action and the position of the action in A_{seq} that it is unable to find configurations for. Since the algorithm may fail for several actions before it has considered all possibilities, the action that is furthest in the action sequences and its position is returned (P_{max}). The procedure GENERATE-CONFIGURATION-PLAN goes through the sequence of actions A_{seq} in a recursive manner, and stops the recursion when there are no more actions (lines 2 – 7 Algorithm 5.2). If the goal G holds in the final state s (i.e., if the goal is achieved), then the procedure returns success, otherwise it returns fail.

Lines 8 – 33 show how the configuration plan is created based on the sequence of actions. The first step is to take the first[2] action a in A_{seq} (line 9) and to set P_{max} (line 10). On line 11, r configurations are generated for action a[3]. The number of configurations r to generate is set by the user. If it is likely that the configurations fail, it can be wise to generate a larger number of configurations. If the algorithm is unable to find any configurations, it returns FAIL together with action a and the position of the action in A_{seq} (lines 12 – 13). If there are configurations, we need to try to do the same thing for the succeeding actions. In the else statement there is a repeat–until loop (line 15 – 25) that takes care of the recursion and the backtracking step. The first step of the repeat is to take the first configuration c in C and to apply its postconditions to state s resulting in a new state s_x (lines 16 – 17). On line 18, the procedure calls itself recursively with the remaining sequence of actions, the new state s_x, and the domain. This means that the steps above will be repeated until there are no more actions in A_{seq} or if there is an action for which GENERATE-CONFIGURATIONS fails to find any configurations. If there is a fail, the result (the action and its position) will be propagated back trough the recursive calls (lines 12 – 13 again).

[2] POP takes the first element of a sequence and removes the element from the sequence.

[3] When configurations are generated for actions succeeding the first action, we assume that all the resources of different types that exist in the state are available for configuration.

Algorithm 5.2 A procedure for generating a configuration plan

Input: A sequence of actions A_{seq}, goal G, state s, and domain D.
Output: On success: configuration plan CP. On failure: action a_f and the position of a_f.

```
1: procedure GENERATE-CONFIGURATION-PLAN(A_seq, s, D, G)
2:     if A_seq = ⟨⟩ then
3:         if HOLDS(G, s) then
4:             return ⟨SUCCESS, nil⟩
5:         else
6:             return ⟨FAIL, nil⟩
7:         end if
8:     else
9:         a ← POP(A_seq)
10:        P_max = ⟨a,POS(a)⟩
11:        C ← GENERATE-CONFIGURATIONS(a, s, D, r)
12:        if C = ∅ then
13:            return ⟨FAIL, P_max⟩
14:        else
15:            repeat
16:                c ← POP(C)
17:                s_x ← APPLY-POSTCONDITIONS(Po_c, s)
18:                ⟨res, P⟩ ← GENERATE-CONFIGURATION-PLAN(A_seq, s_x, D, G)
19:                if res = FAIL then  ▷ If the result a failed action and its state
20:                    C ← REMOVE-CONFS-WITH-EQUAL-POSTCONDS(C, c)
21:                    if POS(a_P) > POS(a_{P_max}) then
22:                        P_max ← P
23:                    end if
24:                end if
25:            until C = ∅ or res ≠ FAIL
26:            if res = FAIL then
27:                return ⟨FAIL, P_max⟩
28:            else if res = SUCCESS then
29:                return ⟨SUCCESS, APPEND(c, P) ⟩
30:            end if
31:        end if
32:    end if
33: end procedure
```

This is also true when the end of the action sequence is reached, however in this case the configurations for the actions are returned. In either case, the result of the recursive call is stored in the variables $\langle \mathtt{res}, \mathtt{P} \rangle$. res can either be FAIL or SUCCESS. When res is FAIL, P holds the value of a_f and $\mathrm{Pos}(a_f)$. When res is SUCCESS, P holds a (partial) configuration plan CP. If there is a failure (line 19), all configurations that have the same postconditions as c are removed from C (line 20). On line 21 and 22 P_{max} is updated. If there still are configurations left for the action, this loop is repeated and the procedure will again try to call itself but with a state s_x that was created with a different c. This loop continues until either there are no more configurations, or the recursive call actually returns a partial configuration plan (i.e., the algorithm has found configurations for all actions in A_{seq}). If the loop is ended because of failure, it returns the failed actions and its position (lines 26 – 27). This is the step that propagates the failing action back through the recursive calls. If instead we have found a partial configuration plan, append the configuration c and return it (line 29).

It is important to note that for state s_0 (the state that we give the first time we call the procedure) we have the number of available resources for each type defined by $tn^{s_0}_{avail}$. Even though each configuration changes the state according to its postconditions (Apply-Postconditions on line 13), this does not affect or reflect the number of resources of different types for states succeeding s_0 since all resources are returned after each step and no other process can claim the resources. Thus, for configuration generation of configurations for states succeeding s_0, we assume that all resources of each type that exist in state s_0 are available in states s_{0+n}, i.e., $tn^{s_i}_{avail} \leftarrow tn^{s_i}_{max}$ when $i > 0$. That is, the algorithm does not care if the resources are available or not at the moment, it simply assumes that if the resource exist, it is also available. This implies that a verification of resource availability must be performed before configuration execution (See Section 5.5).

5.4.4 Reconsider the Action Plan

If the configuration planning process is unable to generate configurations for a certain action in an action plan, the action plan must be reconsidered. This is done by first storing information about the failing action and its position in the action plan. This information is used to update the transition graph that the action planner used to create the first action plan. Consider the transition graph in Figure 5.5 and that the configuration planner is unable to find a configuration for action a_6 at position 2 in the plan (i.e., in state s_1). This action is removed from the transition graph, but only in state s_1. Hence, there may exist configurations for that action in a different state. When the action link for a_6 is removed, the search continues. Figure 5.6 shows how PTLplan finds an alternative sequence of actions $\langle a_1, a_5, a_8 \rangle$ that leads to a goal state. This action plan is then sent to the configuration planner that again must verify that there exists a configuration plan.

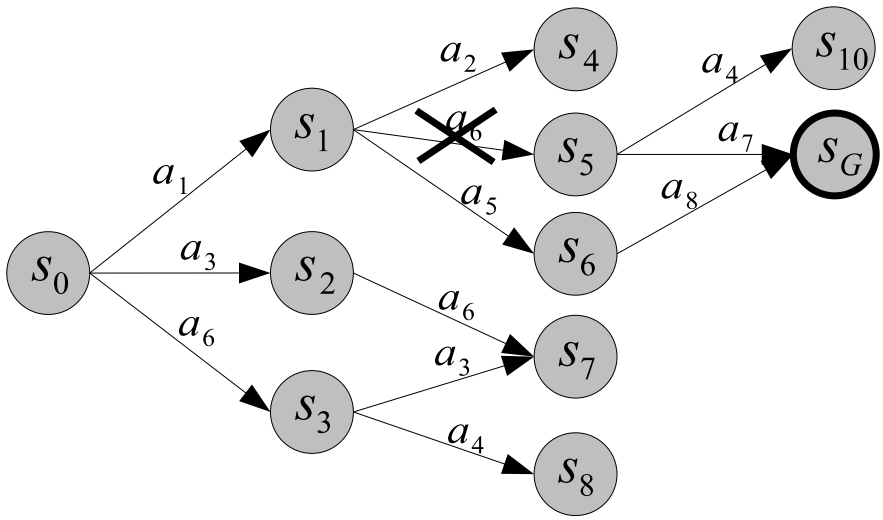


Figure 5.6: Nodes represent states, edges represent actions. A transition graph where action a_6 is removed in state s_1 and an alternative route to a goal state is found.

5.4.5 Example

We here give an example on how the loosely coupled approach works and how it can handle situations when there are actions for which there are no configurations.

In the domain for this example we have functionalities that define different sensors and actuators such as cameras, lasers, odometers, wheel motors, grippers and there are also functionalities for navigation, localization, object localization, object tracking, and gripper control. The domain also defines methods for how these can be combined and a set of actions for robot navigation and for gripper control.

The initial state is based on the map in Figure 5.7 for a small apartment. Further, the state defines that there are two robots available; Pippi and Astrid, and a stationary computer called the Home Security Monitor. It also specifies which functionalities are available on the different robots and on the computer.

Both robots are equipped with functionalities for navigation and low-level control of sensors and actuators. They both have standard sensors such as sonars, bumpers and wheel encoders and actuators for the wheels and for a gripper. The two robots are identical except that Astrid is equipped with a laser range finder and Pippi is not.

The stationary computer HSM is connected to a set of web-cameras mounted in the ceiling. HSM have a functionality that is able to track a robot and localize it in the apartment. The computer also has an action planner and a configuration planner, and the reconfigurations of the distributed robot system in these experiments are done from here. Note however that these could as well be done elsewhere, e.g., in Pippi. Consider the following scenario:

a. At start up, Pippi is located in the living-room and Astrid in the kitchen. When the morning paper arrives, the HSM wants to wake up Johanna[4], who is sleeping in the bedroom, and give it to her.

b. With this task and the initial state, an action plan is generated (step 1 Fig. 5.2). This plan has the actions: (dock-to Pippi entrance), (take Pippi newspaper), (move-to Pippi bedroom), (dock-to Pippi bed), (wake-up Pippi Johanna).

c. In step 2 (Fig. 5.2), the search for a configuration for each action is started. For the first three actions, configurations are found. The first and third action ((dock-to Pippi entrance) and (move-to Pippi bedroom)) uses the cameras mounted in the ceiling for localization. For the fourth action (dock-to Pippi bed), the search fails since no configuration can be found. The ceiling cameras used in the other actions can only track robots in the living-room and kitchen, and Pippi has no other means of localization.

[4]Johanna is the person who lives in the apartment

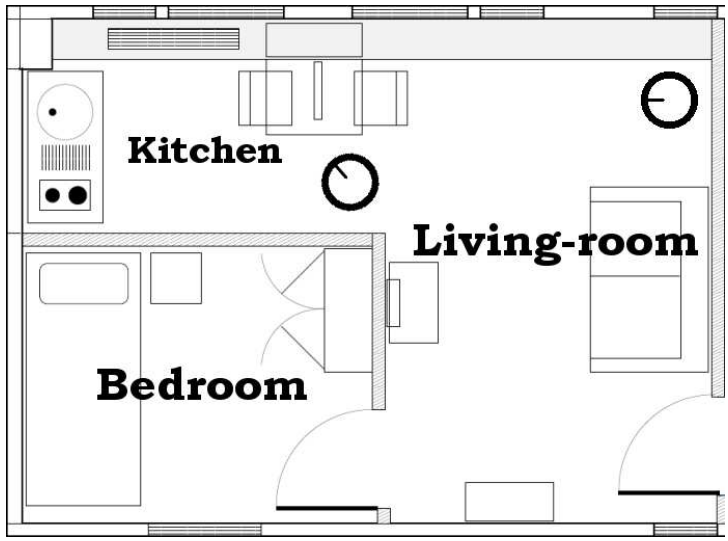


Figure 5.7: A map of the apartment. The black circles represent the robots. Pippi to the left and Astrid to the right.

The information about the failed action is stored (step A Fig 5.2), and the action planer is again called.

d. The action planner finds an alternative plan with the following actions: (move-to Astrid living-room), (dock-to Astrid entrance), (take Astrid newspaper), (move-to Astrid bedroom), (dock-to Astrid bed), (wake-up Astrid Johanna). When revisiting step 2 with the new action plan, it is possible to find a configuration for each action. Unlike Pippi, Astrid is able to localize on its own using a laser range finder and scan matching techniques.

We leave the example without explaining the execution of the configuration plan. The critical point of this example is step c where a configuration cannot be found for action (dock-to Pippi bed) and the action planner is again called to find a new action plan. In the approach with independent action planning and configuration planning, Pippi would have started to execute the action plan and would not discover that it cannot achieve the goal until it reached the bedroom. The HSM would try to find a new plan to reach the goal. The HSM cannot simply generate the same plan as above, where Astrid gets the newspaper at the entrance and delivers it to Johanna, since Pippi is now holding the newspaper in the bedroom. If there is an action for giving a newspaper between robots, the HSM may find an alternative plan, otherwise it will fail.

5.4.6 Formal Properties

The loosely coupled approach presented above is used to verify that there is a configuration plan for a given task. Of course it is desirable that the above approach finds a configuration plan if there is one and that the generated configuration plan is admissible and is able to achieve the goal G. That is, it is important that the approach is complete and sound. We can show that the approach is complete for a maximum number of steps of the sequence of configurations and for a maximum number of functionalities allowed for each configuration for each step. Note that here we mean that the approach is complete in the sense that if there is an action plan and a configuration plan for that action plan, then it will be found.

In order to prove soundness, we need to show that the action to configuration plan algorithm (Algorithm 5.1) only generates solutions to the action to configuration plan problem defined by $\mathcal{ACP}(A_{seq}, D, s, G)$ (See Section 5.4.3). That is, all configuration plans generated by the algorithm must be admissible and they must achieve goal G. A configuration plan is admissible if and only if each c_i is admissible in the state s_{i-1} it will be executed in where $s_0 = s$. From the definition of an admissible configuration on Page 27, we have that a configuration is admissible only if it is information, causal, and resource admissible defined in Equation 3.3–3.5. We state the following:

Lemma 5.1. *All the configurations in the configuration plans generated by the action to configuration plan algorithm (Algorithm 5.1) are information admissible if: the domain* D *only contains information admissible methods, for each* $a \in A_{seq}$, $G = \{m_0, \ldots, m_k\}$ *(a set of methods), and* $\forall m \in G : I_m = \emptyset$, *that is each method* m *in* G *does not need any external input.*

Proof. Follows directly from the use of the configuration generation algorithm (Algorithm 4.2) and Theorem 4.1 that states that all configurations generated by the configuration generation algorithm are information admissible if the domain D only contains information admissible methods, $G = \{m_0, \ldots, m_k\}$ (a set of methods), and $\forall m \in G : I_m = \emptyset$, that is each method m in G does not need any external input. □

Lemma 5.2. *Under the condition that all changes to the state are known, all the configurations in the configuration plans generated by the action to configuration plan algorithm (Algorithm 5.1) are causally admissible.*

Proof. Assume the contrary, that there exist a configuration in a configuration plan generated by the action to configuration plan algorithm that is causally inadmissible. We argue that this case can never occur.

There are two cases that can lead to causally inadmissible configurations. The first case is if the configuration generation algorithm can return inadmissible configurations for a state s. Theorem 4.1 states that this can never occur since the configuration generation algorithm only returns causally admissible

configurations. The second case is if the state used to generate configurations is not updated correctly. This can never occur since in the third step of Algorithm 5.1 the state is updated according to what is defined by the postconditions of the configurations. Thus, all the configurations in the configuration plans generated by the action to configuration plan algorithm are causally admissible. □

Lemma 5.3. *Under the condition that all resources of different types in state* s *are available (*$\mathtt{tn}_{avail} = \mathtt{tn}_{max}$*), all the configurations in the configuration plans generated by the action to configuration plan algorithm (Algorithm 5.1) are resource admissible.*

Proof. Follows directly from Theorem 4.1 that the configuration generation algorithm only generates resource admissible configurations. □

Lemma 5.4. *The action to configuration plan algorithm (Algorithm 5.1) only generates configuration plans that achieve goal* G.

Proof. Assume the contrary, that the algorithm generates configuration plans that does not achieve goal G. We argue that this case can never occur.

In step 4 of Algorithm 5.1 it is verified that the configuration plans returned achieves the goal G. All other configuration plans are discarded. Thus, we have the thesis. □

Lemma 5.5. *Under the conditions mentioned in Lemma 5.1, Lemma 5.2, and Lemma 5.3 the action to configuration plan algorithm (Algorithm 5.1) only generates admissible configuration plans that achieve goal* G.

Proof. Follows directly from Lemma 5.1, Lemma 5.2, and Lemma 5.3 that guarantee that all configuration plans generated by the action to configuration plan algorithm are information, causal, and resource admissible if D only contains information admissible methods, and from Lemma 5.4 that guarantee that all configuration plans generated by the action to configuration plan algorithm achieve goal G. □

From Theorem 4.2 we have that the configuration algorithm is complete, i.e., the configuration algorithm eventually generates any admissible configuration that is defined by the functionalities and methods in D. Since the configuration is able to find any configuration defined by the functionalities and methods in D, it is also guaranteed to find all configurations with a maximum number of functionalities n, for any number of functionalities n. Recall that from Section 4.3.2 that we do allow recursive methods as long as they are not front recursive, i.e., expanding the method implies adding of functionalities. By constraining the configuration algorithm on the number of functionalities, the configuration algorithm is guaranteed to terminate, which is necessary if one is to find configurations for all actions i A_{seq}. If one then stepwise increases the maximal number of functionalities, one can eventually generate all configuration plans for A_{seq}.

Lemma 5.6. *For any number* n*, the action plan to configuration plan algorithm (Algorithm 5.1) eventually generates all admissible configuration plans in* $\mathcal{ACP}(A_{seq}, D, s, G)$ *with at most* n *functionalities for each configuration.*

Proof. Follows directly from the completeness of the configuration algorithm (Algorithm 4.2) for any number of functionalities n. By Lemma 5.5, the generated configuration plans are admissible and achieve G. □

We are now in a position to prove soundness and completeness for our configuration plan algorithm. We assume that the domain D is finite.

Theorem 5.7 (Soundness)**.** *Under the conditions mentioned in Lemma 5.1, Lemma 5.2, and Lemma 5.3 the algorithm in Figure 5.2 that implements the loosely coupled approach only generates admissible configuration plans that achieve goal* G.

Proof. Follows directly from the soundness of the action planner [Karlsson and Bouguerra, 2009] and from the soundness of the action plan to configuration plan problem (Lemma 5.5). □

The loosely coupled approach addresses the configuration plan problem by decomposing it into an action plan problem and an action plan to configuration plan problem. In order prove that the full approach is complete, the algorithms for the individual solutions must be complete. The formal details of the action planner are not given in this thesis, however in [Karlsson and Bouguerra, 2009] it is formally proved that PTLplan is complete. Above we discussed why it is necessary to restrict the number of functionalities n in configurations generated at each step. In a similar way one can also restrict the number of steps of an action plan m. By systematically running the planner with different values for m and n (e.g., by stepwise increasing the sum $m + n$) one can eventually find all configuration plans.

Theorem 5.8 (Completeness)**.** *For any* m *and* n*, the loosely coupled action and configuration plan approach eventually generates all action plans that achieve goal* G *with at most* m *actions and for each action plan generates all admissible configuration plans defined by the actions, functionalities and methods in* D*, where each configuration has at most* n *functionalities.*

Proof. Follows directly from the completeness of the action planner (PTLplan) and the completeness of the action plan to configuration plan algorithm (given in Lemma 5.6). □

Note that Theorem 5.8 does not claim that our integrated planner will find all configuration plans, but only those for which there is an intermediary action plan.

5.5 Top-Level Process

In order to execute a configuration plan and to use it in a real distributed robot system, the approach must be embedded in a larger process in a similar way as it

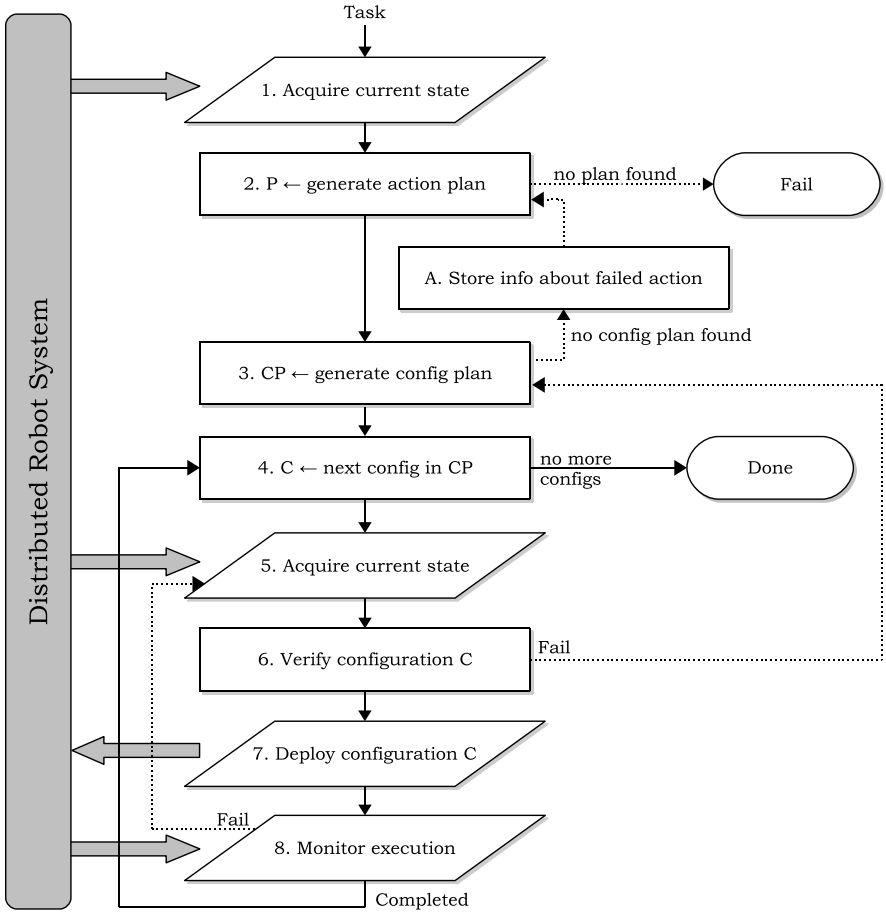


Figure 5.8: Flow chart of the top-level process.

was done for the single configuration approach in Section 4.9. Figure 5.8 shows the top-level process for sequences of configurations. The upper part of the figure (steps 1, 2, 3, A) performs the planning phase of the top-level process, and the lower part (steps 4 – 8) the execution phase. The loosely coupled action and configuration planning approach and its top-level process have been implemented and tested in a real distributed robot system called the PEIS-Ecology (See Section 7.4).

5.5.1 State Acquisition

The current state of the environment and the distributed robot system is acquired in steps 1 and 5 of the top-level process. In step 1, the state is acquired before the action and configuration planning starts. Both planners use the same type of state to ensure that only admissible action plans and configuration plans are generated. In step 5, the state is acquired before the execution of a configuration in distributed robot system to ensure that the configuration is (still) admissible. Section 4.3.3 describes the representation of the state.

5.5.2 Configuration Plan Loop

The steps (2, 3, A) in Figure 5.8 constitute the configuration plan loop described in Section 5.4 above. This loop continues until an admissible configuration plan is found, or until the action planner is unable to find an action plan.

5.5.3 Configuration Plan Sequencer

When a configuration plan is found, it is given to a sequencer (step 4 in Figure 5.8) that is responsible for taking the next configuration in the configuration plan. When an action/configuration is reported to be completed (step 8), the sequencer takes the next configuration in the plan to deploy.

5.5.4 Verify a Configuration

As with the execution of a traditional action plan, it is important to verify before execution of an action that it is executable in the current state. Hence, the configurations in the configuration plan are generated based on the states calculated from state-transitions depending on the postconditions of the actions and the postconditions of the configurations. During the generation of the configurations, all the resources of different types are assumed to be available for all states succeeding the initial state. For example, assume that there are 3 resources available of type q and 5 resources in total of type q. When the configurations are generated for the third action in A_{seq}, the configuration algorithm will assume that all 5 resources of type q are available.

In the verification of a configuration (step 6), the preconditions Pr_c and the availability of the resources of different types $\mathrm{Re}_c(\mathrm{tn})$ are checked against the current state for execution. Thus, when it is time to execute the third action in the example above, we must verify the number of resources of type q actually used are lower than the real number of available resources of type q (not necessary 3 anymore). If the configuration is (still) admissible, it can be deployed in the distributed robot system (step 7). Of course it may be the case that the configuration is not admissible anymore due to some external contingency. Algorithm 5.3 details the behavior of the top-level process when the verification discovers that a configuration is inadmissible. The step references in the algorithm are to the steps in Figure 5.8.

Algorithm 5.3 The behavior of the top-level process after verification failure

```
1:  if Preconditions of c does not hold then
2:      Try to generate an alternative configuration c′(step 3).
3:      if c′ is found then
4:          Compare the postconditions of c and c′.
5:          if Postconditions are equal then
6:              Replace c with c′ in the configuration plan.
7:          else
8:              Replace c with c′ in the configuration plan
                and update the configuration plan.
9:          end if
10:         Continue execution (step 4).
11:     else
12:         Store info about failed action (step A)
13:         Action planner tries to find a new action plan
            from the current state(step 2).
14:     end if
15: else if Not all resources required by c are available then
16:     Try to generate an alternative configuration c′(step 3)
            as above (lines 2 – 14).
17: end if
```

Some of the actions in Algorithm 5.3 may need some further explanation. On line 4, when an alternative configuration is found, the postconditions of the inadmissible configuration (the initial one) are compared to the postconditions of the new alternative configuration (line 5). If they are equal, the configuration can safely be added to the configuration plan (line 6). If they differ, the remaining part of the configuration plan must be regenerated to comply with the new configuration and its postconditions, before the execution can continue (line 8). When the execution of the configuration plan continues (line 10) the sequencer (step 4) retries the same action instead of taking the next one as it

usually does. Lines 11 – 13 take care of the case when no alternative configuration was found. The information about this action is stored in step A and the action planner tries to find a new action plan (step 2) as described in the previous section. The lower part of the algorithm (lines 15 – 17) is concerned with when the verification identifies the configuration as resource inadmissible. In this case, an alternative configuration is generated in the same way as described on lines 2 – 14. Instead of directly generating an alternative configuration one could wait for the missing resources to become available. However, this might be tricky and outside the scope of this thesis.

5.5.5 Deployment of a Configuration

When it is verified that a configuration is admissible in the current state, it must be deployed on the distributed robot system (step 7) as described in Section 4.9.3.

5.5.6 Configuration Execution and Monitoring

After a configuration has been deployed, the execution and the monitoring of the configuration starts (step 8 Fig. 5.8) in a similar way as described in Section 4.9.4. This continues until the action is completed or it fails. When a configuration is completed, the next one is selected (step 4). If a configuration fails during execution, the top process tries to generate an alternative configuration as described in Section 5.5.4.

Chapter 6
Parallel Actions and Configurations

In the previous chapter we described how we can generate and execute sequences of configurations. However, in many cases it is desirable to execute more than one configuration at a time. For instance, there is no real reason why a configuration for opening a door and a configuration to turn on the light could not be executed at the same time, as long as the door opening configuration does not require that the light is on or vice versa.

In this chapter we discuss why and when parallel configurations can be used (Section 6.1). Further, we describe how to merge configurations (Section 6.2) and how to reduce merged configurations by removing parts that are no longer needed (Section 6.3). In Section 6.4, we describe how configuration merging and reduction can be used in the work of parallelizing a configuration plan. In order to execute configurations in parallel in a real distributed robot system, the top-level process given in Section 5.5 needs to be modified. The new top-level process is described in Section 6.5.

6.1 Why/When to Use Parallel Configurations

There are several cases when it may be desired to execute several configurations in parallel:

1. in order to reduce total execution time of an action plan;
2. to enable the possibility to assign new tasks to a single configuration process while other tasks are being executed;
3. when there are several configuration processes in the same distributed robot system, and it is possible to assign different tasks to different configurators concurrently.

The first case considers an action plan as described in the previous section. An action plan describes a total ordering of actions. However, it is not necessarily the preconditions and postconditions (effects) of the actions that impose a total order. Actually, the total ordering is often an artifact of the way the planner works. Thus to use a partial ordered planner or to parallelize a total ordered plan may offer opportunities to reduce the total plan execution time. In addition, one can give the planner a conjunctive goal, where each conjunct can be pursued independently. In such cases, it might be more efficient to run configurations in parallel. Thus, to request more than one task is possible by having a conjunctive goal formula, as long as they are requested at the same time and are not leading to a causal conflict. The possibility to run several tasks/actions at the same time requires that it is possible to run several configurations at the same time.

The second case is concerned with the possibility to request a new task when one or more tasks have already been requested. This possibility allows a distributed robot system to execute several tasks simultaneously without having to wait for one task to finish before the next can be started. With a single configurator in the distributed robot system, this problem can be handled with several processes sharing the same memory/knowledge. By sharing the same memory/knowledge it is easier to avoid resource conflicts and to run configurations in parallel. However, it is still necessary to verify that the configurations can run in parallel.

The third case is when it is possible to have several configurators active at the same time in the same distributed robot system. In such a system, several tasks might be performed concurrently, and new tasks might dynamically appear. If tasks are assigned to the system, task allocation techniques can be used to assign different tasks to different configuration processes. With an assigned task, each individual configuration process must know which functionalities, resources etc are available for it to use. Each configurator must know if it is possible to run its configurations in parallel with the configurations of the other configuration processes. This is quite a complex problem.

All three cases have a key problem in common: How to guarantee that only configurations that can run concurrently are allowed to do so. This is an easy task if the configurations are clearly disjoint (i.e., they do not use the same robots, functionalities or resources). However this is not always the case. We propose a method to merge configurations that guarantees that when two admissible configurations are merged, the resulting configuration is also admissible. The merging process may also have positive effects on the total configuration cost, if two configurations may share functionalities. In this way the total cost can actually be lower than the sum of the costs of the involved configurations. This merging process can be used for the different cases above.

A merged configuration is a composition of several configurations, and if one of these configurations is finished or fails, it is desirable to *reduce* the merged configuration, that is, to remove unused parts. In this way it is not nec-

essary to wait for all configurations in the merged configuration to finish before the next step is considered, and it is not necessary to replace or regenerate the merged configuration if individual configurations fail.

6.2 Merging Configurations

As mentioned above, merging of configurations is a way to verify that two configurations can run concurrently without conflict. If it is possible to run the configurations concurrently, a single *merged* configuration is created. This merged configuration can then be merged with yet another configuration, creating a new merged configuration, and so on. In this way it is possible to verify if several configurations can be run concurrently.

The resulting merged configuration is an executable configuration that is optimized for concurrent execution of the configurations it includes. That is, during the merging, the different parts of the configurations are added to the new merged configuration. Some parts may be redundant and not included which results in redirected channels and/or removed functionalities. For instance, if a functionality that needs a specific resource appears in both configurations and the functionality is used for the same purpose in both configurations (i.e., the functionalities are identical), there will only be one instance of this functionality in the merged configuration, and this functionality and its resources are "shared" by the individual configurations. This type of sharing is only possible when the merged configuration is executed — individual execution of the configurations in parallel is not possible.

To enable reduction of merged configurations it is necessary to know which parts belong to which individual configuration (i.e., action). This can be done by marking all functionalities, resources, preconditions and postconditions with the actions they belong to, before merging the configurations. Since configurations may share entities, it is possible for an entity in a merged configuration to belong to several actions at the same time.

6.2.1 Configuration Merge Problem

A *configuration merge problem* is a tuple $\langle c_i, c_j, s\rangle$ where $c_i = \langle F_i, Ch_i\rangle$ and $c_j = \langle F_j, Ch_j\rangle$ are configurations admissible in state s. A merged configuration $c_m = \langle F, Ch\rangle$, where $F \subseteq F_i \cup F_j$, and F includes all terminating functionalities in $F_i \cup F_j$, is a solution to the configuration merge problem if c_m is admissible in state s. We let $\|$ denote the operation of merging two configurations and in subscription the origin of the merged configuration, i.e., $c_1 \| c_2 \equiv c_{1\|2}$.

6.2.2 Algorithm

The merging process takes the descriptions of two admissible configurations (c_1, c_2) to merge and a state s. If merging is possible it returns the description

of a single merged configuration. It is important that each one of the two configurations are individually admissible in state s.

Algorithm 6.1 describes how this merge is performed. The first part of the algorithm (lines 2 - 7) checks if one of the two configurations is a nil-configuration ($c^{nil} = \langle \emptyset, \emptyset \rangle$). When a non-empty configuration is combined with a nil-configuration the merged combination is composed only of the non-empty configuration: e.g., $c_{11} \parallel c^{nil}$ is equivalent to c_{11}. In Section 6.4.2 we describe how the nil-configuration is used in the process of finding possible configuration combinations for concurrent action execution.

The actual merging process starts on line 8 were c_1 is assigned to c^m (c_1 is used as the basis for the merged configuration). On line 9, the state s is updated by removing all the resources used by configuration c_1. The postconditions, channels, and preconditions of the two configurations are merged on lines 10 - 16. Before the postconditions can be merged, it is necessary to verify that there is no conflict between the postconditions as defined in Equation 3.5. If there is a conflict, the merge fails (line 13). The channels is a union of the channels of c_1 and c_2, and the preconditions is a conjunction of the preconditions of c_1 and c_2. The functionalities of the configurations are merged by simply adding the functionalities of c_2 one by one to the functionalities of c^m (lines 17 - 31). If the functionality f to add already exists in the merged configuration (line 19), the functionality is not added, but the channels that is concerned with the functionality is updated and redirected to point to the already present functionality (line 21). The reconnection of channels in Ch_{c^m} from a given f to $f' \in F_{c^m}$ works as follows. If $\langle f, o, f'', i \rangle$ then this is changed to $\langle f', o, f'', i \rangle$. If $\langle f'', o, f, i \rangle$ then this is changed to $\langle f'', o, f', i \rangle$.

On line 23, if f is a new functionality, it is added to the merged configuration as long as the required resources exist. The resources of the functionality are also added to the total resources used by the merged configuration (line 25). The resources used are subtracted from the state s (line 26) each time a functionality is added. If there are not enough resources, the merge fails (line 28).

Finally, the redundancies are removed from the merged configuration (line 32) and the cost of the configuration is recalculated before it is returned (line 33). The Remove-Redundancies function cleans up the merged configuration such that no functionalities can have more than one channel connected to each input. The redundant parts can be removed with the following steps:

1. For each input of some functionality that has more than one channel connected to it, nondeterministically decide what channel to keep and remove the others from the functionality.

Algorithm 6.1 A procedure for merging configurations

Require: Two configurations c_1, c_2, and a state s
Ensure: A single merged configuration c^m

```
1:  procedure MERGE-CONFIGURATIONS(c1, c2, s)
2:      if c1 = c^nil then
3:          return c2
4:      end if
5:      if c2 = c^nil then
6:          return c1
7:      end if
8:      c^m ← c1                                   ▷ Let c1 be the base for c^m
9:      s ← SUBTRACT-RESOURCES(Re_{c^m}, s)
10:     if COMPATIBLE-POSTCONDITIONS(c^m, c2, s) then
11:         Po_{c^m} ← Po_{c^m} ∘ Po_{c2}          ▷ Add the postconditions of c2
12:     else
13:         return FAIL
14:     end if
15:     Ch_{c^m} ← Ch_{c^m} ∪ Ch_{c2}              ▷ Add the channels of c2
16:     Pr_{c^m} ← Pr_{c^m} ∧ Pr_{c2}              ▷ Add the preconditions of c2
17:     while F_{c2} ≠ ∅ do                        ▷ Loop until there are no functionalities
18:         f ← POP(F_{c2})                        ▷ f is the first funct. in F_{c2}
19:         f' ← FIND-FUNCTIONALITY(F_{c^m}, f)
20:         if f' ≠ FAIL then                      ▷ Is f already in F_{c^m}
21:             UPDATE-AND-REDIRECT-CHANNELS(Ch_{c^m}, f, f')
22:         else
23:             if RESOURCES-AVAIL(Re_f, s) then   ▷ are there resources for f
24:                 F_{c^m} ← F_{c^m} ∪ {f}
25:                 Re_{c^m} ← Re_{c^m} + Re_f
26:                 s ← SUBTRACT-RESOURCES(Re_f, s)
27:             else
28:                 return FAIL
29:             end if
30:         end if
31:     end while
32:     c^m ← REMOVE-REDUNDANCIES(c^m)
33:     Cost_{c^m} ← CALCULATE-COST(c^m)
34:     return c^m
35: end procedure
```

2. For each functionality which had input channels removed in step 1, look at what configurations it is marked to belong to, and add this marking to any other functionality that is upstream[1].

3. Remove each functionality (and its input channels) that had a channel removed from its output, and which neither (a) has any other output channel nor (b) is an actuator or terminating functionality.

4. Repeat step 3 until no more functionalities can be removed.

6.2.3 Illustrative Example

To illustrate the merging process we consider an example in which two robots (Pippi and Astrid) clean an apartment. The apartment consists of three rooms: kitchen, bedroom and living-room (See Figure 7.3 on Page 124). Pippi is about to clean the bedroom and Astrid is about to clean the kitchen. During the vacuum cleaning operation, the robots want to know the position of the person who lives in the apartment so that the robots can avoid to disturb him/her, e.g., if the person enters the room where a robot is vacuum cleaning, the robot should decrease the vacuum effect and cleaning speed. In Figure 6.1 the configuration for Pippi is shown. The configuration for Astrid is similar but functionalities on Pippi are on Astrid instead. Both configurations use the person-tracker component to know the current position of the person in the apartment (See operator in Figure 6.3).

The configurations are merged in order to test whether they can be executed concurrently. The merging is performed as follows:

1. The configuration for Pippi is used as the basis for the merged configuration (line 8 in Algorithm 6.1).

2. The postconditions of the two configurations are compatible which means that the postcondition of the merged configuration can safely be set to the composition of the postconditions of the two configurations. The preconditions and channels are also merged.

3. The functionalities of the configuration for Astrid are then added one by one to the merged configuration by first testing that it is not already in the merged configuration. In this example it is only the person-tracker functionality and the ceiling cameras that are already in the merged configuration, i.e., there are functionalities with the same instantiation of the parameters in the name field of the operators for ceiling cameras and person-tracker. When these functionalities are considered, the channels connected to these functionalities are reconnected so that the already existing functionalities are used.

[1] A functionality x is *upstream* to another functionality y if and only if functionality x produces information directly for functionality y, or for another functionality that in turn is upstream to y

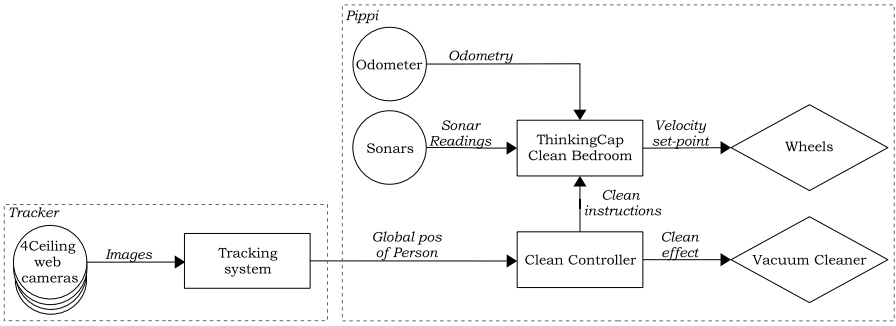


Figure 6.1: Pippi's configuration for vacuum cleaning.

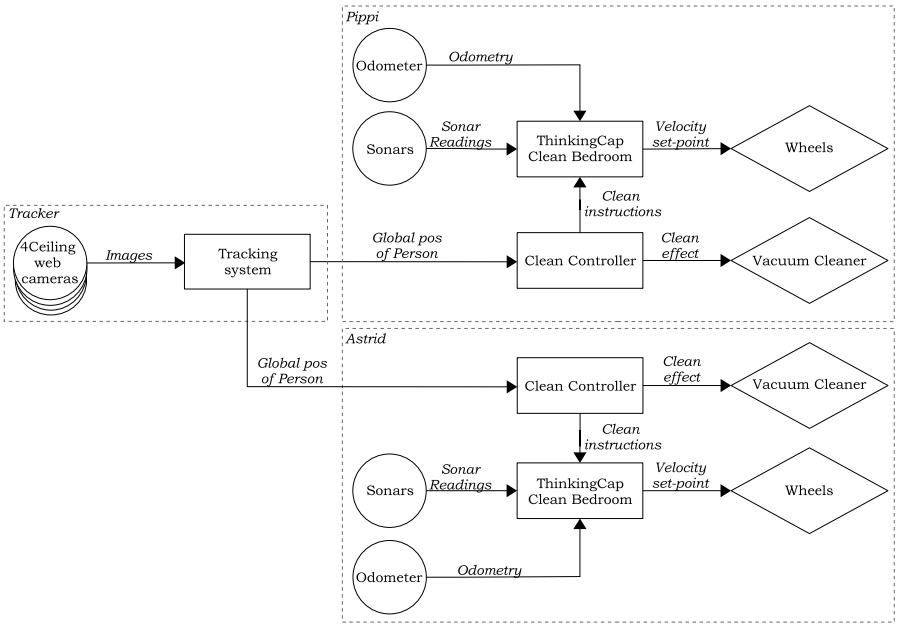


Figure 6.2: A merged configuration for Pippi and Astrid in which they both are cleaning and gets information about the person that lives in the apartment.

```
(functionality
 :name (persontracker ?p ?id ?r)
 :precond ( ((?p) (robotic-device ?p))
            ((?p2) (robotic-device ?p2))
            ((?p3) (robotic-device ?p3))
            ((?p3) (robotic-device ?p4))
            ((?p5) (robotic-device ?p5))
            ((?id) (funct ?p persontracker ?id))
            ((?r) (or (robot ?r) (person ?r))) )
 :resource ((re ?p ?id))
 :in  ( ((image ?p2) image)
        ((image ?p3) image)
        ((image ?p4) image)
        ((image ?p5) image) )
 :out ( ((pos ?r) real-coordinates) )
 :cost 10)
```

Figure 6.3: The functionality operator for the person tracker.

4. Before the merged configuration is returned, the algorithm checks and removes possible redundancies (in this case there are none) and recalculates the cost.

In Figure 6.2, the resulting merged configuration is shown. In this configuration both robots can share the functionalities for tracking the person in the apartment.

In the above example, merging the two configurations was possible. Consider instead the case when the person-tracker is used in both individual configuration to track the position of the robots for localization and not to track the person. That is, Pippi wants to know her own position from the tracker and Astrid wants to know her own position from the tracker. In this case the configurations could not be merged since the person-tracker cannot be used to track two different objects, i.e., there are two different instantiations of the person-tracker: one in which ?r is bound to Pippi and one in which ?r is bound to Astrid. The configurations must be executed one at a time, or one of the robots must find an alternative way to localize.

6.2.4 Maintaining Configuration Admissibility

The configuration algorithm described in Section 6.2.2 is guaranteed to only generate admissible configurations. This is captured in the following lemmas and theorem.

Lemma 6.1. *Given that* c_i *and* c_j *are information admissible in state* s*, Algorithm 6.1 only returns a merged configuration if it is also information admissible in state* s.

Proof. We will give this proofs in two steps: (1) proving that the merging of the configurations c_i and c_j (lines 1 – 31 in Algorithm 6.1) results in an information admissible configuration, and (2) showing that the Remove-Redundancies function preserves the information admissibility. Recall the definition of information admissibility in Eq. 3.3: A configuration is information admissible if and only if each input of each functionality is connected via an adequate channel ($ch = (f_{send}, o, f, i) \in Ch$) to an output of another functionality f_{send} with a compatible specification ($desc(o) = desc(i)$ and $dom(o) = dom(i)$). Since both c_i and c_j are information admissible, all their functionalities are properly connected via adequate channels, that is the functionalities are information admissible. A functionality can only become information inadmissible if an original channel connected to its input is removed and *not* replaced by another channel to some proper output. But the latter can never occur. Thus for (1), the merged configuration is information admissible since the algorithm does not disconnect or remove any of channels necessary for admissibility.

The function Remove-Redundancies removes channels in two cases. The first case is when there exists a functionality input which has more than one channel (step 1) connected to it. The second case is when a functionality is removed and the channels connected to its inputs are also removed (step 3). Neither of the two cases remove channels that are necessary for information admissibility. For the first case, a channel that provides the required input is always kept. For the second case, a functionality (and its input channels) is only removed if it has no output channels and is not an actuator or terminating functionality. Thus, their removal can never make another functionality inadmissible. Consequently, for (2) the function Remove-Redundancies preserves the information admissibility of the merged configuration.

Since both (1) and (2) guarantees information admissibility of the merged configuration, we have the thesis. □

Lemma 6.2. *Given that* c_i *and* c_j *are causally admissible in state* s*, Algorithm 6.1 only returns a merged configuration* c *if* c *is causally admissible in state* s.

Proof. From the definition of causal admissibility in Eq. 3.4, we have that a configuration is causally admissible if and only if all functionalities in the configuration are applicable in the world state, i.e., $Pr_c(s) = T$ and $\neg\exists po_i, po_j \in Components(Po_c) : Conflict(po_i, po_j, s)$. All the preconditions of the merged configuration hold since they are a conjunction of (a subset of) the preconditions of the admissible configurations c_i and c_j. All the postconditions of the merged configuration hold since the postconditions of c_i and c_j are compared for compatibility on line 10. Since all functionalities in the merged configuration are applicable, we have the thesis. □

Lemma 6.3. *Algorithm 6.1 only returns a configuration if it is resource admissible in state* s.

Proof. From the definition of resource admissibility (Equation 3.5) we have that a configuration is resource admissible if there are enough resources available in state s to meet the requirements of the configuration. In Algorithm 6.1, functionalities are added to the merged configuration in two places: on line 8 and 23. In order to make sure that all resources are available, the state s must be updated each time a functionality is added (line 9 and 25). On line 8, all functionalities of configuration c_i are added to c_m. Since we know that c_i is resource admissible, the resources for all functionalities of c_i are available. On line 23, the functionalities of c_j are added to c_m one by one. A functionality is only added if its resources are available in state s (line 22). Since functionalities are only added if their resources are available in s, the merged configuration must be resource admissible. □

Lemma 6.4. *Algorithm 6.1 only returns a merged configuration* c *for* c_i *and* c_j *if the set of functionalities for* c *(i.e,* $F_c \subseteq F_i \cup F_j$*) includes all terminating functionalities in* $F_{c_i} \cup F_{c_j}$.

Proof. There is only one place in Algorithm 6.1 in which functionalities are removed from the merged configuration. This is in step 3 of function RemoveRedundancies. However, in this step functionalities are only removed if they are not terminating functionalities. Since no terminating functionalities can be removed, we have the thesis. □

We are now in a position to prove soundness of our configuration merging algorithm.

Theorem 6.5 (Soundness). *Given that configurations* c_i *and* c_j *are admissible in state* s*, Algorithm 6.1 only returns a configuration* c *if* c *is admissible in state* s *and contains all the terminating functionalities of* c_i *and* c_j.

Proof. Follows directly from Lemma 6.1 – Lemma 6.4 that guarantee that the returned merged configuration c is information, causal, and resource admissible, and that all terminating functionalities of c_i and c_j are included in c. □

Although Algorithm 6.1 is sound, it is not possible to show that the algorithm is complete. That is, there are cases in which the algorithm does not find a merged configuration even though a solution exist. This is best illustrated by an example. Assume we have two very simple configurations c_1 and c_2 with two functionalities each (See Figure 6.4). In c_1, functionality f_a produces information of type i that is used by functionality f_b. Similarly in c_2, functionality f_c produces information of type i that is used by functionality f_d. Note that both f_a and f_c produce the same type of information but are not identical. In a similar way are f_b and f_d different, they only share the same type of information required as input. If these two configuration are merged using Algorithm 6.1, the resulting configuration would still be composed of the same

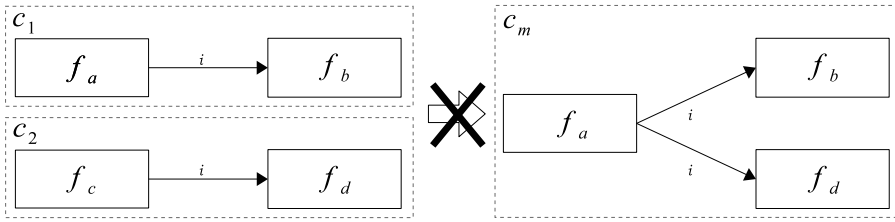


Figure 6.4: Left: Two configurations to be merged. Right: A merged configuration that cannot be found by Algorithm 6.1.

four functionalities connected in the same way. This is a valid solution, but not very efficient. It would be better if either f_a or f_c was removed and the remaining functionality would produce i to both f_b and f_d (The configuration to the right in Figure 6.4). Furthermore, if both f_a and f_c require a non-sharable, or exclusive, resource, then Algorithm 6.1 would not find a solution at all.

6.3 Reducing Configurations

The idea of reducing a merged configuration is to remove all parts that belong to a single subconfiguration that is not used anymore, without affecting the admissibility of the remaining configuration. To be able to reduce configurations is important for efficient execution. For instance, assume we have a merged configuration that consists of three different individual configurations (c_1, c_2, c_3) where c_1 is a configuration for opening a door, c_2 for moving a robot, and c_3 for turning on the light. All of these configurations will probably *not* finish simultaneously, and we do not want the configuration process(es) to wait for all of them to finish before it performs the next step. It may also be the case that some part of configuration c_1 fails, but all functionalities of configurations c_2 and c_3 perform well. In this case it is not desirable to reconfigure all configurations, but rather just the part that is concerned with c_1. Of course, it may be the case that some configurations of a merged configuration share the same functionalities. If it is a shared functionality that fails, then all the configurations that include this shared functionality must be reconfigured/replaced.

The reduction algorithm takes as input a merged configuration c^m and an action a representing the individual configuration to remove. Recall that all entities of a merged configuration (e.g., functionalities, resources) are marked with the actions they belong to (See Section 6.2). The output is a merged configuration $c^{m\prime}$ where all parts related to a are removed. The reduction is performed in four steps:

1. Remove all α-labels from all entities (functionalities, resources, preconditions, postconditions).
2. Go through the list of functionalities and for each functionality that does not belong to any action, remove all channels that are connected to either its output or input.
3. Remove all entities that do not belong to any action.
4. Recalculate the configuration cost.

Since the reduction of the merged configuration just removes parts that only belongs to action α, the resulting merged configuration is obviously still admissible in the state it was created.

6.4 Parallel Actions in an Action Plan

In the beginning of this chapter, parallel configurations and merging of configurations for different cases were discussed. In this section a more detailed description of one of these cases will be given — when the merging process is used to parallelize an action plan in order to reduce total execution time.

In the approach we have selected, parallelization is performed after the action planning and configuration planning steps (after steps 1–3 in Figure 5.5). Another solution could be to directly generate a partially ordered plan [Barrett et al., 1994, Weld, 1994]. However, to parallelize the configuration plan, not taking into account an optimal parallelization, is a straight forward procedure [Bäckström, 1998]. Also, to replace PTLplan with a partial-order planner would require to restructure the approach presented in Chapter 5. For pragmatic reasons, we have decided to do the parallelization after plan generation, while the plan is executed (around step 4 in Figure 5.5). It is done in the following two situations:

- When no configuration is running (e.g., at the start of plan execution). In this case, the first action α in the action plan that is not already finished will be fetched and the remaining part of the action plan will be searched for actions that are possible to run in parallel with α.
- When, in a merged configuration, one or more configurations are completed. In this case, when there is a merged configuration (for a set of actions A) still running, the remaining part of the action plan is searched for actions that can run in parallel with the actions in A.

More details on the execution of the action/configuration plan will be given in Section 6.5. This section focuses on how parallel actions/configurations are found and combined into merged configurations.

Figure 6.5 gives an overview of the approach we take to run several actions/configurations in an action/configuration plan in parallel. The first step

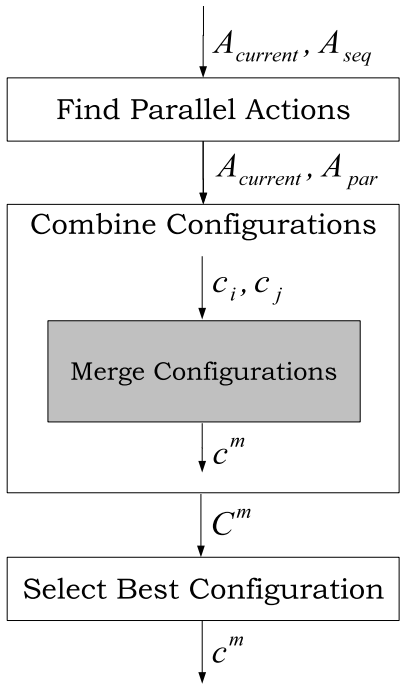


Figure 6.5: A schema for generating a set of merged configurations

is to find a set of parallel actions A_{par}, given a sequence of actions A_{seq} (i.e., an action plan) and a set of actions $A_{current}$. The set $A_{current}$ contains either the currently running actions, or the next single action that is up for execution. A_{par} contains all the actions that can run in parallel with the actions in $A_{current}$. All possible combination of these parallel actions are considered. In the procedure for combining configurations, configurations are merged two at a time, and in the end the procedure generates a set of merged configurations C^m that is able to execute the actions (or a subset of the actions) in A_{par} in parallel. The best configuration c^m in this set is selected for execution.

6.4.1 Finding Parallel Actions

The ordering of actions in an action plan is partially due to causal relationships, e.g., a_i comes before a_{i+1} ($a_i \preceq a_{i+1}$) because the postconditions of a_i are needed in the preconditions of a_{i+1}. However, often the order between two actions is purely coincidental, e.g., a_i could just as well be executed in parallel with or after a_{i+1}. When an action in the action plan is selected for execution the system searches through the remaining part of the plan to see if there is some other action that can be executed in parallel with it. Algorithm 6.2 details how the search for parallel actions is performed. This algorithm is similar to minimal-constrained de-ordering defined in [Bäckström, 1998].

The input to the procedure is a sequence of actions A_{seq} (an action plan), a set of actions $A_{current}$ for which the procedure should find parallel actions, a set of finished actions $A_{finished}$, and an initial state s_0. The output of the procedure is a set of parallel actions A_{par}. In this context, actions can be parallel if they are causally independent and the execution of these actions concurrently does not make any other action in A_{seq} inexecutable or leads to a causal conflict. $A_{finished}$ contains a set of actions that have already finished their execution. The first time FIND-PARALLEL-ACTIONS is called with a new action plan, $A_{current}$ contains only the first action in the plan and the job is to find which other actions can be executed in parallel with this first action. Next time FIND-PARALLEL-ACTIONS is called it may be the case that several actions are currently executed and the job is to find other actions that can be executed in parallel with the already running actions (i.e., $A_{current}$ is then the currently running actions). Figure 6.6 describes the relation between A_{seq}, A_{par}, $A_{current}$, and $A_{finished}$. More details about the full execution of parallel actions are given in Section 6.5.

The FIND-PARALLEL-ACTIONS procedure goes through each action a_i in the sequence of actions A_{seq} and tests if it is possible to execute it in parallel with the actions in $A_{current}$. Only unfinished actions that are not already in the set of parallel actions (line 5) are considered. The tests on line 7 verify that the preconditions of the current action a_i hold in the current state s_0, that all actions previous to the action a_i are still executable after a_i are executed, and that the postconditions of action a_i is compatible with

Algorithm 6.2 Find parallel actions for a set of actions

Require: A set of parallel actions $A_{current}$, a set of finished actions $A_{finished}$, a sequence of actions $A_{seq} = \langle a_0, \ldots, a_k \rangle$, and a current state s_0.
Ensure: A set of actions A_{par} that are parallel to $A_{current}$.

```
 1: procedure FIND-PARALLEL-ACTIONS(A_current, A_finished, A_seq, s_0)
 2:     A_par ← A_current
 3:     Po^par ← Po_{A_current}
 4:     for i ← 0, |A_seq| do
 5:         if a_i ∉ A_finished ∪ A_current then
 6:             s_after ← APPLY-POSTCONDITIONS(Po_{a_i}, s_0)
 7:             if PRECONDS-HOLD(Pr_{a_i}, s_0) and
                  PREV-PRECONDS-HOLD(A_finished, A_seq, a_i, s_after) and
                  COMPATIBLE-POSTCONDITIONS(Po^par, Po_{a_i}, s_0) then
 8:                 A_par ← A_par ∪ {a_i}
 9:                 Po^par ← Po^par ∘ Po_{a_i}
10:             end if
11:         end if
12:     end for
13:     return A_par
14: end procedure

15: procedure PREV-PRECONDS-HOLD(A_finished, A_seq, a_stop, s)
16:     while a ≠ a_stop do
17:         a ← POP(A_seq)
18:         if a ∉ A_finished and
              ¬PRECONDS-HOLD(Pr_a, s) then
19:             return false
20:         end if
21:         s ← APPLY-POSTCONDITIONS(Po_a, s)
22:     end while
23:     return true
24: end procedure
```

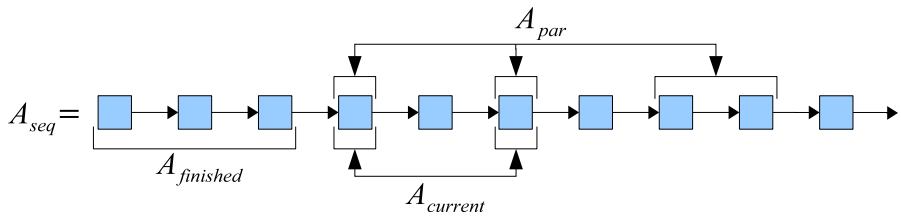


Figure 6.6: An illustration of how A_{seq}, A_{par}, $A_{current}$, and $A_{finished}$ are related to each other.

the postconditions of all other parallel actions. These tests may need further explanation. The first test (PRECONDS-HOLD(Pr_{a_i}, s_0)) is performed by evaluating the preconditions Pr_{a_i} in state s_0. The second test (PREV-PRECONDS-HOLD($A_{finished}, A_{seq}, a_i, s_{after}$)) goes through all the actions in A_{seq} previous to action a_i and verifies that their preconditions still hold after execution. The procedure PREV-PRECONDS-HOLD($A_{finished}, A_{seq}, a_{stop}, s$) starts with the first action in A_{seq} and tests the preconditions of this action in state s. The postconditions of this action are then used to update s before the next action in A_{seq} is tested, and so on until all actions before a_{stop} are tested. This test is performed to make sure that if we execute actions in parallel that are not directly succeeding each other, the actions in between these actions are still executable. PREV-PRECONDS-HOLD does not test the actions that are already finished (line 18). The third test on line 7 (COMPATIBLE-POSTCONDITIONS(Po^{par}, Po_{a_i}, s_0)) uses the variable Po^{par} that contains the postconditions of all the actions in A_{par}. The compatibility test on the postconditions of a_i with Po^{par} is performed to ensure that executing the actions in parallel will not lead to a causal conflict as defined in Section 3.4 on Page 27.

Example

Assume that we have a sequence of actions $A_{seq} = \langle a_0, a_1, a_2, a_3 \rangle$, $A_{current} = \{a_0\}$, $A_{finished} = \emptyset$, and we have the following pre- and postconditions and initial state s_0:

$$\begin{aligned} Pr_{a_0} &= b \wedge \neg c & \qquad Po_{a_0} &= c \wedge \neg d \\ Pr_{a_1} &= c \wedge \neg d & \qquad Po_{a_1} &= h \\ Pr_{a_2} &= b & \qquad Po_{a_2} &= d \\ Pr_{a_3} &= b \wedge \neg e & \qquad Po_{a_3} &= g \end{aligned}$$

$$s_0 = b \wedge \neg c \wedge d \wedge \neg e \wedge \neg g \wedge \neg h$$

Lets go through, step by step, how the procedure acts with this input. First (on line 2 and 3 of Algorithm 6.2), A_{par} is assigned the actions in $A_{current}$ and Po^{par} is assigned the postconditions in $A_{current}$. On line 4 the sequence is processed, action by action. The first time, when $i = 0$ and thus action a_0 is considered, nothing is tested since this action is in set $A_{current}$ (line 5). The remaining part of the sequence (i.e., $\langle a_1, a_2, a_3 \rangle$) is processed action by action on line 4 through 12. In the first iteration when a_1 is considered, the process creates a state $s_{after} = b \wedge \neg c \wedge d \wedge \neg e \wedge \neg g \wedge h$ (line 6) that reflects the conditions that hold after executing the action a_1. Then the different tests are performed. The PRECONDS-HOLD(Pr_{a_1}, s_0) returns fail since a_1 requires c but in state s_0 we have $\neg c$. Thus, a_1 cannot be executed in parallel with a_0. At the second iteration, when a_2 is considered, $s_{after} = b \wedge \neg c \wedge d \wedge \neg e \wedge \neg g \wedge \neg h$. PRECONDS-HOLD($Pr_{a_2}, s_0$) returns true since a_2 only requires b and this holds in s_0. PREV-PRECONDS-HOLD(A_{seq}, a_2, s_{after}) starts by testing the preconditions of a_0 in state s_3 and this returns true. It then applies the postconditions of a_0 to s_3 which gives $s = b \wedge c \wedge \neg d \wedge \neg e \wedge \neg g \wedge \neg h$. The preconditions of a_1 are tested in the new state s and they also hold, and thus PREV-PRECONDS-HOLD returns true. However, the last test COMPATIBLE-POSTCONDITIONS(Po^{par}, Po_{a_2}, s_0) with $Po^{par} = c \wedge \neg d$ and $Po_{a_2} = d$ fails since $d \wedge \neg d$ would result in conflict.

The last iteration of the for loop (lines 4 – 12), when action a_3 is considered, results in $s_{after} = b \wedge \neg c \wedge d \wedge \neg e \wedge g \wedge \neg h$. PRECONDS-HOLD($Pr_{a_3}, s_0$) returns true since both b and $\neg e$ holds in s_0. PREV-PRECONDS-HOLD(A_{seq}, a_3, s_{after}) also returns true (the test is left as an exercise for the reader). COMPATIBLE-POSTCONDITIONS(Po^{par}, Po_{a_3}) returns true since the postconditions do not cause any conflict.

Since all tests were true, a_3 is added to A_{par} and since a_3 is the last action in A_{seq}, the procedure returns $A_{par} = \{a_0, a_3\}$.

6.4.2 Combining Configurations

The next step in the approach is to see which of the actions in the a set of parallel actions A_{par} are possible to run in parallel when their configurations are taken into account.

In Section 5.4.3 we noticed that during the generation of a configuration plan, several configurations can be generated for each action. However, in the configuration plan, each action is represented only with one single configuration, i.e., the configuration with the lowest cost for that particular action. In the process of finding parallel actions it is again interesting to have several alternative configurations for each configuration. This is because it may not be possible to merge the configuration with the lowest cost for an action with any other configuration, but some other configuration with higher cost may be suitable for merging. In order to have several configurations for each action we simply store the alternative configurations for each action at planning time so that they can be used later during the parallelization of actions.

With several configurations for each action, the main point of this step in the approach is to find a merged configuration that can perform as many of the actions in A_{par} as possible. This can be achieved by combining all the configurations in the power set of A_{par}. In this subsection, we show how all the different combinations (merged configurations) can be generated, and in the next subsection we describe how to select which combination (merged configuration) to use.

The process of finding all possible combinations of configurations for the parallel actions can be done stepwise, combining two actions at a time. Algorithm 6.3 shows the pseudo code for how the combinations are generated in our approach. The procedure COMBINE-ACTIONS combines two actions at a time (i.e., creating all possible merged configurations for these two actions) and the result from this combination in turn is combined with the next action. If we for instance have $A_{current} = \{a_1\}$ and $A_{par} = \{a_1, a_4, a_6\}$, a_1 and a_4 are combined first resulting in a set of merged configurations $C_{1\|4}$. This set of configurations $C_{1\|4}$ is combined with the set of configurations C_6 for a_6 resulting in a new set of merged configurations $C_{1\|4\|6}$. COMBINE-ACTIONS uses the procedure GET-ASSOCIATED-CONFIGURATIONS (lines 3 and 6) that returns only the currently admissible configurations associated with the action. If there are no admissible configurations a nil-configuration is returned ($c^{nil} = \langle\emptyset, \emptyset\rangle$). As mentioned in the beginning of Section 6.4 the set $A_{current}$ contains either the actions that are currently running, or the next single action that is up for execution. If $A_{current}$ contains the currently running actions, GET-ASSOCIATED-CONFIGURATIONS returns the running merged configuration, otherwise if $A_{current}$ contains the next single action, the function returns the set of configurations associated to the action.

Recall from Section 5.4.3 that when generating a configuration plan from an action plan, only a subset of configurations are returned (only the k configurations with lowest cost). The reason for only having a subset of all configurations is that the number of combinations grows exponentially.

The procedure COMBINE-CONFIGURATIONS may need some explanation. The purpose with his procedure is to find all possible ways that the two configuration sets C^{first}, C^{second} can be combined. C^{first} can either be a set of configurations or a single configuration. Of course, this implies that it also makes a distinction between a single configuration and a set containing a single configuration. If C^{first} is a single configuration (line 14 Alg. 6.3) we merge this configuration with configuration c_1^{second} (i.e., the first configuration of C^{second}). This merged configuration c^{merge} is then included in the set of merged configurations that will be created when COMBINE-CONFIGURATIONS is called recursively. If C^{first} is a set of configurations (the else case on line 19), the procedure is called recursively to merge all possible combinations. The actual merge of single configurations is done by the procedure MERGE-CONFIGURATIONS (line 16 Alg. 6.3) that is detailed in Section 6.2.

Algorithm 6.3 Combine Actions

Require: A set of current actions $A_{current}$, a set of parallel actions $A_{par} = \{a_1, \ldots, a_k\}$ with associated sets of alternative configuration $\{C_1, \ldots, C_k\}$ where each $C_i = \{c_{i1}, ..., c_{im}\}$ where $m \geqslant 1$ and $k \geqslant 2$

Ensure: A set of merged configurations C^m

```
1: procedure COMBINE-ACTIONS(A_current, A_par, s)
2:     A^rest ← A_par \ A_current                ▷ Remove the current actions
3:     C^first ← GET-ASSOCIATED-CONFIGURATIONS(A_current, s)
4:     while A^rest ≠ ∅ do
5:         a^second ← POP(A^rest)
6:         C^second ← GET-ASSOCIATED-CONFIGURATIONS(a^second, s)
7:         C^first ← COMBINE-CONFIGURATIONS(C^first, C^second)
8:     end while
9:     return C^first
10: end procedure

11: procedure COMBINE-CONFIGURATIONS(C^first, C^second)
12:     if C^first = ∅ or C^second = ∅ then
13:         return c^nil
14:     else if C^first is a single configuration then
15:         c_1^second ← POP(C^second)
16:         C_merge ← MERGE-CONFIGURATIONS(C^first, c_1^second)
17:         C_rest ← COMBINE-CONFIGURATIONS(C^first, C^second)
18:         return APPEND(C_merge, C_rest)
19:     else
20:         c_1^first ← POP(C^first)
21:         C_first ← COMBINE-CONFIGURATIONS(c_1^first, C^second)
22:         C_rest ← COMBINE-CONFIGURATIONS(C^first, C^second)
23:         return APPEND(C_first, C_rest)
24:     end if
25: end procedure
```

The procedure COMBINE-ACTIONS in Algorithm 6.3 returns all combinations of the actions in A_{par}. To make it possible to exclude an action, we include a nil-configuration in all the configuration sets associated with the actions in A_{par}, except those of actions already executing. Note that merging a "normal" configuration with a nil-configuration result only in the "normal" configuration: e.g., $c_{11} \parallel c^{nil}$ is equivalent to c_{11}.

In summary, we have an action plan that can be parallelized. Although we use a polynomial time procedure to parallelize the action plan, it is worth to notice if some actions result in many configurations, we need to take into account an exponential amount of combinations. As a consequence, for domains for which many actions can be executed in parallel it is desirable to keep the number of alternative configurations for each action low.

6.4.3 Select Merged Configuration

The combine configurations algorithm returns a list of merged configurations for a set of parallel actions. This list contains all admissible combinations of configurations and is sorted lexicographically according to:

1. The number of actions combined in the merged configuration (in decreasing order).
2. The cost of the merged configuration (in increasing order).

With a sorted list of merged configurations, the selection is as trivial as picking the first merged configuration in the list.

6.5 The Top-Level Process: Modified for Parallel Action Execution

In Section 5.5 we detailed the top-level process required to generate and execute configuration plans. This top-level process needs to be modified in the execution part when it is desired to parallelize the action/configuration plan.

Figure 6.7 shows the new top-level process. The generation of configuration plans (steps 2, 3, A) is done as described in Chapter 5 with one exception; the alternative configurations are saved as we described in Section 6.4. The execution part also shares some of its steps with the previous version, but is modified to handle parallel configurations. The main difference is that since actions are executed in parallel by merged configurations, there may be an already running merged configuration that needs to be considered in the top-level process. That is, when considering the next action in the plan, the different steps in the top-level process must take into account that there may be other actions currently executing and that the next action should be one that can be executed in parallel with this running action. In practice this means that the sequencer does not just take the next action (sequentially) in the action plan if there are actions

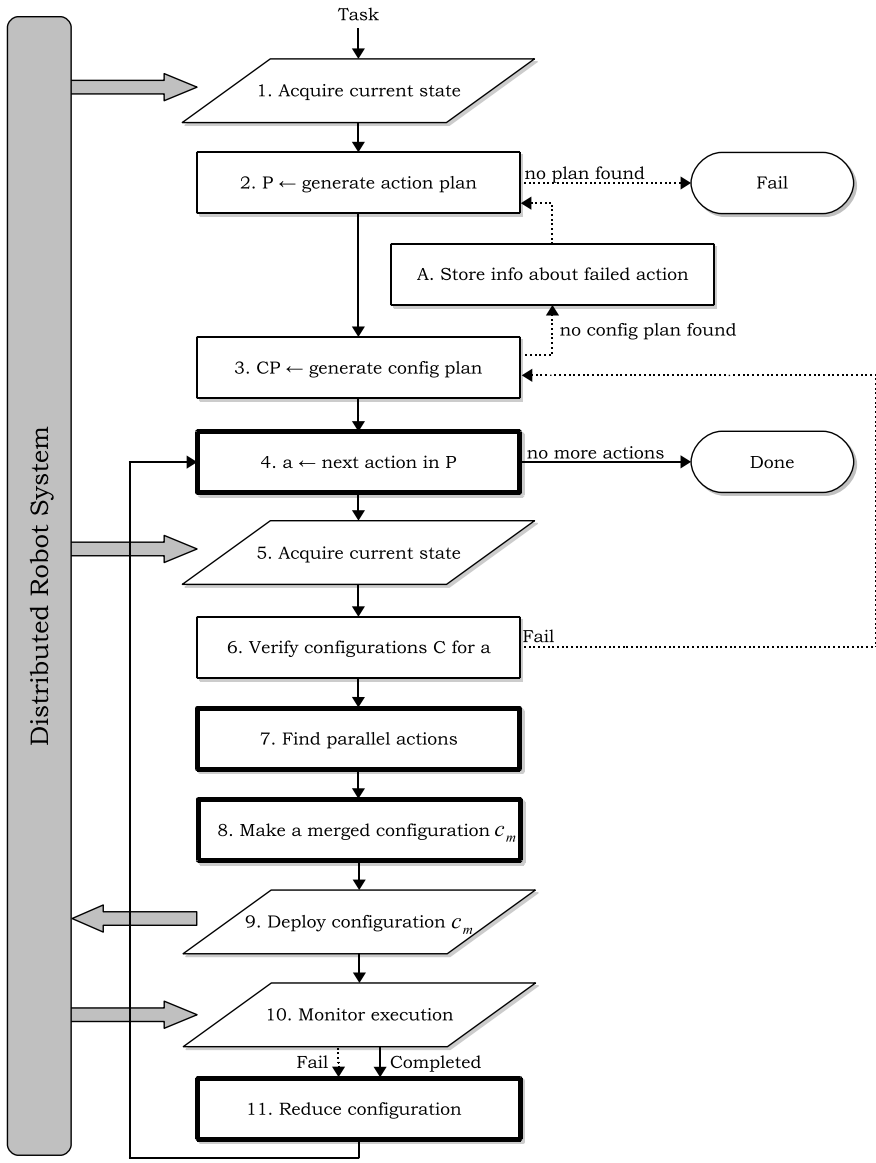


Figure 6.7: Flow chart of the top-level process. Steps that are substantially different or new compared to the top-level process in Figure 5.8 are marked with a thick border.

running, but rather searches through the remaining actions of the plan to get an action that can run concurrently with the already running actions.

6.5.1 Find Next Action/Configuration

When a configuration plan has been generated it is time to decide which configuration in the configuration plan to execute (Step 4 Figure 6.7). The normal procedure is to take the first configuration and then each time a configuration is finished to take the next configuration in the plan. This works fine if the configuration plan is executed in sequence, one configuration after the other. However, if we allow several configurations to execute in parallel, the selection of which configuration to execute next is more complicated. For the parallel execution it is actually the action plan that is considered, and which action that is up for execution. The main reason for considering actions instead of configurations is that an action actually represents several configurations. Recall that in the "find parallel actions/configurations" procedure, several configurations are considered for each action. The different configurations for the same action can have different pre- and postconditions. However, they all share the same base of preconditions and postconditions — the preconditions and postconditions of the action. The configurations can easily be retrieved from the configuration plan when it is necessary.

To select the first action to execute is done as in the previous version by picking the first action in the action plan. However, to select the next action when another action has finished is done in a different way. There are a number of different parameters that need to be considered in the selection of the next action. These are: if the action was completed or if it failed, if there is an action currently running or not, and if this is the last action in the plan or not. The pseudo code in Algorithm 6.4 shows how these parameters can be considered when selecting the next action in the action plan. The selection needs to know which action(s) that triggered the selection of "next" action, i.e., which action(s) that has completed or failed in the currently running configuration. These actions are denoted by A in the algorithm. The other inputs to this algorithm are the action plan (A_{seq}), the actions that are currently running ($A_{running}$), and the actions that have already finished ($A_{finished}$). The output of the selection is the actions, if there are any, that should be considered for parallel execution in the remaining steps of the top-level process. If there are no more actions to be executed in the plan, SUCCESS is returned.

The upper half (lines 2 – 14) of the algorithm considers the case when action A was completed successfully. Before the next action is fetched from the plan, the algorithm verifies that there are no other actions currently running (line 3). The next action is fetched by repeatedly popping the action plan until an action that is not already completed is found or until the end of the plan is reached (line 4 – 6). If the end of the plan is reached, the entire task has finished (line 8), otherwise the found action is returned as the next action (line 10). If there

instead are some other actions currently running (the else statement on line 13) the algorithm returns these actions as the actions to find parallel actions for.

The lower half (lines 15 – 21) considers the case when the action A has failed. If there are no other actions currently running, the failed action is returned. If there are other actions running (the else statement on line 18), the algorithm tries to find parallel actions for these. This is done because it might still be possible to execute the action that failed in parallel with the running configurations, provided that the configuration that failed is replaced by another configuration that does not include the functionalities that failed.

Algorithm 6.4 Select next action

```
 1: procedure GET-NEXT-ACTION(A, A_seq, A_running, A_finished)
 2:     if Action A completed then
 3:         if A_running = ∅ then                ▷ There are no other actions running
 4:             repeat
 5:                 a ← POP(A_seq)
 6:             until (A_seq = ∅) or (a ∉ A_finished)
 7:             if A_seq = ∅ then                ▷ The end of the action plan is reached
 8:                 Task has finished
 9:             else
10:                 return {a}
11:             end if
12:         else
13:             return A_running
14:         end if
15:     else if Action A failed then
16:         if A_running = ∅ then                ▷ There are no other actions running
17:             return A
18:         else
19:             return A_running
20:         end if
21:     end if
22: end procedure
```

6.5.2 State Acquisition

The current state is acquired in the same way as when a configuration plan is executed sequentially (See Section 5.5.1).

6.5.3 Verify Action/Configurations

The verification step works in a similar way as the verification in the top-level process for sequential execution (see Section 5.5.4). Verification is only performed if the action(s) returned by the next action function is not $A_{\mathtt{running}}$, otherwise it tries to find parallel actions directly. First it is verified that the action preconditions hold. That is, if the action is not applicable, none of its configurations can be applicable. If the action preconditions does not hold, then it is necessary to generate a new action plan and configuration plan for the new state.

In the case when the action preconditions actually hold, that is the normal case, the configurations for this action are validated. Each configuration is tested if its preconditions hold and if the resources are available. If there are no configurations that are applicable in the current state, then alternative configurations and alternative configurations plans are considered in a similar way as described in Algorithm 5.3.

6.5.4 Find Parallel Actions

This step includes finding which actions that can run in parallel with the action as described in Section 6.4.1.

6.5.5 Make a Merged Configuration

With a set of actions that are possible to be executed in parallel this step tries to make a merged configuration that can realize them on the configuration level. For each action, the configurations are verified before they are combined. If an action does not have any admissible configurations, this action is removed before the combination and merging process starts. The actions/configurations are then combined and merged as described in Section 6.2, Section 6.4.2 and Section 6.4.3.

6.5.6 Deploy Configuration

The deployment of a merged configuration is performed in a similar way as with an ordinary configuration. The only difference is that if there is a configuration already running, this configuration must be taken into account. The running configuration is always a part of the merged configuration that is about to be deployed (See Section 6.4.2), thus it is important to only deploy the parts of the merged configuration that are not already under execution. That is, only channels and functionalities that are not already running are setup and started.

When all the components in the merged configuration are deployed, the merged configuration is considered to be the new running configuration.

6.5.7 Monitor Execution

Is performed in the same way as when configurations are monitored in sequence (See Section 5.5.6).

6.5.8 Reduce Configuration

When a terminating functionality signals that it has terminated (i.e., completed its action), this configuration must be removed from the merged configuration as described in Section 6.3. The action that the configuration implemented is also added to the set of finished actions $A_{\texttt{finished}}$.

If instead there is a functionality in the running configuration that has failed, all configuration that depends on this functionality must be removed from the merged configuration.

No matter if the running configuration is reduced as a consequence of a failed or a completed configuration, the next step is always to take the "next" configuration as described in Section 6.5.1.

Chapter 7
Experiments

In the previous chapter we have described an approach for how a distributed robot system can be automatically configured to solve a task, how tasks can be solved with different configurations executed in sequence and how some configurations can be executed in parallel. As part of our research methodology (Section 1.3) to achieve the objectives given Section 1.2, we have implemented and tested our approach on a real distributed robot system. We have also conducted a number of different experiments to demonstrate that the approach is actually applicable to a real system.

The experiments have been performed to illustrate the different parts of our approach that are described in Chapters 4 – 6. The first two experiments (Section 7.3) were conducted on two real robots and only the approach to generate a *single configuration* described in Chapter 4 was used. In Section 7.4 we report five experiments that were conducted to test the approach for *sequences* of configurations described in Chapter 5. These experiments were tested on real robots and robotic devices. The last experiment in this chapter was conducted to illustrate the *parallel execution* of a configuration plan described in Chapter 6. This experiment was performed using a simulator and is reported in Section 7.5.

7.1 Purpose of the Experiments

The overall goal of the experiments is to provide a "proof of concept" of the different techniques presented in this thesis, i.e., to indicate their applicability to a real distributed system robot system. Additionally, the experiments serve the purpose to further clarify these techniques, by providing an illustration of how they work in practice. The aim of the experiments has *not* been to perform a quantitative evaluation of performance of the functional configuration approach. A full quantitative evaluation would have limited value, since the experiments would measure the performance of the entire system. To isolate and

quantify what is due to our approach, and what is due to some other aspect of the experimental system, would be very difficult.

In the next section we describe the experimental setup used for the different experiments.

7.2 Experimental Setup

In Section 4.9, the configuration approach is briefly mentioned as especially suitable for network robot systems [Sanfeliu et al., 2008], since these provide network services for communication and cooperation between robots, and since they also include other intelligent embedded devices besides robots. For our experiments we have selected a specific instance of a network robot system called the Peis-Ecology, described below.

Even though the experiments described in this thesis are implemented on this specific system, the approach can be used not only for network robot systems, but also multi-robot systems or even multi-agent systems in general, as long as the system is able to provide:

1. a way to dynamically acquire the current state of the environment and of the functional components in it,
2. a way to deploy the automatically generated configurations on the functional components that it incorporates, and
3. a way to execute and monitor the configurations to know when configurations have finished or failed.

In this section we give an overview of the Peis-Ecology concepts and of how the different parts of the top-level process(es) are realized in this framework. We also describe the robots used in the experiments, the environment in which the robots are situated, and the characteristics of the simulator used in the last experiment.

7.2.1 The Peis-Ecology

The concept of Peis-Ecology, originally proposed by Saffiotti and Broxvall [2005], is one of the few existing concrete realizations of the notion of network robot system [Sanfeliu et al., 2008]. The main constituent of a Peis-Ecology is a *physically embedded intelligent system*, or Peis. This is any computerized system interacting with the environment through sensors and/or actuators and including some degree of "intelligence". A Peis generalizes the notion of robot, and it can be as simple as a toaster or as complex as a humanoid robot. A Peis-Ecology consists of a number of Peis embedded in the same physical environment, and endowed with a common communication and cooperation

```
(functionality
 :name    (object-tracker ?r ?f ?obj)
 :precond ( ((?r ?f) (funct ?r object-tracker ?f))
            ((?obj) (object ?obj)) )
 :in      ( ((image ?r) image) )
 :out     ( ((pos ?r ?d) real-coordinates)
            ((orient ?r ?d) radians) )
 :cost    5)

(functionality
 :name     (odor-classifier ?p ?f ?obj ?pred1 ?pred2)
 :precond  ( ((?r ?f) (funct ?p odor-classifier ?f))
     ((?obj) (object ?obj)) )
 :in       ( ((smell ?obj) smell)
             ((rfid-reading ?obj) tag) )
 :output   ( ((odor ?obj) signature) )
 :resource ((odour-classifier ?p ?f))
 :term     T
 :cost     10)
```

Figure 7.1: Two functionalities operators for a PEIS-Ecology.

model. The PEIS-Ecology model has been implemented in an open-source middleware, called the PEIS-kernel. For communication, the PEIS-kernel creates and maintains a fully decentralized P2P network, and performs services like network discovery and dynamic routing of messages between PEIS. The PEIS-kernel hides the underlying network heterogeneity, and it can cope with an environment in which PEIS and their individual components may appear and disappear at any time.

The cooperation model is also implemented via the PEIS-kernel, which provides a distributed tuple-space augmented by the event-based primitives `subscribe` and `unsubscribe`. If a component A wants to use a functionality from a component B, possibly residing in a different PEIS, it subscribes to a suitable tuple key. When an insert operation is performed by B with that key, component A is notified and can consume the corresponding data. The PEIS-kernel also implement a reflection mechanism: the subscription links are themselves represented by tuples in the tuple-space, called *meta-tuples* , thus allowing for both introspection and dynamic management of the subscriptions.

In our experiments, we use our self-configuration framework to automatically configure a PEIS-Ecology. There are three elements in our framework that depend on the specific system being configured: the domains used by the planners, the way to acquire the current state, and the way to deploy and monitor a configuration.

For the domains, since these are static they have been hand-coded. Fig. 7.1 shows two examples of functionality operators for the configuration planner in the PEIS-Ecology domain (See Appendix A for the full PEIS-Ecology domain).

To acquire the current (robot and environment) state from the PEIS-Ecology, we use the mechanisms provided by the PEIS-kernel. To acquire the robot state, the configuration process subscribes to the reserved key `name` to retrieve, for each functionality in the ecology, a tuple that contains its name, location (PEIS), status (on/off), cost, and the resources it provides. On each PEIS there is a special functionality, called PEIS-init, which publishes this information in the distributed tuple-space.[1] For the environment state, each functionality publishes a tuple describing its own part of the state, e.g., its current physical location and/or the state of an actuator attached to it. All these state variables are put together into an overall environment state, used by both the configuration planner and the action planner. In our experiments we complemented the acquired state with state variables that are static. The static state variables do not normally change. Examples of such variables are the names of the different rooms, places in rooms and positions of static objects. These facts can be provided by different components in the environment, for instance a robot could map the environment initially and in that way obtain the required basic facts. In our experiments, the static state variables are hand-coded. Figure 7.2 shows some

```
(room kitchen) (room living-room) (room bedroom) (door Door1)
(door Door2) (connects Door1 kitchen living-room)
(connects Door2 living-room bedroom) (place fridge kitchen)
(place entrance living-room) (place table kitchen)
(place bed bedroom) (place charger living-room)
```

Figure 7.2: Example of static state variables

static state variables for the PEIS-Home described in the next section. These variables describe that there are 3 rooms: kitchen, bedroom, and living-room. They also describe how the rooms are connected, different places in the rooms and so on.

Deployment of a configuration description into a PEIS-Ecology is done in four steps. First the resources of the configuration are allocated. Second, inactive functionalities are activated through the PEIS-init functionalities. Third, the needed channels are set-up by creating tuple subscriptions between the involved functionalities. Finally, the configuration process subscribes to `done` tuples from terminating functionalities to be notified of completion; it also subscribes to `fail` tuples from all the functionalities in the configuration, to be notified about failures.

[1] The cost of an active functionality is provided by the functionality itself; for an inactive functionality, PEIS-init provides an estimate of the cost of starting it.

7.2.2 The PEIS-Home Testbed

We have run our experiments in a physical test-bed facility, called the PEIS-Home, which looks like a typical bachelor apartment of about $25m^2$ — see Figure 7.3 and Figure 7.4. It consists of a living-room, a bedroom and a small kitchen. The PEIS-Home is equipped with a communication and computation infrastructure, and with a number of PEIS. All the PEIS run the PEIS-kernel, and they are all connected via either wired or wireless Ethernet (802.11 WLAN), except for the **Lamp** and **Ambient Light Sensor** which is connected via ZigBee. The following PEIS are of particular importance for our experiments.

Pippi

Pippi is an iRobot Magellan Pro robot from iRobot Corporation (See Figure 7.5). The Magellan Pro robots are rather small with a height of 25 cm and a diameter of 40 cm. They have three wheels; two wheels on each side with differential drive and a third rear wheel (for balance) that rotates freely. The two driving wheels are equipped with encoders for odometry measurements. Moreover, this platform has three different types of distance/occupancy sensors mounted together, equally distributed around the robot, enabling the robot to sense obstacles in all directions. These sensors are 16 tactile sensors (bumpers), 16 infrared senors, and 16 sonars. The robot is also equipped with a flux gate compass, a CCD color camera, a Cyranose 320 electronic nose (E-nose) used to classify odors, and a RFID-reader.

The Magellan Pro robots are also equipped with an on-board computer with a 550 MHz Pentium III processor running Linux. **Pippi** runs the Thinking Cap [Saffiotti et al., 1995] navigation controller, and several instances of Player [Player/Stage Project, 2007] providing functionalities to interface with different sensors and actuators. She can also run the configuration process. To interpret the information from the camera, there is a vision system that can identify simple predefined shapes using color segmentation. In order to use this simple vision system, the shapes that need to be identified must be in uniform colors.

Emil

Emil is also an iRobot Magellan Pro robot like **Pippi**, but instead of the E-nose he is equipped with a SICK PLS laser range finder. **Emil** runs the same software as **Pippi** except that instead of the E-nose software, he is running a self-localization program based on scan-matching techniques.

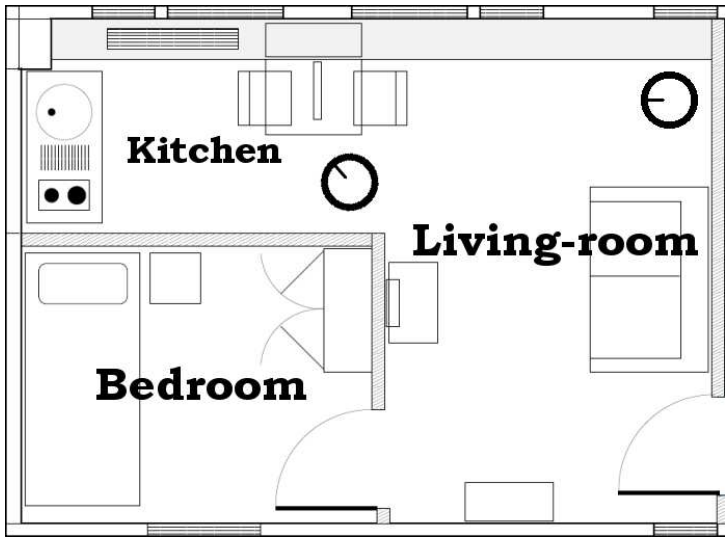


Figure 7.3: A map of the apartment. The black circles represents two of the robots. **Pippi** to the left and **Astrid** to the right.

Figure 7.4: Snapshots from the PEIS-Home. Left: **Pippi** in the kitchen. Right: **Astrid** in the living-room.

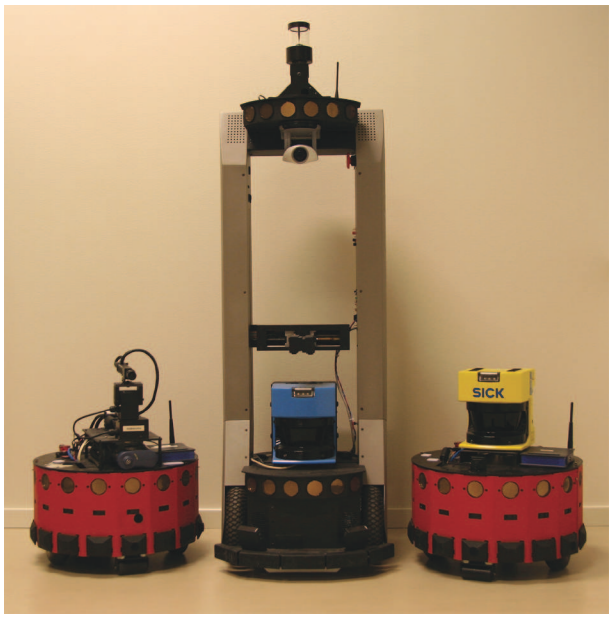


Figure 7.5: The robots used. Form left to right: **Pippi**, **Astrid**, and **Emil**

Astrid

Astrid is a PeopleBot robot from ActivMedia Robotics (See Figure 7.5). The PeopleBot platform is a rather tall robot (height 112cm) with a width of 38 cm and depth of 48 cm. The sensor suite is composed of:

- three sonars rings of 8 sonars each to get a 360° coverage. Two of the sonar rings are placed 20 cm above ground with one ring facing forward and the other facing backward. The third sonar ring is placed facing forward at the top of the platform.
- 16 bumper sensors.
- 2 IR sensors that point up and slightly forward, for sensing a table.
- An omnidirectional video camera (CCD) and a Pan-Tilt-Zoom color camera (Canon VC-C4)
- A SICK LMS laser range finder.

In addition to the sensors, the PeopleBot is equipped with a 2 DOF gripper, a speech synthesis system, and (optionally) a Cyranose 320.

The on-board PC is a low-power industrial SBC Pentium M 1.6 GHz running Linux. As **Pippi** and **Emil**, **Astrid** can run the Thinking Cap, Player, and the configuration process, as well as a scan-matching self-localization algorithm.

Fridge

An apartment sized refrigerator ($51\text{cm} \times 53\text{cm} \times 81\text{cm}$) with two simple Figaro gas sensors, a motorized door, an RFID tag reader that can read the tags on the items placed in the **Fridge**, and an embedded computer.

Tracker

A tracking system connected to a stereo camera mounted on the ceiling, capable of tracking multiple persons [Munoz-Salinas et al., 2007]; the tracker is also capable of tracking the position of robots. In an earlier version, the **Tracker** was connected to four ceiling cameras, only tracking marked objects [Lilienthal and Duckett, 2003].

Table

An ordinary table equipped with an RFID tag reader that can register whether there is a tagged object on the table.

Lamp

A lamp controlled by a MoveIV T-Mote. A T-Mote is a small electronic device with limited processing and communication capabilities. The T-Mote runs a tiny version of the PEIS-kernel [Bordignon et al., 2007].

Ambient Light Sensor (ALS)

A light sensor on a second MoveIV T-Mote. Publishes the ambient light status of the PEIS-Home.

Home Security Monitor (HSM)

A standard PC that runs a monitoring component that reacts to alarms published in the tuple-space. It also runs an instance of the configuration program, a phone interface, and a ZigBee bridge.

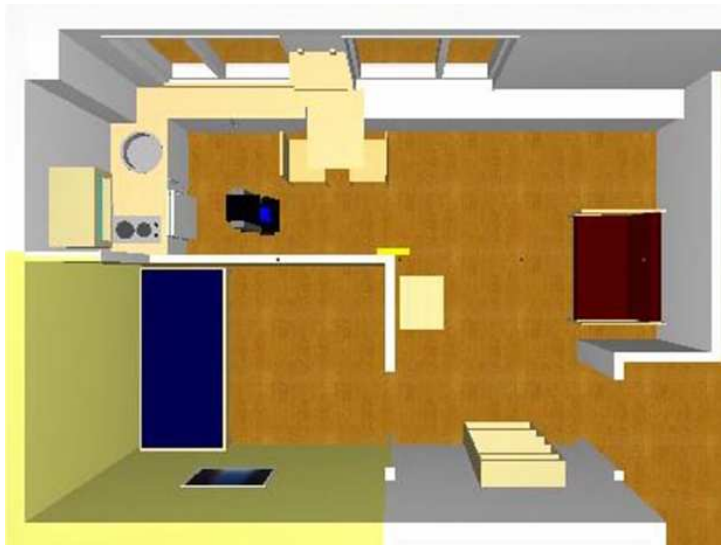


Figure 7.6: The PEIS-Home simulation environment, in top-view.

7.2.3 The PEIS-Simulator

A mid-fidelity 3D simulation of the PEIS-Home testbed has also been implemented using Gazebo [Player/Stage Project, 2007][Larsson, 2007]. Figure 7.6 shows a top-view of the simulated PEIS-Home. The simulated version includes

models of **Astrid**, the **Fridge**, **Tracker**, **Lamp**, and **AMS** (Ambient Light Sensor). The simulated **Fridge** is equipped with a manipulator that can pick up things inside the fridge and give them to a robot that is near ("docked") the **Fridge**[2]. Most of the experiments reported in this thesis have first been tested in the simulator before they have been performed on the real robot ecology. The last experiment (See Section 7.5.1) was only performed in the simulator.

7.2.4 Experimental Methodology

In each one of the experiments performed the following information was hand-coded:

- a domain description of all the existing functionalities with methods to combine them. In the latter experiments (Section 7.4 – Section 7.5) the domain also includes action operators (See Appendix A),
- a set of static state variables (e.g., describing the topology or the apartment), and
- a task, described with a first order logic formula.

This information encodes the set of physical PEIS and their placement, the set of available functionalities and their properties, and the target task. These can be seen as the independent variables in our experiments. Some, if not all, of these independent variables are changed from one experiment to another as described in the setup section of each experiment collection.

The expected result from the experiments is that the system is able to automatically solve the task by self-configuration, and to possibly deal with induced exceptions like functionality failures and/or inability to find configurations. We expect the task to be completed without any human intervention. The only role of the human operator is to provide the independent variables above to the system, introduce failures to demonstrate robustness, and to observe the execution until the task is either completed, or it fails. Thus, the success criteria for each experiment is that the corresponding task is completed.

7.3 Experiments on Single Configurations

In this section we describe two different experiments performed to illustrate the single configuration generation described in Chapter 4 and the top-level process in Figure 4.13. These experiments were performed in a lab environment and not in the PEIS-Home.

[2]A similar manipulator for the real **Fridge** has been recently developed.

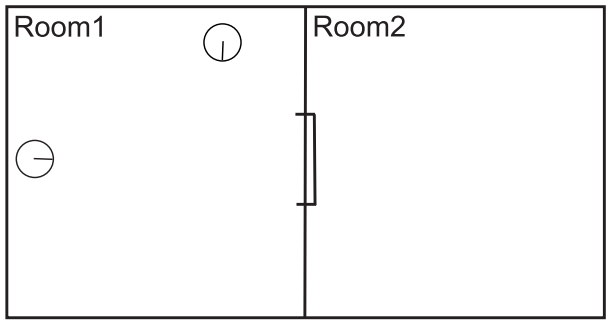


Figure 7.7: A map of two rooms connected with a door. Robots are displayed as circles with a line giving their heading. In room1, **Emil** is to the left and **Pippi** to the right.

7.3.1 Experiment 1: Cross a Door

The first experiment is inspired by the example given in Chapter 3 in which a robot helps another robot to push a box through a door.

Question

Can our approach for generating configurations, and thus sharing functionalities, enable robots to help each other to solve individual tasks that they cannot solve alone?

That is, can a robot with broken sensors or a poor sensor suite generate configurations in which it "borrows" functionalities from other robots, such that the robot is able to perform its task? The success criteria for this experiment is that the robot is able to perform the assigned task by using the configuration framework.

Setup

In the setup of this experiment we have two robots, **Emil** and **Pippi**, and two rooms, R1 and R2. Both rooms are square-shaped with sides of 3 meters. Figure 7.7 shows the map of the environment with **Emil** and **Pippi** at their initial positions. The rooms are connected with a door that the robots must pass through.

In order to cross the door, a robot must be aware about the position and orientation of the door relative to itself. Under normal circumstances, when the robots can use sonars and odometry, this is not a difficult task. However, if the robot is pushing a box, these sensors may be unusable/unreliable, i.e., the sonars may be obscured by the box and the odometry may suffer due to slippage

Figure 7.8: Emil is guiding **Pippi** through the door. Door posts and robots are "dressed" in uniform colors.

of the wheels induced by the pushing. It can also be the case that one of the robots, or even both, lacks this type of sensor. To show that it is possible to solve this task cooperatively, without sonars and odometers, the robots are restricted to only use their cameras and compasses. The cameras are fixed and can only measure distances to objects further away than 2 meters. Some modifications to the environment have been carried out in order to simplify the vision task, e.g., the door and the robots have been marked with uniform colors (see Figure 7.8). If the robots can perform this task by executing the configurations produced by the configuration planner, we can say that the experiment has been successful.

Execution

The experiment unfolds as follows:

a. **Pippi** and **Emil** are in Room1. **Pippi** is assigned the task to go from Room1 to Room2. The configuration process located at **Pippi** acquires the current state (See Figure 7.9), and based on this generates a configuration (steps 1 and 2 of the process in Figure 4.13 on Page 64). Since the camera can only measure distances to objects further away than 2 meters, **Pippi** is not able to perform the action on its own. **Emil** is equipped with the same sensors as **Pippi**, and he is able to observe both the door and **Pippi** from a distance during the whole procedure. The configuration generated is shown in Figure 7.10.

b. This configuration is then deployed and executed by **Pippi** and **Emil** (steps 3 and 4). During the execution of "cross-door", **Pippi** continuously re-

```
( (robot Pippi) (robot Emil) (door Door1)
  (room Room1) (room Room2) (connects Door1 Room1 Room2)
  (in Pippi = Room1) (in Emil = Room1)
  (funct Emil camera id21) (funct Emil compass id22)
  (funct Emil measure-door id23) (funct Emil measure-robot-pos id24)
  (funct Emil measure-robot-angle id25)
  (funct Emil measure-robot-orient-compass id26)
  (funct Emil measure-robot-orient-camera id27)
  (funct Emil transform-info id28) (funct Emil cross-door id29)
  (funct Pippi camera id11) (funct Pippi compass id12)
  (funct Pippi measure-door id13)
  (funct Pippi measure-robot-pos id14)
  (funct Pippi measure-robot-angle id15)
  (funct Pippi measure-robot-orient-compass id16)
  (funct Pippi measure-robot-orient-camera id17)
  (funct Pippi transform-info id18) (funct Pippi cross-door id19) )
```

Figure 7.9: The (partial) state used at start up in Experiment 1.

ceives information about the position and orientation of the door. When **Pippi** enters Room1 the terminating functionality (`cross-door`) signals that the task is accomplished, and the current configuration is played out.

c. Next, **Emil** is assigned the task of going from Room1 to Room2. The configuration process on **Emil** acquires the state and the same configurations as before is generated, but with the roles exchanged — i.e., **Pippi** is now guiding **Emil** (See top configuration in Figure 7.11). This configuration is then deployed to **Emil** and **Pippi**.

d. During the execution of the "cross-door" behavior, the operator introduces an artificial failure on **Emil**'s compass by setting its failure tuple. Since the top-level process subscribes to all failure tuples, it is able to detect that the compass has failed (step 4 in Figure 4.13). This makes the current configuration not admissible, and a reconfiguration is necessary to proceed.

e. The state is again acquired (step 1), this time with **Emil**'s compass marked as failed. Since **Emil** is not able to use the compass, the robots cannot get the orientation of each other in the same way any longer. Without the possibility to use compasses, the bottom configuration in Figure 7.11 is generated and selected. Instead of using compasses to obtain the orientation differences, these measurements are obtained by having the two robots facing each other. In this way, the robots are able to measure the bearing to each other, and with this information it is possible to calculate

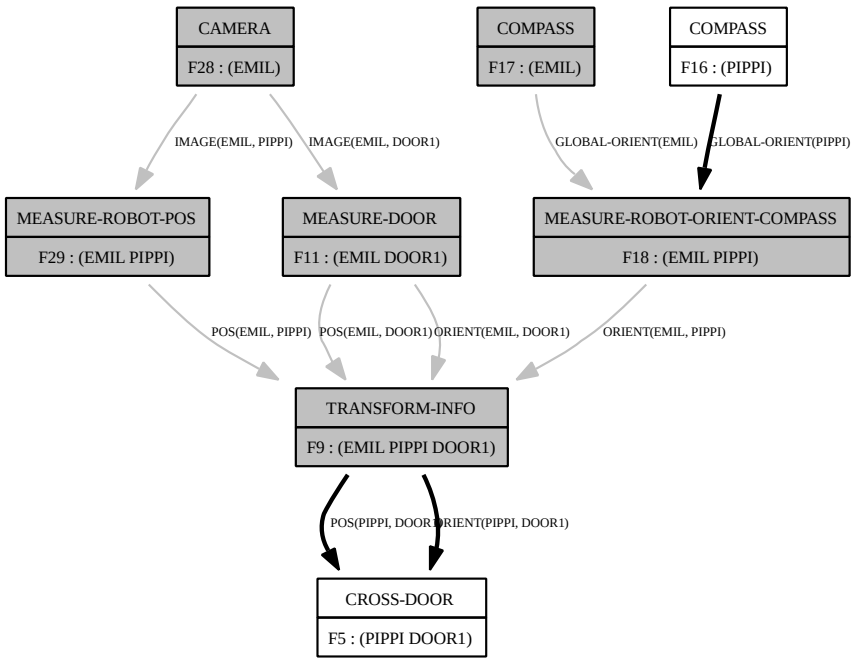
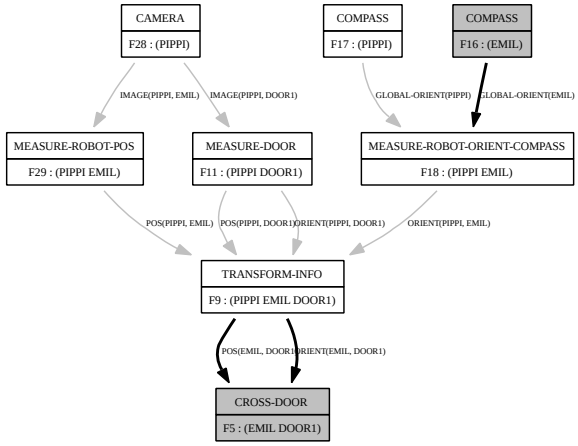


Figure 7.10: A configuration for the "cross a door" experiment. **Pippi** crosses the door with help from **Emil**. The colors of the boxes show where the functionalities are located; gray for Emil and white for Pippi. Channels between robots are represented with black thick arrows and channels within robots with gray thin arrows.



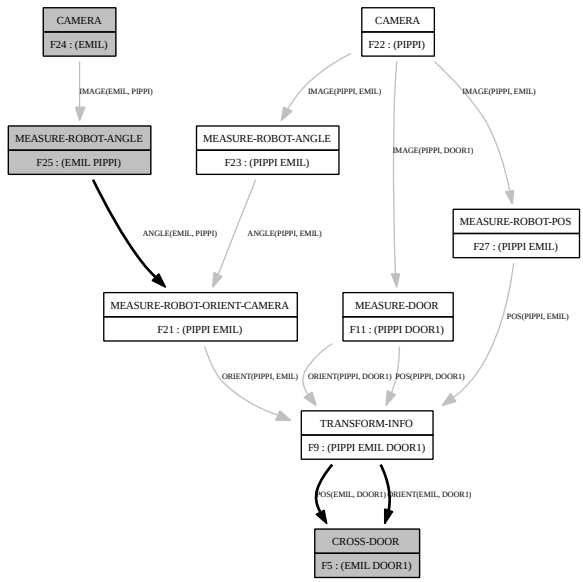


Figure 7.11: Two configurations for the "cross a door" experiment. **Emil** crosses door with help from **Pippi**. Top: Orientation from on board compasses. Bottom: Orientation from cameras.

the orientation differences. This new configuration is used to carry out the remaining part of the task.

Results

The general aim of these experiments was to to indicate the applicability of the single configuration planner to a real distributed robot system and to illustrate how it works in practice. The specific aim of the above experiment was to illustrate that by enabling robots to cooperate through functional configurations, robots can help each other to solve individual tasks in alternative ways. Have these aims been met?

In the experiment, **Pippi** generated a configuration that made **Emil** help her to cross the door. After the execution of this configuration, **Pippi** successfully reached room R2. **Emil** then got the task to also go to room R2. He generated a configuration in which **Pippi** helped **Emil** to cross the door. That configuration became inadmissible and **Emil** reconfigured so that he could continue to perform the task. After the execution of that configuration, **Emil** also successfully reached room R2. At the end of the experiment, both robots had reached R2 as desired. No human intervention was required at any time between the initial assignment of the goal and the final verification of the results except for the injection of the faulty condition. The experiment successfully met these aims.

Figure 7.12 shows the trajectories performed by the robots in a sample run of this experiment. In the picture, **Pippi** is standing at the observing position and **Emil** has just accomplished his task. The temporal diagram (i.e., Gantt chart) in Figure 7.13 gives a clear overview of when and for how long the different robots and their functionalities were used in the experiment. In the diagram, the vertical axis gives the different robots (or PEIS) with their functionalities. On the horizontal axis we have time. The rectangles represent the activity of the functionalities or the top-level process during a period of time defined by its horizontal size. At the top we also have the protocol followed by the operator in the experiment.

The diagram in Figure 7.13, together with the configurations in Figure 7.10 and Figure 7.11, also shows how the same functionalities can be used and connected to address different tasks and/or different conditions.

7.3.2 Experiment 2: Carry a Bar

The second experiment is a cooperative object transportation experiment. Cooperative object transportation is usually considered to require tight coordination.

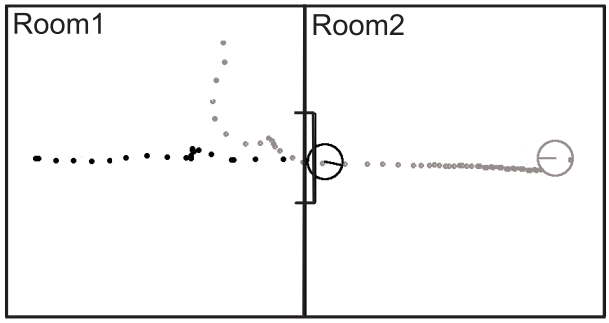


Figure 7.12: Pippi and **Emil** have both reached room R2. Gray dots show **Pippi's** trajectory and black dots show **Emil's** trajectory.

Question

Can our approach to generate configurations address a traditional tight coordination task that involves the motion of two robots?

For this experiment to be successful, the robots must automatically generate a configuration that can be used to carry out an object transportation task.

Setup

As in the previous experiment, the two robots **Pippi** and **Emil** are used. To enable the robots to carry objects, we have mounted a stand on each robot that is able to hold a bar. The stand is constructed in a way so the bar is resting in a row lock and is not physically gripped. This gives the robots a bit more freedom in terms of the accuracy of their motions; they are allowed to not be perfectly synchronized in their motions. The bar is placed in the stands of the two robots by a human operator A and the task is to transport it to a human operator B in the other room. The sensors available in this experiment are laser range finder, sonars and odometry on **Emil**. For **Pippi**, only sonars and odometry are available. Figure 7.14 shows a picture of the two robots with the bar. The picture to the right shows a close up of the stand.

In the cross door experiment it was relatively clear that in order for a robot to cross a door it needs to know the position and orientation of the door to cross. In contrast, it is not clear what kind of information the robots require in order to cooperatively carry the bar. Hence, there are various ways to carry a bar. For example, the robots can carry the bar with one robot going first and the other robot simply following the first one. One alternative way to carry the bar would be to go side by side. These two solutions would require different information. Further, there are several ways to keep the distance when carrying a bar

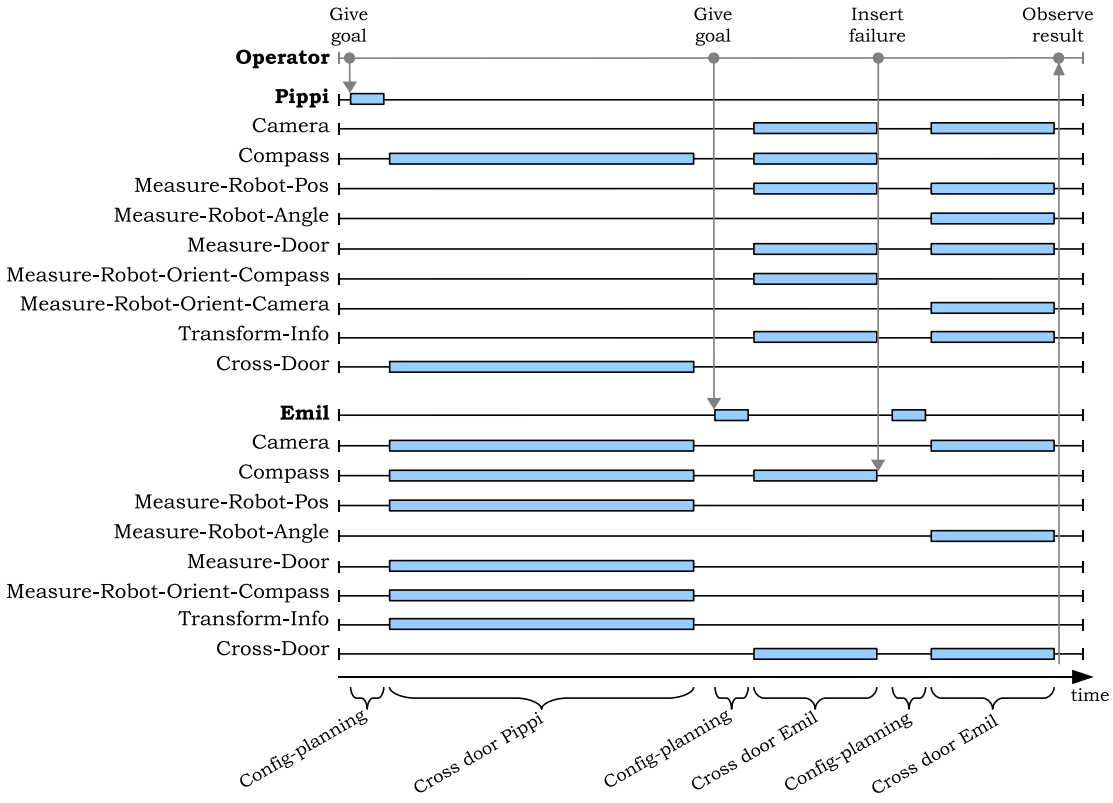


Figure 7.13: A temporal diagram of the cross a door experiment that shows which functionalities are active and when during the execution of the tasks. The actions by the human experimenter are shown in the top row.

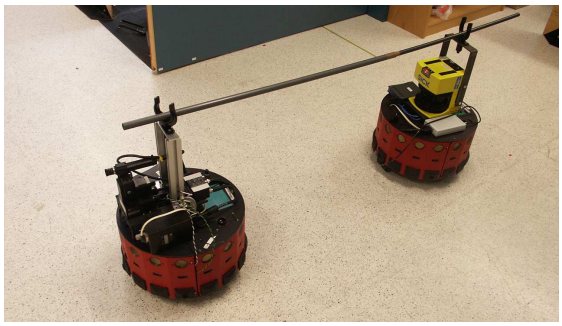


Figure 7.14: The experimental setup for the carry bar experiment. Right: the stand that enables the robot to hold the bar.

in a lead-follow manner. If possible, the robots can measure the force/moment from the object. The follower robot can then adjust its speed according to the push or pull force (e.g. as in [Wang et al., 2004]). If the robot is not able to directly sense these forces, information about angle to, distance to, and speed of the leading robot can be used in a similar way. In theory, the information about the distance and angle would be enough to perform the carry task, letting the follower robot keep the distance. In order to get an accurate and smooth behavior this assumes that the information is received continuously. In practice, it is better to also use the speed to get a more accurate and smooth behavior. There is also another aspect to consider. If we adopt a simple lead-follow behavior, the follower robot is responsible of keeping up with the lead robot. This is not always a good solution. The follower robot can run into problems and in that case the leading robot must slow down or stop to assure that the object is not dropped. Therefore, it is necessary that the leading robot is aware of at least the distance to, but preferably also the speed of, the follower robot.

There are probably many more data that can be measured or extracted to make the robots transport an object in a better manner. However, the aim of this experiment is neither to make the robots carry the bar in all possible ways, nor to make them perform the task perfectly. As stated previously, the aim is to demonstrate that it is possible to address this type of task using our configuration framework, i.e., to acquire the state, automatically generate configurations, deploy, and finally execute the configuration. Therefore, in this experiment we are satisfied that the "follower" robot uses the distance and angle to the leading robot together with the relative speed between the robots. The leading robot needs to know the distance between the robots.

Execution

The experiment unfolds as follows:

a. **Pippi** is assigned the task to transport a bar from A to B with help from **Emil**. **Pippi** acquires the current state according to step 1 in Figure 4.13 on Page 64. The (partial) state looks as follows:

```
( (robot Pippi) (robot Emil) (funct Pippi sonars id10)
  (funct Pippi odometer id11) (funct Emil sonars id20)
  (funct Emil odometer id21) (funct Emil laser id22) )
```

The functionality operators (describing functionalities for moving, following, measuring and so on) and methods for combining them constitute the domain for the experiment. With this state, domain and the goal (`do-carry-bar Pippi`), the configuration planner (step 2 Figure 4.13) generates the configuration shown in the upper part of Figure 7.15. In this configuration, **Emil** measures the distance and angle to Pippi using a laser range finder. There are two alternative ways in which the task can be solved. In these alternative configurations, the distance and angle to the leading robot is obtained by global localization of both robots. One of the alternative configurations is shown in the lower part of Figure 7.15. The other alternative configuration (not shown) is similar but with roles changed, i.e., **Pippi** follows **Emil**. The cost of the generated configuration is 33 and the costs of the alternative configurations are 45 each.

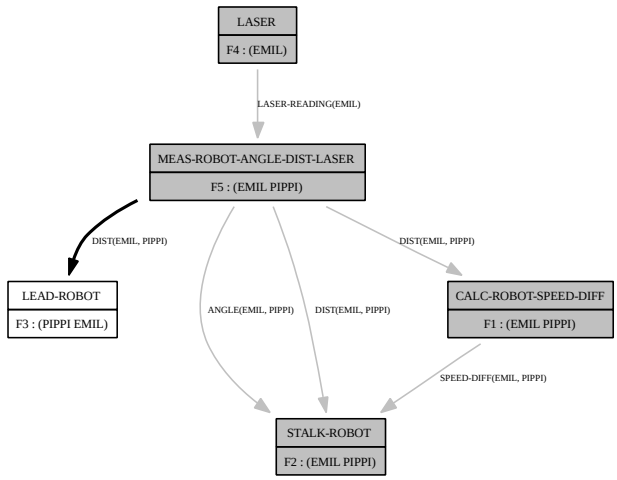
b. The top configuration in Figure 7.15 is deployed (step 3) to the robots and execution starts (step 4).

While executing this configuration, **Emil** is measuring the angle and distance to **Pippi**. The distance is transferred to **Pippi** trough a channel but also to a functionality that calculates the speed. This functionality has a memory and access to a clock. The lead-robot action uses the information of distance to avoid getting too far ahead and the follow-robot uses the distance, angle, and speed to smoothly keeping up with **Pippi**.

c. During the execution of this configuration, the operator simulates a failure of the laser by setting its failure tuple. The configuration process is notified of the failure through its subscription and stops the current configuration. The configuration process then again acquires the state (step 1) and the configuration planner generates new configurations (step 2). When the laser is unavailable, the current (partial) state looks as follows:

```
( (robot Pippi) (robot Emil) (funct Pippi sonars id10)
  (funct Pippi odometer id11) (funct Emil sonars id20)
  (funct Emil odometer id21) )
```

For this state, the configuration planner generates only the two expensive configurations, with costs of 45. The configuration in which **Emil** follows **Pippi** is selected (the configuration at the bottom in Figure 7.15) When



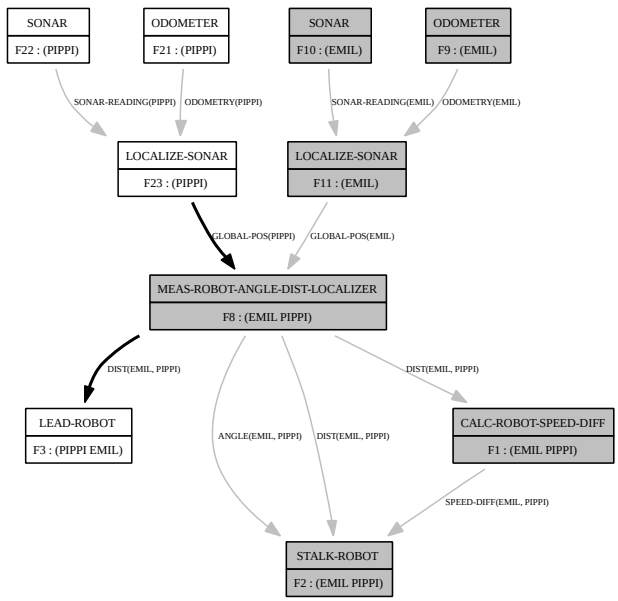


Figure 7.15: Two configurations generated for two robots that carry a bar. Top: **Emil** uses laser to measure angle and distance to **Pippi**. Bottom: Both robots are localized and global positions are used to calculate necessary information.

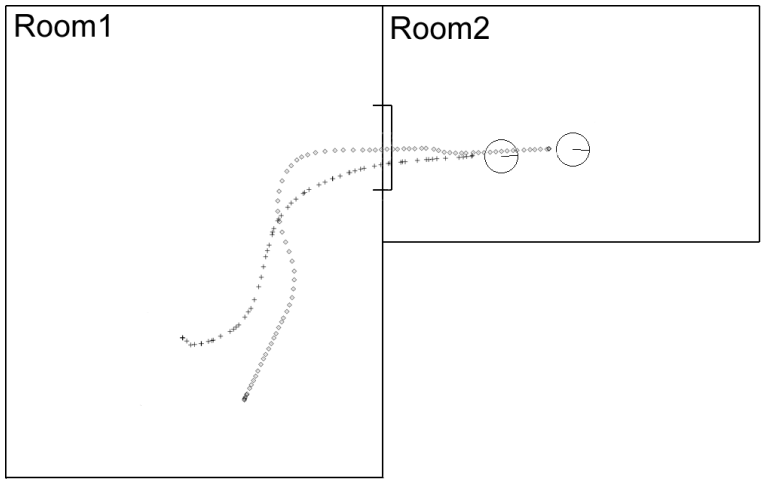


Figure 7.16: Pippi and **Emil** have reached human operator B. Diamonds(◇) show **Pippi**'s trajectory and pluses(+) show **Emil**'s trajectory.

configurations have equal cost, the order of the method specifications in the domain decides the order of the configurations. That is, there are two versions of the top method for this experiment: one in which the helper follows and one in which the helper leads. In the configuration in the figure, **Pippi** is the leader and **Emil** is the follower, and in the configuration not shown, the roles are interchanged. The selected configuration is then deployed (step 3) to the robots and execution starts (step 4).

d. While executing the configuration in Figure 7.15, **Emil** and **Pippi** must localize in the environment with their sonars and the odometry. The position information is sent to a functionality that calculates the distance and angle to **Pippi** in **Emil** coordinates. The other parts of the configuration are the same as for the configuration with the laser.

   With this configuration, **Pippi** and **Emil** are able to successfully carry the bar from human operator A to human operator B. See Figure 7.16.

Results

In addition to the general aims described in Section 7.1, the specific aim of the above experiment was to demonstrate that the configuration framework is able to handle tasks that require tight coordination. Did the experiment meet these aims?

In this experiment, **Pippi** automatically generated a configuration in which she and **Emil** carried a bar together. During the execution of the configuration, the two robots continuously exchanged information that enabled them to perform the task in a tightly coordinated manner. The robots managed to complete the task, even though a functionality failure was introduced by the operator. As a response to the failure, **Pippi** generated an alternative configuration that also enabled the robots to, tightly coordinated, perform the task. The aims of the experiment was successfully met.

7.4 Experiments on Sequences of Configurations

In this section we describe five different experiments performed to illustrate the approach for generating sequences of configurations described in Chapter 5 and its top-level process (Figure 5.8).

7.4.1 Experiment 3: Smell Inside the Fridge

This experiment and the next three experiments are based on a scenario originally presented by Broxvall et al. [2006]. It should be emphasized that in the original work all the configurations were encoded by hand, and the aim of the experiment was to show the benefit of using a PEIS-Ecology approach for mobile olfaction.

We performed a different experiment using a similar setup to answer the following question.

Questions

Can our approach automatically configure a network robot system to perform a given task, even when this task requires several sub-tasks to be accomplished in a sequence?

Setup

The experiment was performed in the PEIS-Home with the task to determine what is smelling in the **Fridge**. As in the original scenario [Broxvall et al., 2006], there are several simple gas sensors distributed in the PEIS-Home. The **Home Security Monitor** subscribes to information from each one of these sensors and in that way it is notified when a smell is detected (the gas sensor publish a tuple). However, the simplicity of the gas sensors make them unable to classify smells, i.e., they can only detect that there is a smell. The idea is that a mobile robot equipped with a more sophisticated gas sensor (E-nose) can use the location information provided by the distributed gas sensors to approach the odour source and determine what is smelling. The smell classifier also use addition information provided by other sensors to better classify the smell. For

example, when smelling inside the **Fridge** an RFID-reader is used to obtain identity information of objects currently located in the **Fridge**. In that way, the classification context is restricted and a more accurate result can be obtained. In this first experiment, our framework will generate automatically the same sequence of configurations that was hand-coded in the original scenario [Broxvall et al., 2006]. The next three experiments go beyond what could have been hand-coded in the original system.

In this experiments the following PEIS are available: **Pippi**, **Astrid**, the **Tracker**, the **Fridge**, the **Lamp**, the **ALS** (Ambient Light Sensor), and the **HSM** (Home Security Monitor). Both **Pippi** and **Astrid** are equipped with an electronic nose. At start-up, **Pippi** is located close to the entrance, and **Astrid** is charging her batteries at the charging station in the living-room.

Execution

This experiment illustrates the ability of our framework to automatically generate a sequence of configurations to perform a given task in the current context (state). The description below will make reference to the top-level algorithm in Figure 5.8 on Page 88, and to the specific steps in that algorithm. The experiment unfolds as follows.

a. The gas sensors in the **Fridge** trigger an alarm, and post a tuple in the tuple-space. This is notified to the deliberator component in the **HSM**, which decides to send **Pippi** to make an olfactory inspection of the contents of the **Fridge**. **Pippi** gets this task since **Astrid** is currently charging. The deliberator sends a tuple with the high level goal `(smell-fridge)` to **Pippi**.

b. The configuration process located at **Pippi** acquires the current state, and based on this generates an action plan (steps 1 and 2 of the process in Figure 5.8). The plan consists of the actions `(goto Pippi kitchen)`, `(open fridge door)`, `(dock-to Pippi fridge)`, and `(smell Pippi fridge)`. For this action plan, a configuration plan is generated (step 3) based on the acquired state (See Figure 7.17). The configurations in the configuration plan are shown in Figure 7.18.

c. The first configuration in the configuration plan (i.e., the configuration for (goto Pippi kitchen) is selected (step 4 in Figure 5.8). Step 5 and 6 in Figure 5.8 are skipped since for the first action the state is newly acquired and the configuration was generated for this state.

d. This configuration is deployed to the PEIS-Ecology, and execution begins (step 6). While navigating, the `ThinkingCap` component in **Pippi** receives position updates from the **Tracker** in the form of a stream of tuples (first row in Figure 7.18).

```
( (peis Pippi) (peis Astrid) (peis HSM) (peis Fridge)
  (room kitchen) (room living-room) (room bedroom)
  (connects Door1 living-room kitchen)
  (connects Door2 living-room bedroom)
  (in Pippi = living-room) (at Pippi = entrance)
  (in Astrid = living-room) (at Astrid = charger)
  (docked Astrid charger = t)
  (funct Pippi camera id10) (cost Pippi id10 40 on)
  (funct Pippi object-tracker id11) (cost Pippi id11 130 on)
  (funct Pippi odometer id12) (cost Pippi id12 100 on)
  (funct Pippi TC id13) (cost Pippi id13 20 on)
  (funct Pippi e-nose id14) (cost Pippi id14 50 on)
  (funct Pippi odor-classifier id15) (cost Pippi id15 40 on)
  (funct Pippi wheel-actuators id16) (cost Pippi id16 60 on)
  (funct Fridge door id21) (cost Fridge id21 40 on)
  (funct Fridge rfid-reader id22) (cost Fridge id22 35 off)
  (funct Tracker cameras id31) (cost Tracker id31 20 on)
  (funct Tracker tracking-system id32) (cost Tracker id32 20 on)
  (funct Astrid camera id41) (cost Astrid id41 100 on)
  (funct Astrid laser id42) (cost Astrid id42 100 off)
  (funct Astrid odometer id43) (cost Astrid id43 10 on)
  (funct Astrid TC id44) (cost Astrid id44 20 on)
  (funct Astrid e-nose id49) (cost Astrid id49 50 on)
  (funct Astrid object-tracker id45) (cost Astrid id45 130 on)
  (funct Astrid self-localization id46) (cost Astrid id46 20 on)
  (funct Astrid wheel-actuators id47) (cost Astrid id47 55 on) )
```

Figure 7.17: The state used in Experiment 3 (partial).

e. When the `ThinkingCap` establishes that **Pippi** has reached the kitchen, it posts a `done` tuple. This is notified to the configuration process (step 8), which then (step 4) selects the next configuration in the configuration plan (the (open fridge door) configuration in Figure 7.18).

f. The current state is again acquired (step 5), and the configuration is verified with the state (step 6). The new configuration is deployed (step 7) and executed (step 8).

g. When the `open-fridge-door` component signals termination, the next configuration for action (dock-to Pippi fridge) is selected (third row in Figure 7.18). In this configuration, **Pippi** uses the on-board camera and a vision system to track the **Fridge** edges. The current state is acquired (step 5), the configuration is verified, deployed and executed.

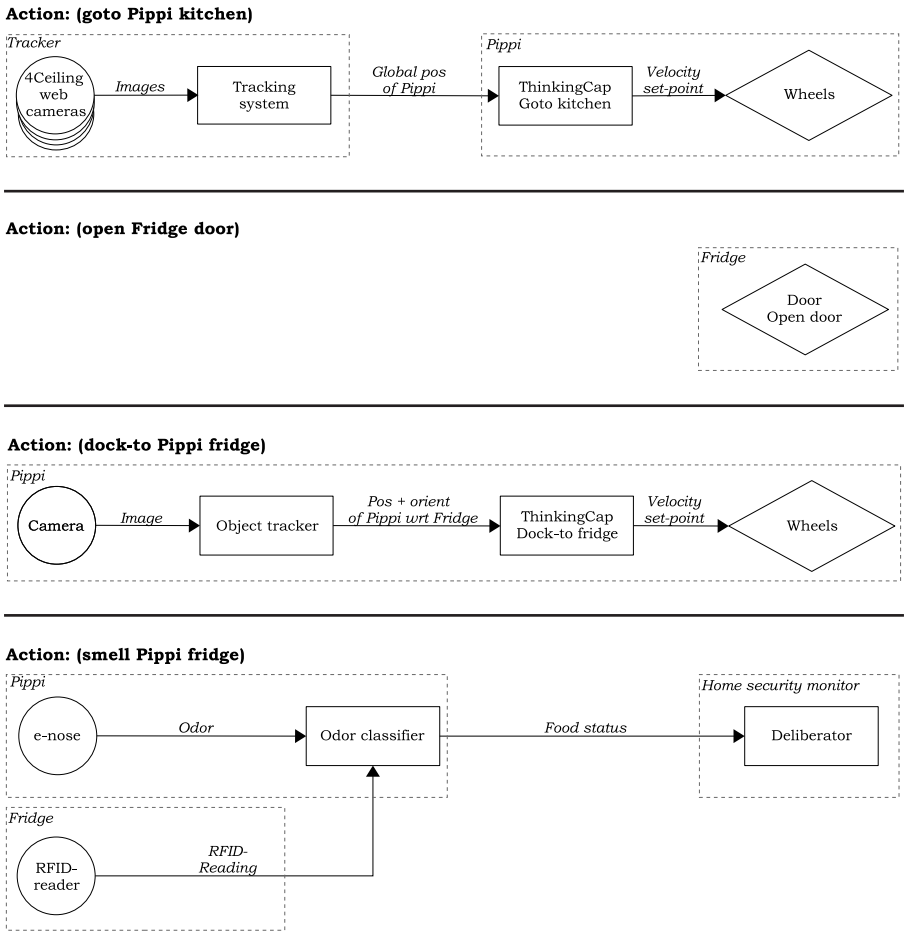


Figure 7.18: The PEIS-Ecology configurations (in simplified form) generated for the actions: (goto Pippi kitchen), (open fridge door), (dock-to Pippi fridge), and (smell Pippi fridge).

h. When the action is completed, the next configuration is selected ((smell Pippi fridge) configuration in Figure 7.18), verified with the current state and then deployed. In this configuration, the `odor-classifier` on **Pippi** requires context information about the objects being smelled, which can be obtained from the RFID tags on the items in the **Fridge**.

i. Finally, the `odor-classifier` in the configuration sends the classification results to the deliberator in **HSM** and signals termination (step 8). Since there are no more actions, the execution of the top level task completes (step 4).

Based on the classification results, the deliberator may decide to start a new task, e.g., to ask **Pippi** to move to the bedroom and alert the occupants of the PEIS-Home. This task would be done in a way similar to the one above.

During the generation of the action plan, partial plans in which **Astrid** performs the steps are also considered. However, since **Astrid** is at the charger there is an initial extra step (undock astrid charger) in all these partial plans that makes them more expensive than the plans with **Pippi**.

In step b, when the configuration plan is generated, several alternatives are considered for each action, before the configuration with the lowest cost is added to the configuration plan. For the first action (goto pippi kitchen) there are three possible ways for the `ThinkingCap` component in **Pippi** to obtain self-position information: (1) from the odometry on **Pippi**; (2) from the tracking system on the **Tracker**, which can track the position of **Pippi** using the ceiling cameras; or (3) from **Astrid**, which can track the position of **Pippi** using its own camera. The configuration that uses the second alternative is added to the configuration plan since it has the lowest cost. For the (open fridge door), the only configuration which achieves this is the one that consists of the single `open-fridge-door` component in the **Fridge**, which does not require any information. A smarter component could use information about free space in front of the **Fridge**, obtained, for instance, from the ceiling cameras: accordingly, the configuration planner would include these components in the configuration. In order to perform the (dock-to Pippi fridge) action, **Pippi** needs to know the location of the **Fridge** with respect to herself. This information can be obtained by using a camera on-board **Pippi** and a vision system to track the **Fridge** edges. For the last action (smell Pippi fridge), the RFID-reader inside the **Fridge** is used to read the tags of the items in the **Fridge**, but it is also possible to use the RFID-reader on **Pippi** to read the tags.

Results

The aim of the above experiment was to illustrate that our framework is able to handle tasks that require multiple steps.

Figure 7.19 shows the temporal diagram for the experiment. The sequence of actions is clearly visible. The high-level task was decomposed into four ac-

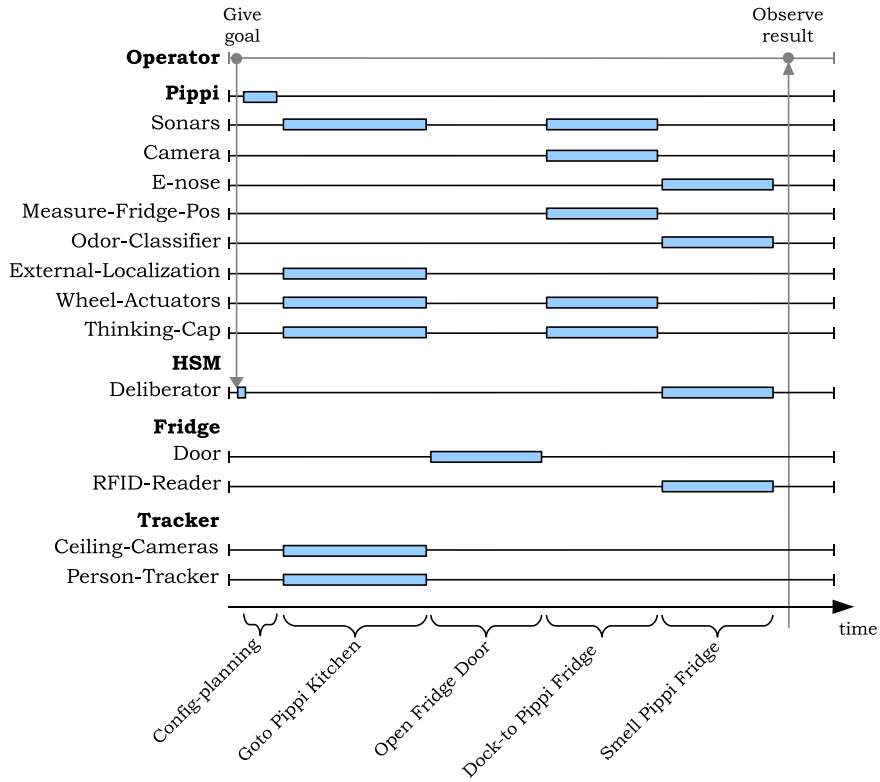


Figure 7.19: A temporal diagram of the execution of the "smell fridge" experiment that shows which functionalities are active and when during the execution of the actions. Only the PEIS and functionalities used during the experiments are shown.

tions, and the system self-configured to perform each action in turn. Notice that, compared with the diagram in Figure 7.13 (Experiment 1) now there is no need to provide manually a new goal for each subtask (action) of the full task. Also, there is no need to invoke the configurator again when starting a new action, since all configurations in the sequence are generated in the initial planning. Some actions required tight coordination between robots: e.g., in the first action, the **Tracker** was sending a continuous stream of position information to **Pippi**. No human intervention was required at any time between the initial assignment of the goal and the final verification of the results. The experiment successfully meets its aim.

7.4.2 Experiment 4: Smell Inside the Fridge — Different Situation

Question

Can our approach generate different configurations to cope with different tasks and conditions?

Setup

The fourth experiment reproduces the third one, with the only difference that **Pippi** was removed from the PEIS-Ecology. As a consequence, in step (a) the deliberator assigns the task (smell-fridge) to **Astrid** instead of **Pippi**, and the configuration process was executed on **Astrid**. Note that **Astrid** has a different sensor suite and a different starting position than **Pippi**. These facts are automatically acquired during the state acquisition phase: no manual adjustments were made between the third and the fourth experiment.

Astrid successfully completed the task. The execution unfolded as in Experiment 3 replacing "Pippi" by "Astrid", with the following three exceptions:

- The state acquired by **Astrid** was a subset of the previous one (see Figure 7.17), in which all the literals associated to **Pippi** were not present.
- There is an extra action (undock astrid charger) at the beginning of the action plan generated for Experiment 3.
- The `ThinkingCap` component on **Astrid** can obtain position information in two ways: (1) from the scan-matching self-localization system on **Astrid**, which takes laser data plus odometry as input; or (2) from the tracking system on the **Tracker**. Correspondingly, the set of possible configurations to perform the action (goto Astrid kitchen) is different from the previous case. In our experiment, the configuration planner selected the first option, shown in Fig. 7.20, since this had the lowest cost.

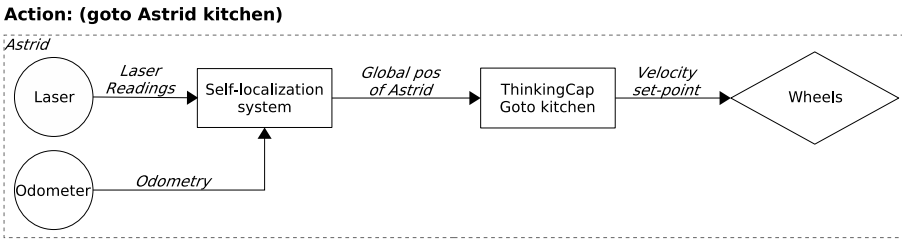


Figure 7.20: Configuration for the action (goto Astrid kitchen) performed by **Astrid**, using a laser and a scan-matching algorithm.

Results

The aim of the above experiment was to illustrate that our framework is able not only to handle tasks that require multiple steps, but also different situations and tasks.

In this experiment, the same task as in Experiment 3 was successfully accomplished in a situation in which both the world state and the system state were different. The only manual interventions were to switch off the first robot (**Pippi**). This experiment met its aim successfully.

7.4.3 Experiment 5: Smell Inside the Fridge — Respond to Failure by Re-Configuration

The fifth experiment is a variant of the third one, in which a component fails during execution.

Question

Can our approach dynamically re-configure the system in response to a failure of some of its components while executing a sequence of configurations?

Setup

The same PEIS are available as in Experiment 3, i.e, **Pippi**, **Astrid**, the **Tracker**, the **Fridge**, the **Lamp**, the **ALS** (Ambient Light Sensor), and the **HSM** (Home Security Monitor). At start-up, **Pippi** is located close to the entrance, and **Astrid** is charging her batteries at the charging station in the living-room.

Execution

Steps (a) to (d) proceed as in Experiment 3: **Pippi** is given the top-level task (`smell-fridge`), generates an action plan, a configuration plan, and starts to execute the configuration for the first action (goto Pippi kitchen). However, during step (d) a failure occurs. Execution then proceeds as follows.

(d') The operator simulates a failure of the `object-tracker` functionality on the **Tracker** by posting a `fail` tuple. This is received by the configuration process (step 8 in Figure 5.8), which halts the execution of the current configuration.

(d'') The current state of the system is acquired excluding the component which has failed, and the verification of the configurations leads to a search for a new configuration (step 5, 6, 3). This time the only available options to provide position information to the `ThinkingCap` in **Pippi** are: (1) from odometry on **Pippi**; or (2) from visual tracking on **Astrid**. The configuration planner selects the second option, shown in Figure 7.21, based on its cost/reliability. The postconditions of this new configuration are equal to the postconditions of the original configuration, and thus it can be safely added to the configuration plan.

(d''') The new configuration is deployed, and it is executed cooperatively by **Pippi** and **Astrid** (steps 7–8).

The rest of the experiment (steps e–j) proceeded as in Experiment 3.

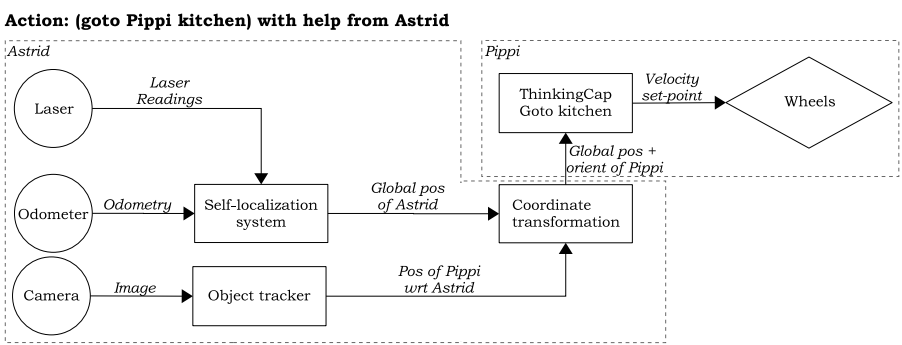


Figure 7.21: A configuration for action (goto Pippi kitchen). **Pippi** gets help from **Astrid**.

Results

The aim of the above experiment was to illustrate that our framework is able to handle failures that require re-configuration during the execution of tasks that require multiple steps.

Experiment 5 replicated Experiment 3, but a functionality failure was forced during execution. The temporal diagram in Figure 7.22 shows how the experiment progressed and which functionalities were used for the different actions. The system autonomously re-configured and continued the task until it was successfully accomplished. Notice how, after the failure, **Astrid** has replaced **Tracker** as a helper of **Pippi**. No manual intervention was required except for the injection of the faulty condition. The aim of this experiment was successfully met.

7.4.4 Experiment 6: Smell Inside the Fridge — Respond to Failure by Re-Planning

In Experiment 5, the failure could be addressed by generating an alternative configuration for the same action. This was possible in that case since there were alternative ways to solve the action (goto Pippi kitchen). In this experiment a failure during (dock-to Pippi fridge) action is considered. In this case, there are no alternative configurations for this action and the full action plan needs to be reconsidered.

Question

Can our approach dynamically replan when a reconfiguration is not possible as a response to a component failure?

The difference between this question and the question for Experiment 5 may need to be clarified. Algorithm 5.3 on Page 90 details how the top-level process recovers from failures. The top-level process first tries to reconfigure by finding an alternative configuration for the current action. If there is no alternative configuration, the top-level process instead tries to find an alternative configuration plan that is able to reach the goal state. The question for Experiment 5 concerns the situation when there is an alternative configuration, and the question for this experiment concerns the situation when there is none, but an alternative configuration plan exists. Thus, this experiment was performed to illustrate how a failure can be handled at the action plan level.

Setup

This experiment uses the same setup as Experiment 3 and Experiment 5.

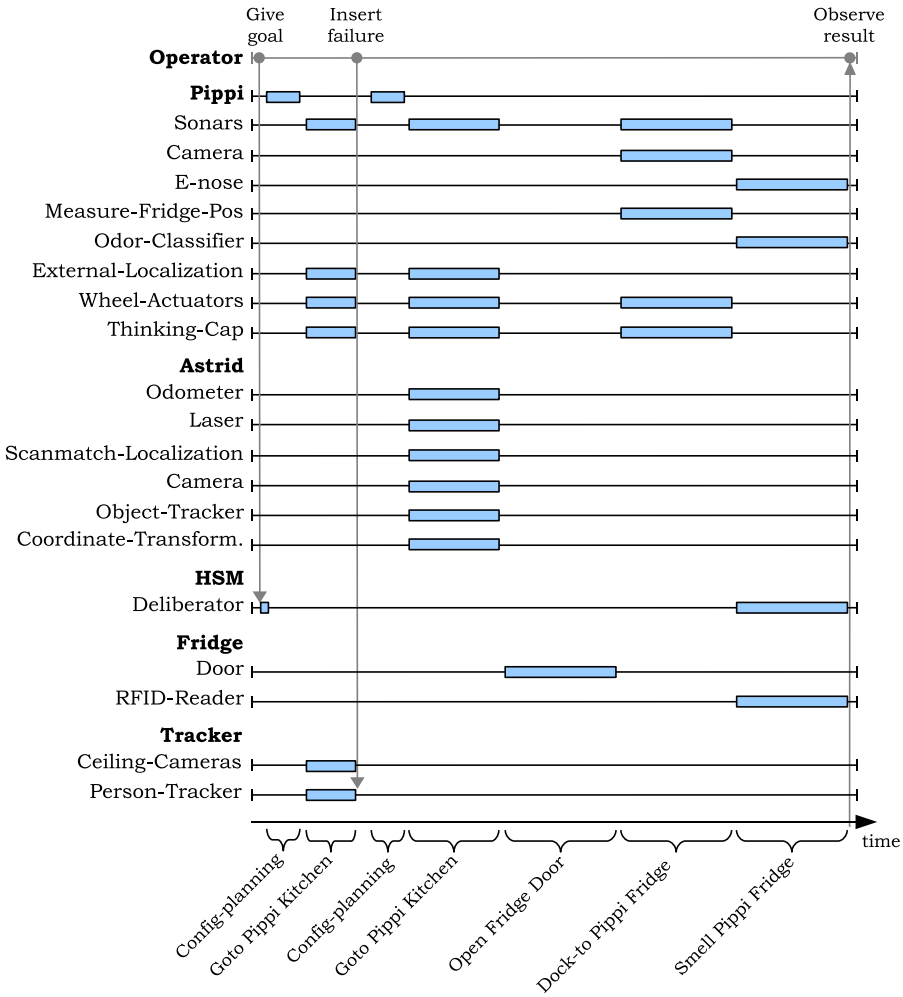


Figure 7.22: A temporal diagram of the "smell fridge" experiment with re-configuration after a failure. Notice that after the reconfiguration **Astrid** is used instead of the **Tracker**.

Execution

The steps (a) to (g) proceed as in Experiment 3, and during step (g) a failure occurs. Execution then proceeds as follows.

(g') The `measure-fridge-pos` component on **Pippi** is unable to track the **Fridge** because of poor lighting conditions (the operator manually turned off the **Lamp** in the kitchen), and it posts a `fail` tuple. This is received by the configuration process (step 8 in Figure 5.8), which halts the execution of the current configuration.

(g'') The current state of the system is acquired excluding the component which has failed, and the verification of the configurations leads to a search for a new configuration (steps 5, 6, 3). Since there are no alternative configurations, the action planner (step 2) is called to find an alternative action plan. In this action plan, the action (turn-on Lamp kitchen) is included before the (dock-to Pippi fridge) action. The new action plan is: (turn-on Lamp kitchen), (dock-to Pippi fridge), and (smell Pippi fridge). A new configuration plan is generated for this action plan (step 3 in Figure 5.8). When the configuration planner generates the configuration for the (dock-to Pippi fridge) action it again considers the configuration in which **Pippi** measures the **Fridge** position using the camera. The `object-tracking` functionality is not excluded from the configuration as it was in Experiment 5 since the state it failed in is different from the state it is now considered for. With the light on in the kitchen it is possible to use the `object-tracking` functionality, i.e., the `object-tracking` functionality has light as a precondition.

(g''') The first configuration for action (turn-on Lamp kitchen) in the configuration plan is executed.

The remaining part of the configuration plan and the experiment proceeded as in Experiment 1 (steps g–i).

Results

The aim of the above experiment was to illustrate that our framework is able to handle not only failures that require re-configuration, but also failures that require re-planning.

This experiment also replicated Experiment 3. However, this time a component failure was forced for which no re-configuration was possible (i.e., no alternative configuration existed). The system reconsidered the action plan and a new action and configuration plan was generated which had an additional action that made the original configuration admissible again. The system used this new plan to complete the task and the aim of the experiment was successfully met.

Figure 7.23 gives the temporal evolution of the performed experiment. Notice how the Dockto action has been interrupted by the failure, and that the execution of the Turnon action has been inserted in the plan.

7.4.5 Experiment 7: Fetch Book

The seventh experiment was used as the final demonstration of the PEIS-Ecology project [Saffiotti et al., 2008]. The PEIS-Ecology project was a four-year collaborative research project between Sweden and Korea for the development of the overall concept and technology of the "PEIS-Ecology". The work reported in this thesis was one of the mayor contribution to this project.

Question

Can the configuration framework be integrated in a real PEIS*-Ecology composed of many different types of devices of different complexity?*

Setup

The experiment was run inside the PEIS-Home demonstrator. The following PEIS were used: **Astrid**, **Tracker**, **Table**, **Lamp**, and **HSM**.

Execution

The experiment unrolls as follows. Snapshots from some of the steps are shown in Figure 7.24.

a. Alex is lying in bed with a broken leg, and he asks the PEIS-Ecology to bring him a given book using a mobile phone interface (Figure 7.24-1). The request is received by the **HSM**, which forwards the task to **Astrid**.

b. **Astrid** queries the tuple-space to retrieve the current state of the ecology, including the position of the book. It finds the tuple giving `table` for the book position, published by the **Table** which had previously detected the RFID tag of the book.

c. **Astrid** generates the action plan:[3] (turnon light), (goto table), (anchor book1), (grasp book1), (goto bed), (deliver book). The first action is needed for reliable anchoring of the book. With this action plan, **Astrid** generates a configuration plan.

d. **Astrid** executes the first configuration in the configuration plan for the action (turnon light); this configuration only involves activating the **Light** PEIS with the parameter `on`.

[3]The plan has been slightly condensed for explanation purposes.

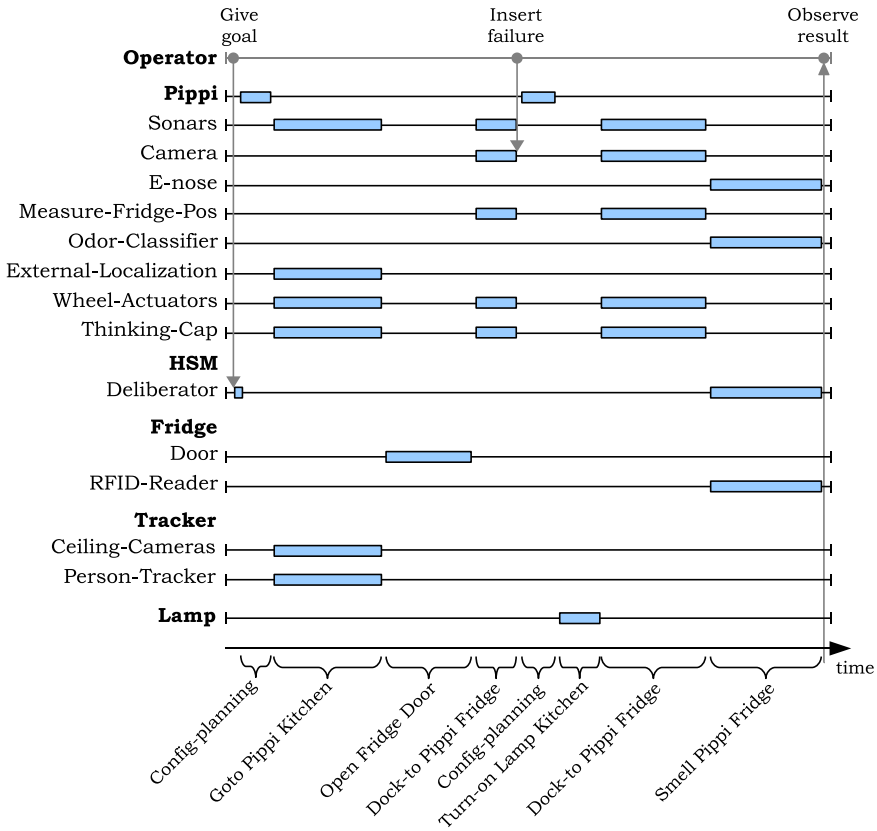


Figure 7.23: A temporal diagram of the execution of the "smell fridge" experiment with full re-planning after a failure. Notice the new action "Turn-on" inserted by re-planning.

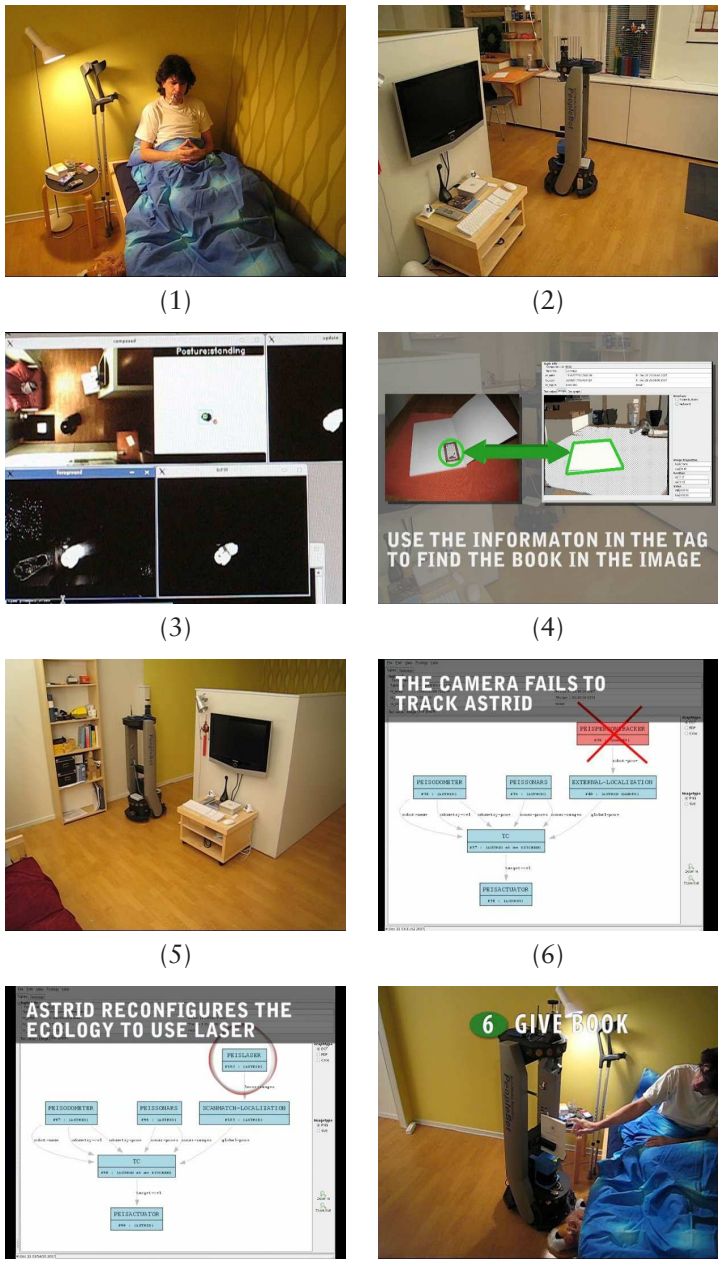


Figure 7.24: Some snapshots from the execution of the "fetch book" experiment.

e. **Astrid** executes the second configuration in the configuration plan for the action (goto table) (Figure 7.24-2); this configuration includes the activation of the **Tracker** and the connection of its output to **Astrid**'s navigation component. Figure 7.24-3 shows how the **Tracker** tracks **Astrid**.

f. **Astrid** executes the third configuration in the configuration plan for the action (anchor book1). That is, locking visually the object book1. This includes the **Table**, which reads the properties in the book's RFID tag (e.g., the color of its cover) and **Astrid**'s camera, in order to identify which of the objects on the table is `book1`. (Figure 7.24-4)

g. **Astrid** grasps the book (here, with the help of a human because of the limited manipulation abilities of the PeopleBot) and takes the configuration for (goto bed), again involving the output from the **Tracker**.

h. When **Astrid** enters the bedroom (Figure 7.24-5), the **Tracker** fails to track it because of its limited field of view and it sends a `fail` tuple (Figure 7.24-6). **Astrid** then searches an alternative configuration for the action (goto bed), and finds one using its on-board laser plus scan-matcher for self-localization (Figure 7.24-7).[4]

i. **Astrid** reaches the bed and delivers the book (Figure 7.24-8).

Results

This experiment constituted the final demonstrator of the larger PEIS-Ecology-project, delivered on December 15, 2007. A video of this experiment can be accessed from the project website [PEIS Ecology Project, 2009]. The aim of this experiment was to illustrate that the configuration framework can be integrated in a real PEIS-Ecology composed of many different types of of devices of different complexity.

In the task of delivering the book to Alessandro, the configuration framework used a wide variety of different PEIS of different complexity in the PEIS-Ecology, e.g., **Lamp**, **Table**, **Tracker**, and **Astrid**. By combining the functionalities of these and other PEIS, the configuration framework is able to deliver the correct book to Alessandro. Since the task was successfully completed, the aim of the experiment was also successfully met.

7.5 Experiments on Parallel Configurations

In the previous experiments, the configurations were executed in sequence or there has only been one configuration to execute. The last experiment in this Chapter aims to illustrate our approach to parallelize the execution of some configurations in the plan. In contrast to the previous experiments, which were

[4]The laser was not selected in the first place because of its greater cost in terms of energy.

performed on real robots, the experiment on parallel configurations was performed in the simulated version of the PEIS-Home. The simulator was chosen since it currently provides a richer set of robotic devices that can manipulate the environment than what is currently available in the real PEIS-Home testbed.

7.5.1 Experiment 8: Get the Milk

To demonstrate the parallelization of actions, we of course consider a task in which it possible to execute several actions in parallel. The task is simple, but it includes several different robotic devices that can manipulate the environment.

Question

Can parallelization of actions and merging of configurations be used to reduce the execution time of a configuration plan while still achieving the target task?

As in the previous experiments, we do not want to a perform systematic empirical evaluation to show that this is always the case, and we do not want to make a quantitative evaluation to measure how much we can gain. As in all the experiments, our question is only qualitative, i.e., to show that there are cases when the answer to the above question is yes.

Setup

The experiment was run in the simulated PEIS-Home demonstrator. The following PEIS were used: **Astrid**, **Tracker**, **Fridge** and **HSM**. In the simulated version of the **Fridge** there is a 2 DOF manipulator that can pick up items inside the fridge and place them on a robot docked to the **Fridge**. To perform an experiment in the simulated PEIS-Home and the real PEIS-Home is very similar. The same PEIS middleware and PEIS-components are used as in the physical experiments, but they are connected to the simulated devices rather than the real ones. Hence, from the point of our configuration process, there is no difference — exactly the same code would be use for the real one.

Execution

The experiment unfolds as follows. The description below makes references to the execution of the top-level process given in Figure 6.7 on Page 113, and its steps. Figure 7.25 shows some snapshots from the experiment.

a. At start up, **Astrid** is located in the Living-room, awaiting to be assigned a task. One of the persons in the apartment asks the PEIS-Home for some milk. **Astrid** is assigned the task to deliver the milk to the person sitting in the living-room sofa.

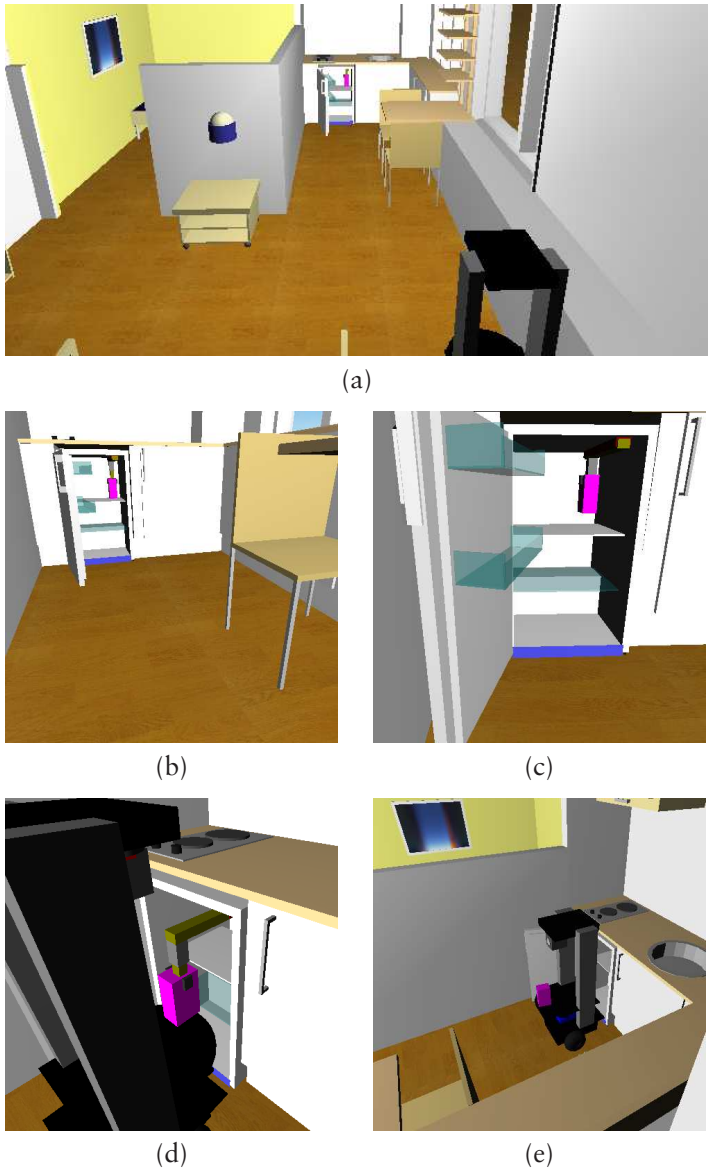


Figure 7.25: Some snapshots from the execution of the "get the milk" experiment. (a) and (b) **Astrid** starts to navigate to the kitchen and simultaneously the **Fridge** door is opening and the **Fridge** manipulator is gripping the milk-box. (c) The (open fridge door) and (pick-up fridge milk) actions has finished. (d) **Astrid** is given the milk by the **Fridge** manipulator. (e) **Astrid** with the milk executing the (undock astrid fridge) action.

b. **Astrid** acquires the current state (step 1 in Figure 6.7 on Page 113) and generates an action plan (step 2) with the following actions: (goto astrid kitchen), (open fridge door), (dock-to astrid fridge), (pick-up fridge milk), (deliver fridge milk), (undock astrid fridge), (close fridge door), and (goto astrid living-room). With this action plan a configuration plan is generated to confirm that it is possible to execute the task (step 3).

c. **Astrid** takes the first action (goto astrid kitchen) in the action plan (step 4) and verifies that the configuration for this action is still admissible (step 6).

d. The configuration is valid and as a next step **Astrid** tries to find actions in the plan that can be executed in parallel with the action (goto astrid kitchen). The algorithm to find parallel actions (Algorithm 6.2 on Page 107) returns the following actions: (goto astrid kitchen), (pick-up fridge milk), and (open fridge door). Recall that the fridge door opens autonomously and the manipulator is in the **Fridge** and not on the robot.

e. The different configurations for the actions are combined and a merged configuration is found (step 8). The merged configuration (See Figure 7.26) is then deployed to the PEIS-Ecology and the execution starts.

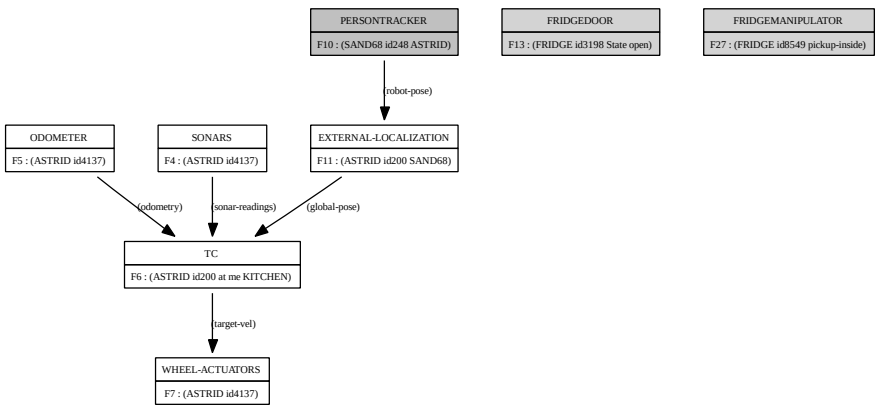


Figure 7.26: A merged configuration for the actions (goto Astrid kitchen), (open Fridge door) and (pick-up Fridge milk).

f. The (open fridge door) action finishes first and the merged configuration is reduced (step 11) so that the parts of the configuration that belong to the finished action are removed. The top-level process on **Astrid** then takes the next action in the plan (step 4) according to Algorithm 6.4.

Since there are currently two actions running, they are returned as the next actions. The state is again acquired and the action plan is searched for actions that can be executed in parallel with (goto astrid kitchen) and (pick-up fridge milk). No other actions can be executed.

g. The next action to finish is the (pick-up fridge milk) action. As in the previous step, the configuration is reduced such that all parts that belong to the finished action are removed. The same steps are taken and no other actions are found that can be executed in parallel with action (goto astrid kitchen).

h. **Astrid** finally reaches the kitchen and the entire configuration has now finished. The next action in the plan is (dock-to astrid fridge). No other actions can run concurrently with this action. The configuration is retrieved and verified with the newly acquired state. **Astrid** executes the configuration.

i. At the **Fridge**, the configuration for delivering the milk to the robot is executed.

j. With the milk on board, **Astrid** executes the (undock astrid fridge) action.

k. When **Astrid** is not docked to the **Fridge** any more, the remaining actions (close astrid fridge) and (goto **Astrid** living-room) are executed simultaneously and the task is completed.

In order to compare the approach where the configurations are parallelized with the approach in which configurations are executed in sequence, we have also performed the same experiment using the latter approach.

Results

The aim of this experiment was to show a concrete case in which parallelization of actions and merging of configurations can reduce the execution time of a configuration plan. Further, the aim was to give a "proof of concept" of the approach and to show in practice how the parallelization of actions and merging of configurations work.

The temporal diagram in Figure 7.27 summarizes the experiment. **Astrid** generated an action plan and a configuration plan to make sure that the task was executable. The action plan was then parallelized such that the actions (goto astrid kitchen), (open fridge door), and (pick-up fridge milk) could be executed concurrently. The actions finished one by one and the merged configuration was reduced each time. The actions in the middle of the plan were executed in sequence, but the two final actions (close astrid fridge) and (goto astrid living-room) were executed in parallel. The total execution time of the task was 230 seconds.

The temporal diagram in Figure 7.28 shows how the task execution proceeds in the case when actions/configurations are not executed in parallel. Comparing the two diagrams gives the obvious result; to parallelize a configuration plan can reduce the execution time. In this case, the total execution time was 270 seconds when configurations were executed in sequence compared to 230 seconds for parallel execution. It is also important to emphasize that the same final goal was reached (i.e., the milk was delivered) in both runs. The aim of experiment was successfully met.

It should be mentioned that the above experiment is very simple and does not fully explore the capabilities of the configuration merging approach. That is, all the merged configurations are composed of actions that are obviously parallelizable since they do not share any resources or functionalities. In Section 6.2.3, an example was given in which the merging of configurations is less obvious.

7.6 Discussion

The goal of the experiments presented in this chapter was twofold. First, to give the reader a better understanding of the different techniques presented in this thesis by providing concrete examples of how they work in practice. Second, to show their applicability to a real, fully implemented network robot system. Have these goals been achieved?

In Section 7.3 we reported two experiments that illustrate how the configuration generation framework, described in Chapter 4, works in practice. These two experiments demonstrated the applicability of single configuration generation to a real distributed robot system.

In Section 7.4 we reported five experiments that illustrate the loosely coupled action and configuration planning approach described in Chapter 5. These experiments demonstrated that the approach can handle: tasks that require multiple steps, different situations and tasks, failures that require re-configuration, and failures that requires re-planning. The last experiment also showed that this approach can successfully be incorporated as part of a complete system.

In Section 7.5 we reported an experiment that illustrates the techniques to merge and reduce configurations described in Chapter 6. The experiment showed that execution time can be reduced by parallelization of actions using these techniques.

The above experiments demonstrate that the different techniques used in our approach are applicable to a real distributed system. We also believe that they adequately illustrate how these different techniques works in practice. Thus, we consider that the aims of the experiments have been met.

In these experiments we have not attempted any quantitative evaluation of our approach. A quantitative evaluation is not adequate for our approach since this would measure the performance on the full system (which depends

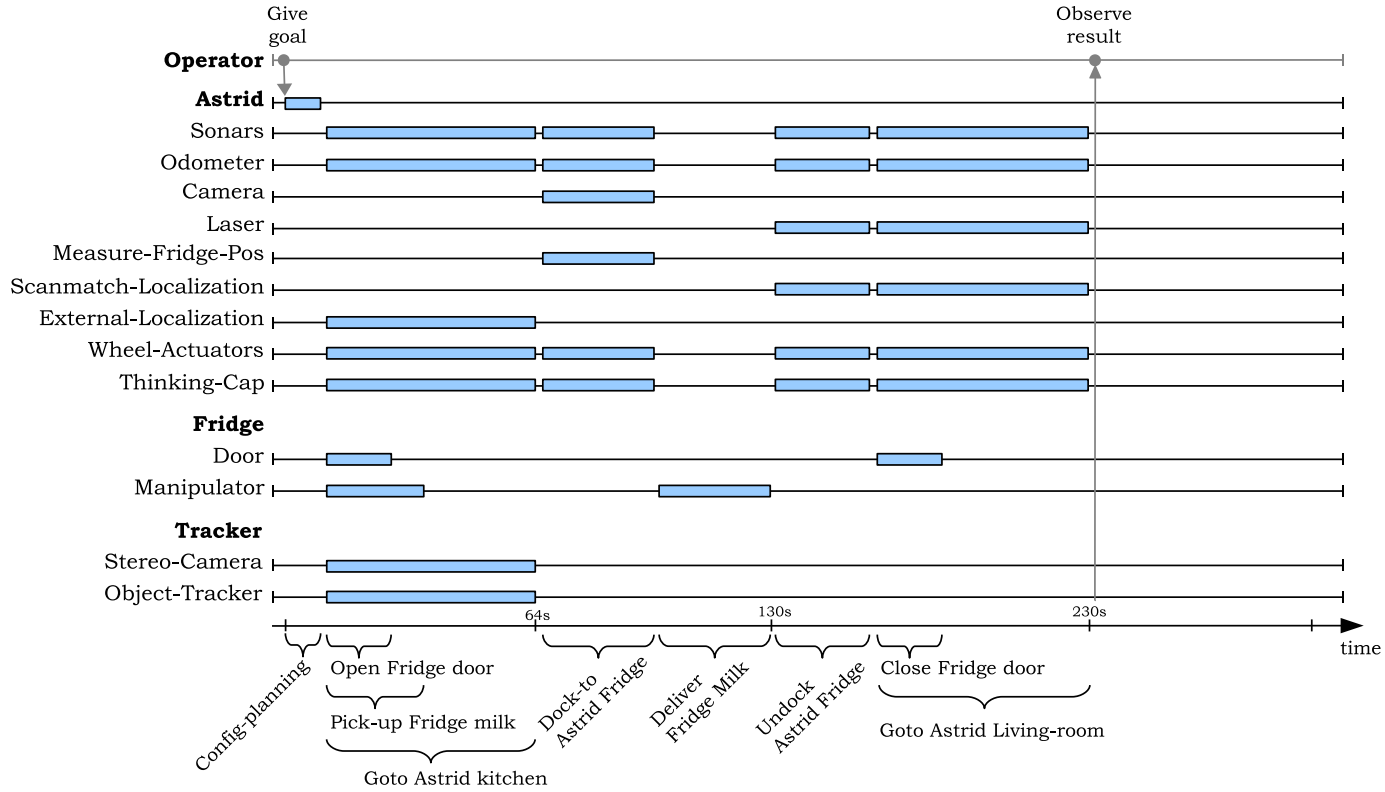


Figure 7.27: A temporal diagram of the execution of the "get milk" experiment, in the case where actions can be executed in parallel.

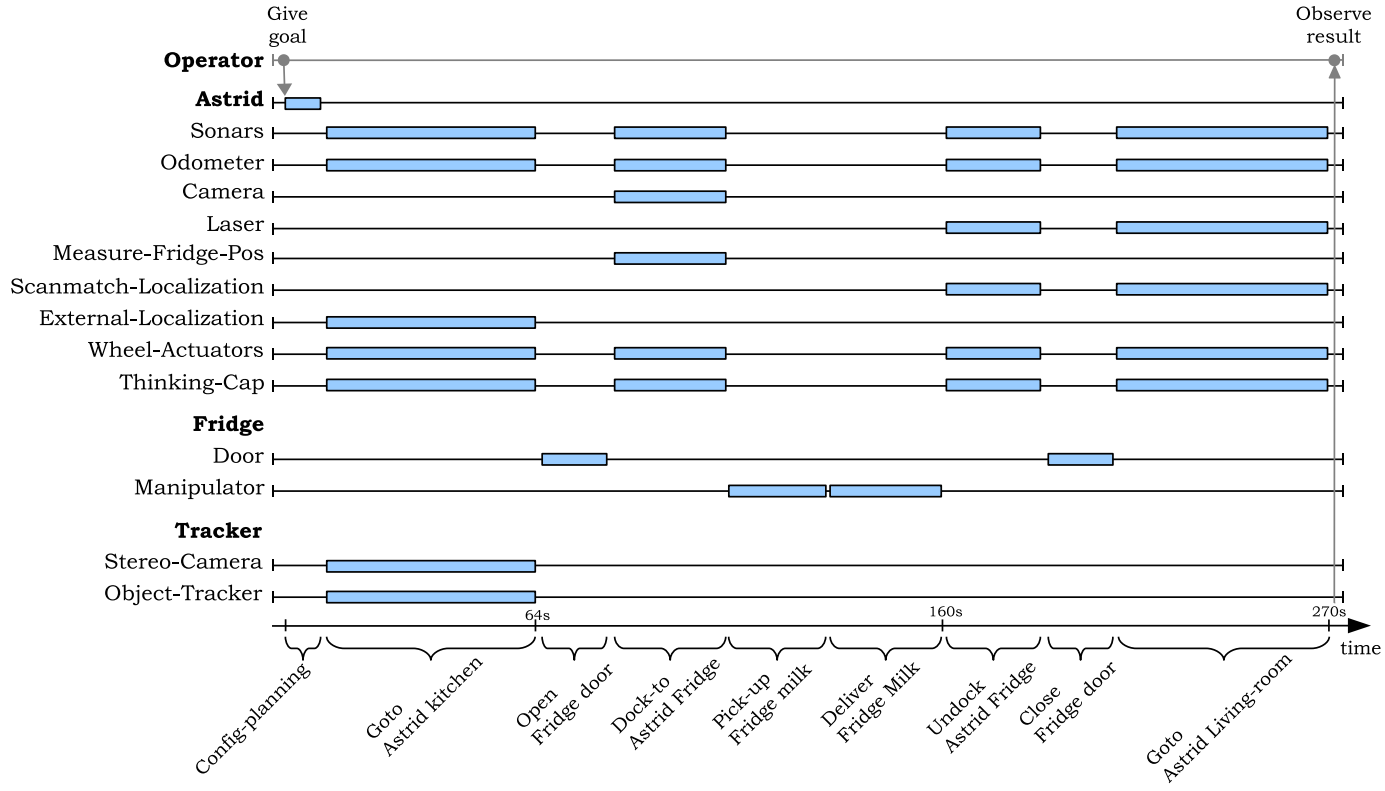


Figure 7.28: A temporal diagram of the execution of the "get milk" experiment. No parallelization of the actions was performed.

on the quality of the implemented functionalities), and not the performance of the configuration framework.

There are a few aspects of our approach that have not been tested yet, and that would be interesting to test in the future, e.g., scalability and portability. On scalability, it would be interesting to see how the approach can handle more complex environments and tasks, with more robots and functionalities available. It would also be interesting to test the approach to parallelize configuration plans with tasks that involve configurations that are more complex to merge and reduce. Regarding portability, in this thesis we have only tested the approach on the PEIS-Ecology-testbed. We believe that our approach would work on any multi robot or network robot system as long as they comply with the criteria stated in Section 7.2. Clearly, it would be interesting to test the approach on different network robot systems or multi robot systems to validate this statement.

Chapter 8
Conclusions

In this final chapter we discuss the contributions of the thesis, the limitations of the approach, and directions of future work.

8.1 What Has Been Achieved?

The approach presented in this thesis gives a distributed robot system the ability to self-configure to perform a given task in the current situation. This approach combines functionalities, residing in different robots, into a functional configuration in which the robots cooperate to perform the task. Another way to look at our approach is that a virtual robot is built on the fly by combining software and hardware components residing on different robots, in order to collectively perform a given task in a given situation. In order to automatically generate functional configurations we use a technique inspired by Artificial Intelligence planning, in particular hierarchical planning.

A task may require several steps to be accomplished and each step requires its own functional configuration. For this purpose we employ two distinct planners: one for generating the steps required for the task (i.e., the action planning) and another one for generating the configurations that implement each step. The planner for generating configurations and the action planner are loosely integrated, i.e., configuration planning is used to validate and correct action plans. With this integration, it is possible to guarantee that the execution of a task is not started if there is no admissible configuration plan. In other words, it is possible to know *before hand* if a plan is executable or not. To use loose integration also makes it easy to replace the current planners with other plan generation techniques if this is desirable.

As expected with a centralized plan-based approach, our approach has good formal properties. We have shown that the algorithm to generate configurations is sound, complete, and optimal. That is, only admissible configurations are generated, all solutions are considered and the best configuration is returned. We have also shown that the loose coupling of the planner for configurations

and the planner for actions is sound and complete (provided the search is done as described in Section 5.4.6). That is, the loosely coupled approach only generates admissible configuration plans and eventually generates all action plans that achieve goal G defined by the task.

It is not necessarily the case that the actions/configurations in the configuration plan must be executed in sequence, one after the other; some actions may be causally independent of each other and their corresponding configurations may be run concurrently. In order to facilitate concurrent execution of configurations, we have proposed an approach in which the configurations are merged. If the configurations are compatible, a new *merged* configuration is created. Examples of applications for configuration merging are: parallelization of a configuration plan for reducing execution time (as demonstrated in this thesis), and guaranteeing safe execution of multiple configurations generated by the same or different configuration processes.

The approach has been implemented in different versions and tested on a real distributed robot system to demonstrate its viability. The entire self-configuration process (See Figure 4.13, Figure 5.8, Figure 6.7 for the different versions), from state acquisition to a possible failure repair, is performed in real time without any manual intervention. Recovery from failure is performed on two levels. First the top-level process tries to reconfigure by finding an alternative configuration for the current action. If there is no alternative configuration, the top-level process instead tries to find an alternative configuration plan that is able to reach the goal state.

8.1.1 Contributions

The main contributions of this thesis can thus be summarized as follows:

- A formal definition of functional configurations and their components, that is suitable for describing functionalities and their interconnections.
- An approach to automatically generate configurations, or sequences of configurations, for a distributed robot system for a given task, environment, and set of resources, using knowledge-based techniques. This is a novel contribution. In particular, the problem of generating sequences of configurations has never been addressed before as far as we know.
- Methods for merging and reducing configurations which allows safe execution of multiple configurations concurrently. These are novel contributions.
- A complete framing of the configuration generation process that is adequate for applying the approach in a real distributed robot system. This includes dynamic state acquisition, automatic configuration deployment, configuration execution and monitoring. To our knowledge, no similar

approach that considers all these aspects has been described in the literature.

- Methods for reconfiguring and replanning that allow the distributed robot system to recover from functionality failures. The ability to either reconfigure or to replan in order to find alternative solutions are novel contributions.

The main contributions above have been complemented with:

- An analysis of the formal properties (soundness, completeness, and optimality) of the approach.
- A working implementation of the approach on a real distributed robot system that has been tested to demonstrate the viability of the approach.
- A broad, although not exhaustive, survey of the research on related systems and problems.

Finally, the approach presented in this thesis has been developed in part within a larger cooperation project, and it has been incorporated in several full system demonstrations within that project.

8.1.2 Applicability

Even though our approach has only been implemented and tested on one specific distributed robot system (the PEIS-Ecology), we believe that this approach can be applied in general to any multi robot system, or even any multi-agent system, as long as the system is able to provide:

1. a way to dynamically acquire the current state of the environment and of the functional components in it,
2. a way to deploy the automatically generated configurations on the functional components that it incorporates, and
3. a way to execute and monitor the configurations to know when configurations have finished or failed.

8.2 Limitations

While our research work has come far on the way to achieve the objectives of the thesis, it still relies on a few important simplifications. In this section we summarize the main limitations. We have divided these limitations into:

Limitations in scope: limitations due to the focus we have chosen for this work, but which are *not* necessarily inherent limitations of our approach.

Limitations of approach: problems that are inherent limitations of the approach.

8.2.1 Limitations in Scope

In the work presented in this thesis, we concentrate on the problem of functional configurations. Obviously, this is just one of several aspects in the work on how to let robots help each other or how to make robots cooperate. Closely related to functional configurations are the problems of task allocation and commitment.

Task allocation typically deals with the question: "Who should do which task?" That is enough for tasks that only require loose coordination. For tasks that require tight cooperation (that is, they cannot be neatly divided among robots or modules), we must address the additional question: "How to execute a task in a cooperative manner?" Configuration generation answers this question and could be integrated with task allocation in different ways. The work by Tang and Parker [2007] suggests one possible way.

Commitment deals with the problem of how to motivate the robots involved in a configuration to commit to it, and how to stay committed during the entire execution. For example, in the first experiment above, we must guarantee that the observing robot does not run away from the scene. For most tasks, robots are motivated by the individual or the team goals. However, if the task is beneficial only to another individual, as in the case of robots that help each other, it may be harder to motivate a commitment.

Although both task allocation and commitment are interesting and important issues, they are not in the scope of this thesis and are not considered in the current approach. Below we present four limitations in scope that we have given moderate attention to in the current approach.

First, we only use a simple estimate of cost to guide our configuration generation to find the best configuration first. To estimate the cost, we use fixed costs for channels and robots and variable costs for functionalities. The cost estimate for a configuration is the sum of the costs for the components used. We do not consider important aspects such as how to obtain a functionality cost that reflects the real cost of the functionality, or performance accuracy and reliability of the configuration. Some or all of these aspects should be included in the cost estimate to make the selection more appropriate. A natural extension, to obtain a more adequate cost value, would be to learn the cost values and/or to use multi-criteria optimatization of the different parameters. Even though we have only used a simple cost function, we have shown that costs can be included in our framework.

Second, we only consider the execution of a single top-level task. In general, several tasks may be performed concurrently, and new tasks may dynamically appear. A natural extension of the current framework would be to use task allocation techniques to assign different tasks to different configuration processes. With such an extension, issues such as resource handling, conflict resolution and deadlocks must also be considered. In this thesis, one important component of a multiple top-level task approach is presented, namely the merging of

configurations (See Chapter 6). By trying to merge two configurations, it is possible to verify that there are enough resources to execute both configurations concurrently and that they are causally compatible.

Third, in the current approach the monitoring of the executing configuration is simple, i.e., we assume that the functionalities themselves are able to detect when they fail and report this to the top-level process. How the functionalities should detect a failure is a complex problem and a research area on its own. Smarter monitoring techniques such as the work by Pettersson [2005], Kaminka et al. [2002] or Li and Parker [2007] could be used to get more adequate monitoring of the configuration execution.

Fourth, in this thesis we consider three types of admissibility for a configuration: information, causal, and resource admissibility. These tell us whether a configuration is admissible in an abstract non-instantiated form, but they do not consider the properties of the physical system that should implement the functionalities and channels. In order to guarantee that a configuration is admissible on a specific system, a fourth type of admissibility would be required: execution admissibility. This type should consider parameters such as bandwidth requirements on channels, frequency of functionalities, and internal memory required.

8.2.2 Limitations of Approach

Our approach also has a few inherent limitations. To overcome these limitations might require substantial changes in the current approach.

First, we use a simple black-box model of a functionality where essentially only information about input, output, preconditions and postconditions of the functionalities is considered. This is the only information given to the configuration process, i.e., the internal state of the functionality is not considered by the configuration process. To consider the internal state of the functionalities could for instance make more complex resource sharing possible. The work by Lyons and Arbib [1989] and Lyons [1990] on representing tasks as networks of concurrent processes, considers a similar formalization where each process (robot schema) also models its internal state (i.e., behavior). Moreover, process algebra is used to analyze the constructed plans (e.g., liveliness). In a similar way, Palamara et al. [2008] and Costelha and Lima [2007] use petri nets for designing, analysis and execution of multi robot systems. In contrast to our work, the connection of processes in the above works is not done dynamically at run-time.

Second, to generate configurations can be a computationally complex problem — the search space grows exponentially in the number of possible functionality instances. This in not an unique problem for configuration generation, rather an inherent limitation for planning problems in general. However, there are methods to reduce the complexity. For instance, if we have many homogeneous robots, the current approach, that always deals with completely

instantiated configurations and methods, may generate a large number of configurations which all are isomorphic relative to the individuals (i.e., robots) involved. Each such instantiation leads to a new branch in the search tree of the planner. Least commitment techniques [Weld, 1994] can be used to address this problem as suggested in [Lundh et al., 2007c].

Third, the current approach to generate configurations and configuration plans is centralized. The information used to generate the configurations is dynamically acquired from the different functionalities in the distributed system, but the generation and monitoring is only performed by one robot or robotic device. Generally, a centralized approach is more vulnerable to failures than a distributed one. That is, if the node that takes all decisions fails, all other parts of the system also fail. If the robot that performs the configuration planning fails, the current task will also fail. This is not a serious limitation in our current approach since it is always the robot that was assigned the task that is performing the planning. However, to have a less centralized approach may be desirable in the future.

8.3 Future Work

The work presented in this thesis is ongoing research and there are several directions for future work. One interesting direction is to investigate how multiple top-level tasks can be allowed simultaneously. The current approach could for instance be coupled with a centralized scheduler [Fratini et al., 2008] (or supervisor [Giacomo and Sardina, 2008, Lacerda and Lima, 2008]) that uses the merging algorithm as a tool for scheduling the execution of the tasks. A more distributed solution would be to use techniques inspired by plan-merging [Alami et al., 1995] and plan management [Joyeux et al., 2009]. To allow several top-level processes would also require a rigorous framework for how robots can lend out their functionalities and commit to different tasks. For instance, if a robot is lending a functionality with actuator capabilities, this may have consequences on the performance and ability of the robot to execute the task it is given, i.e., helping another robot may hinder the helper to accomplish its own task. The merging of configurations studied in this thesis can address parts of this problem.

Another possible direction of future work is to consider semantic knowledge and the use of ontologies for our framework. In the current approach, the connecting of functionalities to each other, in the process of forming a configuration, is performed by syntactically matching the description (`descr`) and domain (`dom`) of the functionalities' inputs and outputs. Organizing the different types of available descriptions and domains in ontologies could allow the inputs and outputs to be connected in a semantic fashion. This would make it easier to use functionalities developed by different software contributors. To use semantic knowledge organized in ontologies is for instance considered in Semantic Web research [The OWL-S Coalition, 2009]. Gritti [2008] presents

the first steps to use ontologies for semantic matching in network robot systems.

Finally, an interesting extension of our framework would be to consider human users as partners in the distributed robot system. Through a proper interface, a human can be seen as just another robot device in the distributed robot system, who can provide and borrow functionalities to/from the other robots. For instance, a walking-aid robot could provide its senior user with lower-level perception and control functionalities, while relying on the humans to provide the higher-level ones. A first step in this direction could be done by using a human-aware planner [Cirillo et al., 2008, Montreuil et al., 2007] for action planning in our approach. To have humans and robots working together as partners is also suggested in work by Dias et al. [2008] and Tang and Parker [2006]. This is also the next important direction of the "container" project of our work: the PEIS-Ecology project [Saffiotti et al., 2008].

Appendix A
A Domain

This appendix details the domain used in the latest experiments. The domain defines the action operators, functionality operators and method schemas for experiments 3 to 8.

A.1 Action Operators

```
(ptl-action
 :name    (goto ?r ?x)
 :precond ( ((?r) (robot ?r))
            ((?y) (room ?y) (in ?r = ?y))
            ((?x) (room ?x) (and (conn ?x ?y) (not (in ?r = ?x))
                  (not (cl-user::exists (?p) (docked ?r ?p))))) )
 :results (and (in ?r = ?x) (at ?r = ?x))
 :execute ((move-to-room ?r ?x)) )

(ptl-action
 :name    (goto-position ?r ?x)
 :precond ( ((?r) (robot ?r))
            ((?y) (room ?y) (in ?r = ?y))
            ((?x) (place ?x ?y) (not (at ?r = ?x))) )
 :results (at ?r = ?x)
 :execute ((move-to-place ?r ?x)) )

(ptl-action
 :name    (dockto ?r ?x)
 :precond ( ((?r) (robot ?r))
            ((?y) (room ?y) (in ?r = ?y))
            ((?x) (place ?x ?y) (and (not (at ?r = ?x))))
            ((?p) (peis ?p) (and (at ?p = ?x) (open ?p))) )
 :results (and (at ?r = ?x) (docked ?r ?x = t))
 :execute ((dock-to-place ?r ?x)) )
```

```
(ptl-action
 :name    (undock ?r ?x ?y)
 :precond ( ((?r) (robot ?r))
            ((?y) (room ?y) (in ?r = ?y))
            ((?x) (place ?x ?y) (and (at ?r = ?x)) (docked ?r ?x)) )
 :results (and (at ?r = ?y) (docked ?r ?x = f))
 :execute ((undock-place ?r ?x)) )

(ptl-action
 :name    (turn-on-light ?p ?r)
 :precond ( ((?p) (peis ?p) (cl-user::exists (?f) (funct ?p lamp ?f)))
            ((?r) (room ?r) (and (in ?p = ?r) (light ?r = f))) )
 :results (light ?r = t)
 :execute ((turn-on-light ?p ?r)) )

(ptl-action
 :name    (turn-off-light ?p ?r)
 :precond ( ((?p) (peis ?p) (cl-user::exists (?f) (funct ?p lamp ?f)))
            ((?r) (room ?r) (and (in ?p = ?r) (light ?r = t))) )
 :results (light ?r = f)
 :execute ((turn-off-light ?p ?r)) )

(ptl-action
 :name    (open-fridge ?p)
 :precond ( ((?p) (peis ?p) (and (open ?p = f)
                  (cl-user::forall (?q) (peis ?q)
                  (cl-user::implies (not (= ?p ?q))
                                    (not (at ?q = ?p)))))) )
 :results (open ?p = t)
 :execute ((open ?p)) )

(ptl-action
 :name    (close-fridge ?p)
 :precond ( ((?p) (peis ?p) (and (open ?p)
                  (cl-user::forall (?q) (peis ?q)
                  (cl-user::implies (not (= ?p ?q))
                                    (not (at ?q = ?p)))))) )
 :results (open ?p = f)
 :execute ((close ?p)) )

(ptl-action
 :name    (grasp-fridge ?r ?b)
 :precond ( ((?r) (peis ?r)
                  (cl-user::exists (?f) (funct ?r manipulator ?f)))
            ((?b) (object ?b) (and (= (at ?r) (at ?b))
                                   (= (in ?r) (in ?b))
                                   (have-object ?r ?b = f))) )
```

```
 :results (have-object ?r ?b = t)
 :execute ((get-object-fridge ?r ?b)) )


(ptl-action
 :name    (deliver-fridge ?r ?b ?p)
 :precond ( ((?r) (peis ?r)
                  (and (cl-user::exists (?f) (funct ?r manipulator ?f))
                       (open ?r)))
            ((?b) (object ?b) (have-object ?r ?b))
            ((?p) (robot ?p) (docked ?p ?r)) )
 :results (and (have-object ?p ?b = t) (have-object ?r ?b = f))
 :execute ((give-object-fridge ?r ?b)) )

(ptl-action
 :name    (grasp ?r ?b)
 :precond ( ((?r) (robot ?r) (cl-user::exists (?f)
                             (funct ?r gripper ?f)))
            ((?b) (object ?b) (and (object-anchored ?b)
                  (= (at ?r) (at ?b)) (= (in ?r) (in ?b))
                  (have-object ?r ?b = f) (not (at ?r = fridge)))) )
 :results (have-object ?r ?b = t)
 :execute ((get-object ?r ?b)) )

(ptl-action
 :name    (deliver ?r ?b ?p)
 :precond ( ((?r) (robot ?r))
            ((?b) (object ?b))
            ((?p) (person ?p) (and (= (in ?r) (in ?p))
                  (= (at ?r) (at ?p)) (have-object ?r ?b))) )
 :results (and (have-object ?p ?b = t) (have-object ?r ?b = f))
 :execute ((give-object ?r ?b)) )

(ptl-action
 :name    (anchor ?r ?b)
 :precond ( ((?r) (robot ?r))
            ((?b) (object ?b) (= (in ?r) (in ?b)))
            ((?rm) (room ?rm) (and (in ?r = ?rm) (light ?rm))) )
 :results (object-anchored ?b = t)
 :execute ((anchor ?r ?b)) )

(ptl-action
 :name    (smell ?r ?o)
 :precond ( ((?r) (robot ?r) (cl-user::exists (?f)
                             (funct ?r noseserver ?f)))
            ((?o) (object ?o) (and (docked ?r ?o)
                  (object-smelled ?o = f) (open ?o = t))) )
```

```
 :results (object-smelled ?o = t)
 :execute ((smell ?r ?o)) )
```

A.2 Functionality Operators

```
(cfg-functionality
 :name     (laser ?p ?f)
 :precond  ( ((?p ?f) (funct ?p laser ?f)) )
 :resource ( (re laser ?p ?f) )
 :out      ( (|distance| |laser-ranges| ?p) )
 :cost     100)

(cfg-functionality
 :name     (odometer ?p ?f)
 :precond  ( ((?p ?f) (funct ?p odometer ?f)) )
 :resource ( (re odometer ?p ?f) )
 :out      ( (|planar-pose-a| |odometry-pose| ?p)
             (|planar-vel| |odometry-vel| ?p)
             (|string| |robot-name| ?p) )
 :cost     5)

(cfg-functionality
 :name     (sonars ?p ?f)
 :precond  ( ((?p ?f) (funct ?p sonars ?f)) )
 :resource ( (re sonars ?p ?f) )
 :out      ( (|distance| |sonar-ranges| ?p)
             (|planar-pose-a| |sonar-poses| ?p) )
 :cost     5)

(cfg-functionality
 :name     (bumpers ?p ?f)
 :precond  ( ((?p ?f) (funct ?p bumpers ?f)) )
 :resource ( (re bumpers ?p ?f) )
 :out      ( (|distance| |bumper-data| ?p) )
 :cost     5)

(cfg-functionality
 :name     (camera ?p ?f)
 :precond  ( ((?p ?f) (funct ?p camera ?f)) )
 :resource ( (re camera ?p ?f) )
 :out      ( (|image| |image| ?p) )
 :cost     5)

(cfg-functionality
 :name     (stereocamera ?p ?f)
 :precond  ( ((?p ?f) (funct ?p stereocamera ?f)) )
 :resource ( (re stereocamera ?p ?f) )
```

```
 :out      ( (|image| |disparity-image| ?p) )
 :cost     10)

(cfg-functionality
 :name     (persontracker ?p ?f ?r)
 :precond  ( ((?p ?f) (funct ?p persontracker ?f))
             ((?r) (robot ?r) (in ?r = living)) )
 :resource ( (re persontracker ?p ?f) )
 :in       ( (|image| |disparity-image| ?p) )
 :out      ( (|planar-pose-a| |global-pose| ?r) )
 :cost     8)

(cfg-functionality
 :name     (actuator ?p ?f)
 :precond  ( ((?p ?f) (funct ?p actuator ?f)) )
 :resource ( (re actuator ?p ?f) )
 :in       ( (|planar-vel| |target-vel| ?p) )
 :cost     10)

(cfg-functionality
 :name     (scanmatch-localization ?r ?f)
 :precond  ( ((?r ?f) (funct ?r scanmatch-localization ?f)) )
 :resource ( (re scanmatch-localization ?r ?f) )
 :in       ( (|distance| |laser-ranges| ?r)
             (|planar-pose-a| |odometry-pose| ?r) )
 :out      ( (|planar-pose-a| |global-pose| ?r) )
 :cost     1)

(cfg-functionality
 :name     (external-localization-robot ?r ?f ?r2)
 :precond  ( ((?r ?f) (funct ?r external-localization-robot ?f)) )
 :resource ( (re external-localization-robot ?r ?f) )
 :in       ( (|anchors| |anchors| ?r2)
             (|planar-pose-a| |global-pose| ?r2) )
 :out      ( (|planar-pose-a| |global-pose| ?r) )
 :cost     100)

(cfg-functionality
 :name     (tc ?r ?f ?action ?predicate1 ?predicate2)
 :precond  ( ((?r ?f) (funct ?r tc ?f)) )
 :postcond ( (?action ?r = ?predicate2) )
 :resource ( (re tc ?r ?f) )
 :in       ( (|distance| |sonar-ranges| ?r)
             (|planar-pose-a| |sonar-poses| ?r)
             (|planar-pose-a| |robot-pose| ?r)
             (|planar-pose-a| |odometry-pose| ?r)
             (|planar-vel| |odometry-vel| ?r)
```

```
            (|string| |robot-name| ?r)
            (|planar-pose-a| |global-pose| ?r)
            (|anchors| |anchors| ?r) )
 :out      ( (|planar-vel| |target-vel| ?r) )
 :term     T
 :cost     1)

(cfg-functionality
 :name     (anchoring ?r ?f)
 :precond  ( ((?r ?f) (funct ?r anchoring ?f)) )
 :resource ( (re anchoring ?r ?f) )
 :in       ( (|list| |vision-objects| ?r) )
 :out      ( (|anchors| |anchors| ?r) )
 :cost     1)

(cfg-functionality
 :name     (csvision ?r ?f)
 :precond  ( ((?r ?f) (funct ?r csvision ?f)) )
 :resource ( (re csvision ?r ?f) )
 :in       ( (|image| |image| ?r) )
 :out      ( (|planar-pose-b| |pos-orient| ?r) )
 :cost     1)

(cfg-functionality
 :name     (lamp ?p ?f ?predicate1 ?predicate2)
 :precond  ( ((?r ?f) (funct ?p lamp ?f)) )
 :resource ( (re lamp ?p ?f) )
 :term     T
 :cost     1)

(cfg-functionality
 :name     (fridge ?p ?f ?predicate1 ?predicate2)
 :precond  ( ((?r ?f) (funct ?p fridge ?f)) )
 :resource ( (re fridge ?p ?f) )
 :term     T
 :cost     10)

(cfg-functionality
 :name     (gripper ?p ?f)
 :precond  ( ((?r ?f) (funct ?p gripper ?f)) )
 :resource ( (re gripper ?p ?f) )
 :in       ( (|string| |gripper-cmd| ?p) )
 :cost     10)

(cfg-functionality
 :name     (noseserver ?p ?f ?predicate1 ?predicate2)
 :precond  ( ((?p ?f) (funct ?p noseserver ?f)) )
```

```
 :resource ( (re noseserver ?p ?f) )
 :out      ( (|signature| |smell| ?p) )
 :cost     4)

(cfg-functionality
 :name     (rfid ?p ?f)
 :precond  ( ((?p ?f) (funct ?p rfid ?f)) )
 :resource ( (re rfid ?p ?f) )
 :out      ( (|planar-pose-a| |robot-pose| ?p)
             (|tag-info| |tag-info| ?p) )
 :cost     4)

(cfg-functionality
 :name     (odour-classifier ?p ?f ?o ?pred1 ?pred2)
 :precond  ( ((?r ?f) (funct ?p odour-classifier ?f)) )
 :resource ( (re odour-classifier ?p ?f) )
 :in       ( (|signature| |smell| ?o)
             (|tag-info| |tag-info| ?o) )
 :out      ( (|string| |odour| ?o) )
 :term     T
 :cost     10)

(cfg-functionality
 :name     (grippercontroller ?r ?f ?s1 ?s2)
 :precond  ( ((?r ?f) (funct ?p grippercontroller ?f)) )
 :resource ( (re grippercontroller ?r ?f) )
 :in       ( (|string| |start-signal| ?r) )
 :out      ( (|string| |gripper-cmd| ?r) )
 :term     T
 :cost     10)

(cfg-functionality
 :name     (readsignal ?p ?f ?r)
 :precond  ( ((?r ?f) (funct ?p readsignal ?f)) )
 :resource ( (re readsignal ?p ?f) )
 :in       ( (|distance| |bumper-data| ?r) )
 :out      ( (|string| |start-signal| ?r) )
 :cost     10)

(cfg-functionality
 :name     (manipulator ?r ?f ?s1 ?s2)
 :precond  ( ((?r ?f) (funct ?p manipulator ?f)) )
 :resource ( (re manipulator ?p ?f) )
 :term     T
 :cost     10)
```

```
(cfg-functionality
 :name     (active-anchoring ?r ?f ?action ?predicate1)
 :precond  ( ((?r ?f) (funct ?p active-anchoring ?f)) )
 :resource ( (re active-anchoring ?p ?f) )
 :term     T
 :cost     1)
```

A.3 Method Schemas

```
(cfg-method
 :name     (turn-on-light ?p ?r)
 :precond  ( ((?p) (peis ?p))
             ((?f) (funct ?p lamp ?f))
             ((?r) (room ?r) (in ?p = ?r)) )
 :body
 ( (?f1 (lamp ?p ?f |light-value| 20)) ) )

(cfg-method
 :name     (turn-off-light ?p ?r)
 :precond  ( ((?p) (peis ?p))
             ((?f) (funct ?p lamp ?f))
             ((?r) (room ?r) (in ?p = ?r)) )
 :body
 ( (?f1 (lamp ?p ?f |light-value| 0)) ) )

(cfg-method
 :name     (open ?p)
 :precond  ( ((?p) (peis ?p))
             ((?f) (funct ?p fridge ?f)) )
 :body
 ( (?f1 (fridge ?p ?f |requested-state| |open|)) ) )

(cfg-method
 :name     (close ?p)
 :precond  ( ((?p) (peis ?p))
             ((?f) (funct ?p fridge ?f)) )
 :body
 ( (?f1 (fridge ?p ?f |requested-state| |closed|)) ) )

(cfg-method
 :name     (move-to-place ?r ?place)
 :precond  ( ((?r) (peis ?r))
             ((?ft) (funct ?r tc ?ft))
             ((?fs) (funct ?r sonars ?fs))
             ((?fo) (funct ?r odometer ?fo))
             ((?fa) (funct ?r actuator ?fa)) )
 :channels ( (?f1 ?f4 (|anchors| |anchors| ?r))
```

```
             (?f6 ?f4 (|planar-pose-a| |global-pose| ?r))
             (?f2 ?f4 (|distance| |sonar-ranges| ?r))
             (?f2 ?f4 (|planar-pose-a| |sonar-poses| ?r))
             (?f3 ?f4 (|planar-pose-a| |odometry-pose| ?r))
             (?f3 ?f4 (|planar-vel| |odometry-vel| ?r))
             (?f3 ?f4 (|string| |robot-name| ?r))
             (?f4 ?f5 (|planar-vel| |target-vel| ?r)) )
 :body
 ( (?f1 (get-object-info ?r))
   (?f6 (extra-localization ?r))
   (?f2 (sonars ?r ?fs))
   (?f3 (odometer ?r ?fo))
   (?f4 (tc ?r ?ft |docked| |me| ?place))
   (?f5 (actuator ?r ?fa)) ) )

(cfg-method
 :name     (dock-to-place ?r ?place)
 :precond  ( ((?r) (peis ?r))
             ((?ft) (funct ?r tc ?ft))
             ((?fs) (funct ?r sonars ?fs))
             ((?fo) (funct ?r odometer ?fo))
             ((?fa) (funct ?r actuator ?fa)) )
 :channels ( (?f1 ?f4 (|anchors| |anchors| ?r))
             (?f2 ?f4 (|distance| |sonar-ranges| ?r))
             (?f2 ?f4 (|planar-pose-a| |sonar-poses| ?r))
             (?f3 ?f4 (|planar-pose-a| |odometry-pose| ?r))
             (?f3 ?f4 (|planar-vel| |odometry-vel| ?r))
             (?f3 ?f4 (|string| |robot-name| ?r))
             (?f4 ?f5 (|planar-vel| |target-vel| ?r)) )
 :body
 ( (?f1 (get-object-info ?r))
   (?f2 (sonars ?r ?fs))
   (?f3 (odometer ?r ?fo))
   (?f4 (tc ?r ?ft |docked| |me| ?place))
   (?f5 (actuator ?r ?fa)) ) )

(cfg-method
 :name     (undock-place ?r ?place)
 :precond  ( ((?r) (peis ?r))
             ((?ft) (funct ?r TC ?ft))
             ((?fs) (funct ?r sonars ?fs))
             ((?fo) (funct ?r odometer ?fo))
             ((?fa) (funct ?r actuator ?fa)) )
 :channels ( (?f1 ?f3 (|distance| |sonar-ranges| ?r))
             (?f1 ?f3 (|planar-pose-a| |sonar-poses| ?r))
             (?f2 ?f3 (|planar-pose-a| |odometry-pose| ?r))
             (?f2 ?f3 (|planar-vel| |odometry-vel| ?r))
```

```
            (?f2 ?f3 (|string| |robot-name| ?r))
            (?f3 ?f4 (|planar-vel| |target-vel| ?r)) )
 :body
 ( (?f1 (sonars ?r ?fs))
   (?f2 (odometer ?r ?fo))
   (?f3 (tc ?r ?ft |undocked| |me| ?place))
   (?f4 (actuator ?r ?fa)) ) )

(cfg-method
 :name     (move-to-room ?r ?room)
 :precond  ( ((?r) (robot ?r))
             ((?ft) (funct ?r TC ?ft))
             ((?fs) (funct ?r sonars ?fs))
             ((?fo) (funct ?r odometer ?fo))
             ((?fa) (funct ?r actuator ?fa))
             ((?y) (room ?y) (in ?r = ?y))
             ((?rm) (room ?rm) (and (conn ?rm ?y)
                                    (not (in ?r = ?rm)))) )
 :channels ( (?f1 ?f4 (|planar-pose-a| |global-pose| ?r))
             (?f2 ?f4 (|distance| |sonar-ranges| ?r))
             (?f2 ?f4 (|planar-pose-a| |sonar-poses| ?r))
             (?f3 ?f4 (|planar-pose-a| |odometry-pose| ?r))
             (?f3 ?f4 (|planar-vel| |odometry-vel| ?r))
             (?f3 ?f4 (|string| |robot-name| ?r))
             (?f4 ?f5 (|planar-vel| |target-vel| ?r)) )
 :body
 ( (?f1 (extra-localization ?r))
   (?f2 (sonars ?r ?fs))
   (?f3 (odometer ?r ?fo))
   (?f4 (tc ?r ?ft |at| |me| ?room))
   (?f5 (actuator ?r ?fa)) ) )

(cfg-method
 :name     (extra-localization ?r)
 :precond  ( ((?r) (peis ?r))
             ((?p) (peis ?p) (not (= ?p ?r)))
             ((?fp) (funct ?p persontracker ?fp))
             ((?fs) (funct ?p stereocamera ?fs))
             ((?r) (robot ?r) (and (in ?r = living) (light living))) )
 :out      ( (?f2 (|planar-pose-a| |global-pose| ?r)) )
 :channels ( (?f1 ?f2 (|image| |disparity-image| ?p)) )
 :body
 ( (?f1 (stereocamera ?p ?fs))
   (?f2 (persontracker ?p ?fp ?r)) ) )
```

```
(cfg-method
 :name     (extra-localization ?r)
 :precond  ( ((?r) (peis ?r)) )
 :out      ( (?f1 (|planar-pose-a| |global-pose| ?r)) )
 :body
 ( (?f1 (laser-localization ?r)) ) )

(cfg-method
 :name     (laser-localization ?r)
 :precond  ( ((?r) (peis ?r))
             ((?fs) (funct ?r scanmatch-localization ?fs))
             ((?fl) (funct ?r laser ?fl))
             ((?fo) (funct ?r odometer ?fo)) )
 :out      ( (?f3 (|planar-pose-a| |global-pose| ?r)) )
 :channels ( (?f1 ?f3 (|distance| |laser-ranges| ?r))
             (?f2 ?f3 (|planar-pose-a| |odometry-pose| ?r)) )
 :body
 ( (?f1 (laser ?r ?fl))
   (?f2 (odometer ?r ?fo))
   (?f3 (scanmatch-localization ?r ?fs)) ) )

(cfg-method
 :name     (extra-localization ?r)
 :precond  ( ((?r) (peis ?r) (robot ?r))
           ((?r2) (peis ?r2) (and (robot ?r2) (not (= ?r ?r2))))
           ((?rm) (room ?rm) (and (in ?r = ?rm) (in ?r2 = ?rm)))
           ((?f) (funct ?r external-localization-robot ?f)) )
 :out      ( (?f3 (|planar-pose-a| |global-pose| ?r)) )
 :channels ( (?f1 ?f3 (|anchors| |anchors| ?r2))
             (?f2 ?f3 (|planar-pose-a| |global-pose| ?r2)) )
 :body
 ( (?f1 (get-object-info ?r2))
   (?f2 (laser-localization ?r2))
   (?f3 (external-localization-robot ?r ?f ?r2)) ) )

(cfg-method
 :name     (get-object-info ?r)
 :precond  ( ((?p) (peis ?p))
             ((?fp) (funct ?p csvision ?fp))
             ((?r) (peis ?r))
             ((?room) (room ?room) (and (in ?r = ?room) (light ?room)))
             ((?fr) (funct ?r cam0 ?fr))
             ((?fa) (funct ?r anchoring ?fa)) )
 :out      ( (?f3 (|anchors| |anchors| ?r)) )
 :channels ( (?f1 ?f2 (|image| |image| ?r))
             (?f2 ?f3 (|list| |vision-objects| ?r)) )
 :body
```

```
 ( (?f1 (cam0 ?r ?fr))
   (?f2 (csvision ?p ?fp))
   (?f3 (anchoring ?r ?fa)) ) )

(cfg-method
 :name     (get-object ?r ?b)
 :precond  ( ((?r) (peis ?r))
             ((?fp)(funct ?r bumpers ?fp))
             ((?fpg) (funct ?r gripper ?fpg))
             ((?fpgc) (funct ?r grippercontroller ?fpgc)) )
 :channels ( (?f1 ?f2 (|distance| |bumper-data| ?r))
             (?f2 ?f3 (|string| |gripper-cmd| ?r)) )
 :body
 ( (?f1 (bumpers ?r ?fp))
   (?f2 (grippercontroller ?r ?fpgc |action| |closed|))
   (?f3 (gripper ?r ?fpg)) ) )

(cfg-method
 :name     (get-object-fridge ?r ?b)
 :precond  ( ((?r) (peis ?r))
             ((?fmp) (funct ?r manipulator ?fmp)) )
 :body
 ( (?f1 (manipulator ?r ?fmp |requested-action| |pickup-inside|)) ) )

(cfg-method
 :name     (give-object-fridge ?r ?b)
 :precond  ( ((?r) (peis ?r))
             ((?fmp) (funct ?r manipulator ?fmp)) )
 :body
 ( (?f1 (manipulator ?r ?fmp |requested-action| |leave-outside|)) ) )

(cfg-method
 :name (give-object ?r ?b)
 :precond  ( ((?r) (peis ?r))
             ((?fp)(funct ?r bumpers ?fp))
             ((?fpg) (funct ?r gripper ?fpg))
             ((?fpgc) (funct ?r grippercontroller ?fpgc)))
 :channels ( (?f1 ?f2 (|string| |start-signal| ?r))
           (?f2 ?f3 (|string| |gripper-cmd| ?r)) )
 :body
 ( (?f1 (human-signal ?r))
   (?f2 (grippercontroller ?r ?fpgc |action| |open|))
   (?f3 (gripper ?r ?fpg)) ) )

(cfg-method
 :name     (human-signal ?r)
 :precond  ( ((?r) (peis ?r))
```

```
             ((?fp)(funct ?r bumpers ?fp))
             ((?p) (peis ?p))
             ((?frs)(funct ?p readsignal ?frs)) )
 :out      ( (?f2 (|string| |start-signal| ?r)) )
 :channels ( (?f1 ?f2 (|distance| |bumper-data| ?r)) )
 :body
 ( (?f1 (bumpers ?r ?fp))
   (?f2 (readsignal ?p ?frs ?r)) ) )

(cfg-method
 :name     (anchor ?r ?b)
 :precond  ( ((?r) (peis ?r))
             ((?f) (funct ?r active-anchoring ?f)) )
 :body
 ( (?f1 (active-anchoring ?r ?f |anchoring.request| ?b)) ) )

(cfg-method
 :name     (smell ?r ?o)
 :precond  ( ((?r) (peis ?r))
             ((?o) (object ?o))
             ((?p) (peis ?p) (= (at ?p) (at ?o)))
             ((?p2) (peis ?p2) (= (at ?p2) (at ?o)))
             ((?fns) (funct ?p noseserver ?fns))
             ((?fod) (funct ?r odour-classifier ?fod))
             ((?frr) (funct ?p2 rfid ?frr)) )
 :channels ( (?f1 ?f3 (|signature| |smell| ?o))
             (?f2 ?f3  (|tag-info| |tag-info| ?o)) )
 :body
  ( (?f1 (noseserver ?r ?fns |doSmell| |yes, please =)|))
    (?f2 (rfid ?p2 ?frr))
    (?f3 (odour-classifier ?p ?fod ?o |Classify| |start|)) ) )
```

References

T. Akimoto and N. Hagita. Introduction to a network robot system. In *Proceedings of the International Symposium on Intelligent Signal Processing and Communication Systems*, pages 91–94, Tottori, Japan, 2006.

R. Alami, F. Robert, F. Ingrand, and S. Suzuki. Multi-robot cooperation through incremental plan-merging. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 2573–2579, Nagoya, Japan, May 1995.

F. Amigoni, N. Gatti, C. Pinciroli, and M. Roveri. What planner for ambient intelligence applications? *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 35(1):7–21, January 2005.

R. C. Arkin. Motor schema based navigation for a mobile robot: An approach to programming by behavior. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 264–271, Raleigh, NC, March-April 1987.

C. Bäckström. Computational aspects of reordering plans. *Journal of Artificial Intelligence Research*, 9:99–137, 1998.

D. Baker, G. McKee, and P. Schenker. Network robotics, a framework for dynamic distributed architechtures. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1768–1773, Sendai, Japan, September-October 2004.

T. Balch. The impact of diversity on performance in multi-robot foraging. In *AGENTS '99: Proceedings of the third International Conference on Autonomous agents*, pages 92–99, Seattle, USA, 1999. ACM Press.

T. Balch and R. C. Arkin. Behavior-based formation control for multi-robot teams. *IEEE Transactions on Robotics and Automation*, 14:926–939, December 1998.

A. Barrett, D. S. Weld, O. Etzioni, S. Hanks, J. Hendler, C. Knoblock, and R. Kambhampati. Partial-order planning: evaluating possible efficiency gains. *Artificial Intelligence*, 67:71–112, 1994.

C. Becker, M. Handte, G. Schiele, and K. Rothermel. Pcom - a component system for pervasive computing. In *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom)*, page 67, Washington, DC, USA, 2004. IEEE Computer Society.

T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 2001.

G. Boella. Norms and cooperation: Two sides of social rationality. In H. Hexmoor, C. Castelfranchi, and R. Falcone, editors, *Agent Autonomy*. Kluwer, Boston/Dordrecht/London, 2003.

M. Bordignon, J. Rashid, M. Broxvall, and A. Saffiotti. Seamless integration of robots and tiny embedded devices in a peis-ecology. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, San Diego, CA, October-November 2007.

S. Botelho and R. Alami. Robots that cooperatively enhance their plans. In *Proceedings of the International Symposium on Distributed Autonomous Robotic Systems (DARS)*, pages 55–68, Knoxville, TN, October 2000.

S. Botelho and R. Alami. M+: a scheme for multi-robot cooperation through negotiated task allocation and achievement. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 1234–1239, Detroit, MI, May 1999.

A. Bouguerra and L. Karlsson. Synthesizing plans for multiple domains. In *Proceedings of the 6th Symposium on Abstraction, Reformulation, and Approximation (SARA-2005)*, pages 30–43, 2005.

M. Bowling, B. Browning, and M. Veloso. Plays as effective multiagent plans enabling opponent-adaptive play selection. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 376–383, Whistler, Canada, June 2004.

M. Broxvall, S. Coradeschi, A. Loutfi, and A. Saffiotti. An ecological approach to odour recognition in intelligent environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 2066–2071, Orlando, FL, May 2006.

W. Burgard, M. Moors, C. Stachniss, and F. Schneider. Coordinated multi-robot exploration. *IEEE Transactions on Robotics*, 21:376–386, 2005.

P. Caloud, W. Choi, J-C. Latombe, C. Le Pepe, and M. Yim. Indoor automation with many mobile robots. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 67–72, Ibaraki, Japan, July 1990.

S. Cambon, F. Gravot, and R. Alami. A robot task planner that merges symbolic and geometric reasoning. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 895–899, Valencia, Spain, August 2004.

M. Carman, L. Serafini, and P. Traverso. Web service composition as planning. In *Proceedings of the ICAPS'03 Workshop on Planning for Web Services*, Trento, Italy, June 2003.

L. Chaimowicz, A. Cowley, V. Sabella, and C. J. Taylor. ROCI: a distributed framework for multi-robot perception and control. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 266–271, Las Vegas, USA, October 2003.

L. Chaimowicz, V. Kumar, and M. Campos. A paradigm for dynamic coordination of multiple robots. *Autonomous Robots*, 17(1):7–21, July 2004.

M. Cirillo, L. Karlsson, and A. Saffiotti. A framework for human-aware robot planning. In *Proceedings of the Scandinavian Conference on Artificial Intelligence (SCAI)*, Stockholm, SE, 2008.

H. Costelha and P. U. Lima. Modelling, analysis and execution of robotic tasks using petri nets. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1449–1454, San Diego, CA, November 2007.

R. Davis and R. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20(1):63–109, 1983.

M. B. Dias and A. Stentz. A market approach to multirobot coordination. Technical Report CMU-RI-TR-01-26, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, August 2001.

M. B. Dias, R. Zlot, N. Kalra, and A. Stentz. Market-based multirobot coordination: A survey and analysis. *Proceedings of the IEEE*, 94(7):1257–1270, July 2006.

M. B. Dias, B. Kannan, B. Browning, E. Jones, B. Argall, M. F. Dias, M. B. Zinck, M. Veloso, and A. Stentz. Sliding autonomy for peer-to-peer human-robot teams. In *10th International Conference on Intelligent Autonomous Systems 2008*, Baden Baden, Germany, July 2008.

B. Donald. Information invariants in robotics. *Artificial Intelligence*, 72(1-2): 217–304, 1995.

B. Donald, L. Gariepy, and D. Rus. Distributed manipulation of multiple objects using ropes. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 450–457, San Francisco, CA, April 2000.

F. Dressler. Self-organization in autonomous sensor/actuator networks. In *Proceedings of the 19th IEEE International Conference on Architecture of Computing Systems*, Frankfurt/Main, Germany, March 2006.

G. Dudek, M. Jenkin, and E. Milios. A taxonomy of multirobot systems. In T. Balch and L. E. Parker, editors, *Robot Teams: From Diversity to Polymorphism*, pages 3–22. A K Peters, 2002.

E. H. Durfee, V. R. Lesser, and D. D. Corkill. Coherent cooperation among communicating problem solvers. In A. H. Bond and L. Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 268–284. Morgan Kaufmann, San Mateo, CA, 1988.

X. Fan and T. C. Henderson. Robotshare: a framework for robot knowledge sharing. Technical Report UUCS-07-009, School of Computing, University of Utah, US, April 2007.

E. Fehr and U. Fischbacher. The nature of human altruism. *NATURE*, 425: 785–791, October 2003.

R. E. Fikes and N. J. Nilsson. Strips: A new approach in the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971. Reprint:Luger, Computation and Intelligence, 1995.

S. Fratini, F. Pecora, and A. Cesta. Unifying Planning and Scheduling as Timelines in a Component-Based Perspective. *Archives of Control Sciences*, 18 (2):231–271, 2008.

V. Frias-Martinez, E. Sklar, and S. Parsons. Exploring auction mechanisms for role assignment in teams of autonomous robots. In *Proceedings of the RoboCup Symposium*, Lisboa, Portugal, 2004.

T. Fukuda and S. Nakagawa. Approach to the dynamically reconfigurable robotic system. *Journal of Intelligent Robotics Systems*, 1:55–72, March 1988.

D. Gale. *The Theory of Linear Economic Models*. McGraw-Hill Book Company, Inc., New York, 1960.

D. Garlan, D. Siewiorek, A. Smalagic, and P. Steenkiste. Project aura: Towards distraction-free pervasive computing. *IEEE Pervasive Computing, special issue on Integrated Pervasive Computing Environments*, 1(2):22–31, April 2002.

B. Gerkey and M. J. Matarić. Multi-robot task allocation: Analyzing the complexity and optimality of key architectures. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Taipei, Taiwan, May 2003.

B. Gerkey and M. J. Matarić. A formal analysis and taxonomy of task allocation in multi-robot systems. *International Journal of Robotics Research*, 23 (9):939–954, September 2004.

B. Gerkey and M. J. Matarić. Sold!: Auction methods for multi-robot coordination. *IEEE Transactions on Robotics and Automation*, 18(5):758–768, October 2002.

G. De Giacomo and S. Sardina. Realizing multiple autonomous agents through scheduling of shared devices. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 304–311, Sydney, Australia, September 2008.

M. Gritti. A mechanism for automatic self-configuration of software components in robot ecologies. Licentiate Thesis. University of Örebro, Sweden, October 2008.

M. Gritti and A. Saffiotti. A self-configuration mechanism for software components distributed in an ecology of robots. In *Proceedings of the 10th International Conference on Intelligent Autonomous Systems (IAS)*, Baden-Baden, Germany, July 2008.

M. Gritti, M. Broxvall, and A. Saffiotti. Reactive self-configuration of an ecology of robots. In *ICRA workshop on Network Robot Systems*, Rome, Italy, April 2007.

Y. K. Ha, J. C. Sohn, and Y. J. Cho. Automated integration of service robots into ubiquitous environments. In *Proceedings of the International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, pages 177 – 182, Seoul, KOREA, October 2006.

M. Handte, C. Becker, and K. Rothermel. Peer-based automatic configuration of pervasive applications. In *Proceedings of the International Conference on Pervasive Services (ICPS)*, pages 249–260, Santorini, Greece, July 2005.

A. T. Hayes and P. Dormiani-Tabatabaei. Self-organized flocking with agent failure: Off-line optimization and demonstration with real robots. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 3900–3905, Washington, DC, May 2002.

T. Heider and T. Kirste. Smart environments and self-organizing appliance ensembles. In *Mobile Computing and Ambient Intelligence*, volume 05181 of *Dagstuhl Seminar Proceedings*, 2005.

T. C. Henderson, L. Parker, R. Dillmann, and X. Fan. Robot semantic web. In *Robot Semantic Web Workshop, IEEE International Conference on Intelligent Robots and Systems (IROS)*, San Diego, CA, October 2007.

J. Hoffmann, P. Bertoli, and M. Pistore. Web service composition as planning, revisited: In between background theories and initial state uncertainty. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1013–1018, July 2007.

F. Ingrand, R. Chatila, R. Alami, and F. Robert. PRS: A high level supervision and control language for autonomous mobile robots. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 43–49, Minneapolis, MN, 1996.

L. Iocchi, D. Nardi, M. Piaggio, and A. Sgorbissa. Distributed coordination in heterogeneous multi-robot systems. *Autonomous Robots*, 15(2):155–168, 2003.

Information Society Technologies Advisory Group ISTAG. Orientations for wp2000 and beyond. Report, September 1999. Online at http://cordis.europa.eu/ist/istag-reports.htm.

J. Jennings and C. Kirkwood-Watts. Distributed mobile robotics by the method of dynamic teams. In *Proceedings of the International Symposium on Distributed Autonomous Robotic Systems (DARS)*, Karlsruhe, Germany, 1998.

N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

E. Jones, B. Browning, M. B. Dias, B. Argall, M. Veloso, and A. Stentz. Dynamically formed heterogeneous robot teams performing tightly-coordinated tasks. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 570 – 575, Orlando, FL, May 2006.

S. Joyeux, R. Alami, S. Lacroix, and R. Philippsen. A plan manager for multi-robot systems. *The International Journal of Robotics Research*, 28(2):220–240, 2009.

N. Kalra, D. Ferguson, and A. Stentz. Hoplites: A market-based framework for planned tight coordination in multirobot teams. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Barcelona, Spain, April 2005.

N. Kalra, D. Ferguson, and A. Stentz. A generalized framework for solving tightly-coupled multirobot planning problems. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Rome, Italy, April 2007.

G. A. Kaminka, D. V. Pynadath, and M. Tambe. Monitoring teams by overhearing: A multi-agent plan recognition approach. *Journal of Artificial Intelligence Research*, 17:83–135, 2002.

L. Karlsson. Conditional progressive planning under uncertainty. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 431–438, Seattle, USA, August 2001.

L. Karlsson and A. Bouguerra. Progressive cyclic planning with search control under uncertainty and partial observability. Technical Report 1, Studies from the school of science and technology, University of Örebro, Sweden, January 2009.

N. Khayati, W. Lejouad-Chaari, S. Moisan, and J.-P. Rigault. A mobile agent-based architecture for distributed program supervision systems. In *Proceedings of the WSEAS International Conference on Information Security, Communications and Computers*, pages 15–20, Tenerife, Spain, 2005.

D. Kim, S. Park, Y. Jin, H. Chang, Y.-S. Park, I.-Y. Ko, K. Lee, J. Lee, Y.-C. Park, and S. Lee. SHAGE: a framework for self-managed robot software. In *Proceedings of the International Workshop on Self-Adaptation and Self-Managing Systems*, Shanghai, China, May 2006.

J. H. Kim, Y. D. Kim, and K. H. Lee. The third generation of robotics: Ubiquitous robot. In *Proceedings of the 2nd International Conference on Autonomous Robots and Agents (ICARA)*, Palmerston North, New Zealand, December 2004.

F. Klassner, V. Lesser, and S. H. Nawab. The IPUS blackboard architecture as a framework for computational auditory scene analysis. *Computational Auditory Scene Analysis*, 1998.

J. Koehler and B. Srivastava. Web service composition: Current solutions and open problems. In *ICAPS 2003, Workshop on Planning for Web Services*, pages 28–35, Trento, Italy, June 2003.

C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

V. Kumar. Algorithms for constraints satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.

B. Lacerda and P. U. Lima. Linear-time temporal logic control of discrete event models of cooperative robots. *Journal of Physical Agents (JOPHA)*, 2(1): 53–61, March 2008.

J. Larsson. PEIS home simulator. Master's thesis, University of Örebro, Sweden, 2007.

H. C. Lau and L. Zhang. Task allocation via multi-agent coalition formation: Taxanomy, algorithms and complexity. In *Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'03)*, Sacramento, CA, November 2003.

E. L. Lawler and D. E. Wood. Branch-and-bound methods: a survey. *Operations Research*, 14:699–719, 1966.

J. Lee and H. Hashimoto. Intelligent space – concept and contents. *Advanced Robotics*, 16(3):265–280, 2002.

J. H. Lee, K. Morioka, N. Ando, and H. Hashimoto. Cooperation of distributed intelligent sensors in intelligent environment. *IEEE/ASME Transactions on Mechatronics*, 9(3):535–543, 2004.

V. R. Lesser. Cooperative multiagent systems: A personal view of the state of the art. *IEEE Transactions on Knowledge and Data Engineering*, 11(1), January/February 1999.

X. Li and L. E. Parker. Sensor analysis for fault detection in tightly-coupled multi-robot team tasks. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Rome, Italy, April 2007.

A. Lilienthal and T. Duckett. An absolute positioning system for 100 euros. In *Proceedings of the IEEE International Workshop on Robotic Sensing (ROSE)*, Örebro, Sweden, June 2003.

R. Lundh, L. Karlsson, and A. Saffiotti. Plan-based configuration of an ecology of robots. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 64–70, Rome, Italy, April 2007a.

R. Lundh, L. Karlsson, and A. Saffiotti. Dynamic self-configuration of an ecology of robots. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3403–3409, San Diego, CA, November 2007b.

R. Lundh, L. Karlsson, and A. Saffiotti. An algorithm for generating configurations of groups of robots. Technical report, University of Örebro, Sweden, May 2007c.

R. Lundh, L. Karlsson, and A. Saffiotti. Automatic configuration of multi-robot systems: Planning for multiple steps. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, Patras, Greece, July 2008a.

R. Lundh, L. Karlsson, and A. Saffiotti. Autonomous functional configuration of a network robot system. *Robotics and Autonomous Systems*, 56(10):819–830, October 2008b.

D. M. Lyons. A process-based approach to task plan representation. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 2142–2147, May 1990.

D. M. Lyons and M. A. Arbib. A formal model of computation for sensory-based robotics. *IEEE Journal of Robotics and Automation*, 5(3):280–293, 1989.

R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, 1992.

F. Mondada, M. Bonani, S. Magnenat, A. Guignard, and D. Floreano. Physical connections and cooperation in swarm robotics. In *Proceedings of the 8th International Conference on Intelligent Autonomous Systems (IAS8)*, pages 53–60. IOS Press, 2004.

V. Montreuil, A. Clodic, M. Ransan, and R. Alami. Planning human centered robot activities. In *Proceedings of the International Conference on Systems, Man and Cybernetics (SMC)*, pages 2618–2623, Montreal, Canada, October 2007.

B. Morisset and M. Ghallab. Learning how to combine sensory-motor functions into a robust behavior. *Artificial Intelligence*, 172(4-5):392–412, March 2008.

B. Morisset, G. Infante, M. Ghallab, and F. Ingrand. Robel: Synthesizing and controlling complex robust robot behaviors. In *Proceedings of the Fourth International Cognitive Robotics Workshop, (CogRob 2004)*, pages 18–23, Valencia, Spain, August 2004.

R. Munoz-Salinas, E. Aguirre, and M. García-Silvente. People detection and tracking using stereo vision and color. *Image Vision Computing*, 25(6):995–1007, 2007. ISSN 0262-8856.

D. Nau, Y. Cao, A. Lothem, and H. Munoz-Avila. SHOP: simple hierarchical ordered planner. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 968–973, Stockholm, Sweden, July-August 1999.

D. Nau, M. Ghallab, and P. Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004. ISBN 1558608567.

A. Newell. *Unified Theories of Cognition*. Harvard University Press, 1987.

H. Nwana and D. Ndumu. A perspective on software agents research. *The Knowledge Engineering Review*, 14(2):1–18, 1999.

P. Palamara, D. Nardi, V. Ziparo, P. U. Lima, L. locchi, and H. Costelha. A robotic soccer passing task using petri net plans. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS) (Demos)*, pages 1711–1712, Estoril, Portugal, May 2008.

L. E. Parker. Current research in multi-robot systems. *Journal of Artificial Life and Robotics*, 7, 2003a.

L. E. Parker. Distributed intelligence: Overview of the field and its application in multi-robot systems. *Journal of Physical Agents (JOPHA)*, 2(1):5–14, March 2008.

L. E. Parker. The effect of heterogeneity in teams of 100+ mobile robots. In *Multi-Robot Systems: From Swarms to Intelligent Automata: Volume II*. Kluwer, 2003b.

L. E. Parker. ALLIANCE: An architecture for fault tolerant multi-robot cooperation. *IEEE Transactions on Robotics and Automation*, 14(2), 1998.

L. E. Parker and F. Tang. Building multi-robot coalitions through automated task solution synthesis. *Proceedings of the IEEE, special issue on Multi-Robot Systems*, 94(7):1289–1305, 2006.

L. E. Parker, B. Kannan, F. Tang, and M. Bailey. Tightly-coupled navigation assistance in heterogeneous multi-robot teams. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sendai, Japan, September-October 2004.

L. E. Parker, M. Chandra, and F. Tang. Enabling autonomous sensor-sharing for tightly-coupled cooperative tasks. In *Multi-Robot Systems: From Swarms to Intelligent Automata: Volume III*. Springer, March 2005.

J. Peer. Web service composition as AI planning – a survey. Technical report, Univ. of St. Gallen, March 2005.

The PEIS Ecology Project. Official web site, 2009. www.aass.oru.se/˜peis/.

C. A. Petri. *Kommunikation mit Automaten*. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

O. Pettersson. Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems*, 53(2):73–88, 2005.

M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated composition of web services by planning at the knowledge level. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1252–1259, Edinburgh, Scotland, July-August 2005.

Player/Stage Project. playerstage.sourceforge.net/, 2007.

D. V. Pynadath and M. Tambe. Multiagent teamwork: Analysing the optimality and complexity of key theories and models. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 873–880, Bologna, Italy, July 2002.

D. V. Pynadath and M. Tambe. Automated teamwork among heterogeneous software agents and humans. *Journal of Autonomous Agents and Multi-Agent Systems*, 7:71–100, 2003.

J. Rao and X. Su. A survey of automated web service composition methods. In *Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC)*, San Diego, California, USA, July 2004.

F. Rechenmann and B. Rousseau. A development shell for knowledge-based systems in scientific computing. *Expert Systems for Scientific Computing*, pages 157–173, 1992.

M. Román, C. Hess, R. Cerqueira, R. H. Campbell, and K. Nahrstedt. Gaia: A middleware infrastructure to enable active spaces. *IEEE Pervasive Computing*, 1:74–83, 2002.

S. Rosenschein. Formal theories of knowledge in AI and robotics. Technical Report CSLI-87-84, Center for the Study of Language and Information, 1987.

D. Rus, B. Donald, and J. Jennings. Moving furniture with teams of autonomous robots. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 235–242, Pittsburgh, USA, August 1995.

S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

R. B. Rusu, B. Gerkey, and M. Beetz. Robots in the kitchen: Exploiting ubiquitous sensing and actuation. *Robotics and Autonomous Systems*, 56(10): 844–856, 2008.

A. Saffiotti and M. Broxvall. PEIS Ecologies: Ambient intelligence meets autonomous robotics. In *Proceedings of the International Conference on Smart Objects and Ambient Intelligence (sOc-EUSAI)*, pages 275–280, Grenoble, France, 2005.

A. Saffiotti, K. Konolige, and E. H. Ruspini. A multivalued-logic approach to integrating planning and control. *Artificial Intelligence*, 76(1-2):481–526, 1995.

A. Saffiotti, N. B. Zumel, and E. H. Ruspini. Multi-robot team coordination using desirabilities. In *Proceedings of the 6th International Conference on Intelligent Autonomous Systems (IAS)*, pages 107–114, Venice, Italy, July 2000.

A. Saffiotti, M. Broxvall, M. Gritti, K. LeBlanc, R. Lundh, J. Rashid, B.S. Seo, and Y.J. Cho. The PEIS-ecology project: vision and results. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2329–2335, Nice, France, September 2008.

A. Sanfeliu, N. Hagita, and A. Saffiotti, editors. *Special issue on Network Robot Systems*, volume 56(10) of *Robotics and Autonomous Systems*. Elsevier, October 2008.

O. Shehory and S. Kraus. Methods for task allocation via agent coalition formation. *Artificial Intelligence*, 101:165–200, 1998.

C. Shekhar, S. Moisan, R. Vincent, P. Burlina, and R. Chellappa. Knowledge-based control of vision systems. *Image and Vision Computing*, 17:667–683, 1998.

D. A. Shell and M. J. Mataric. On foraging strategies for large-scale multi-robot systems. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2717 – 23, Beijing, China, October 2006.

R. Simmons and D. Apfelbaum. A task description language for robot control. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Vancouver Canada, October 1998.

R. Simmons, S. Singh, D. Hershberger, J. Ramos, and T. Smith. First results in the coordination of heterogeneous robots for large-scale assembly. In *Proceedings of the International Symposium on Experimental Robotics (ISER)*, Honolulu Hawaii, December 2000.

R. Simmons, T. Smith, M. B. Dias, D. Goldberg, D. Hershberger, A. Stentz, and R. Zlot. A layered architecture for coordination of mobile robots. In *Multi-Robot Systems From Swarms to Intelligent Automata: Volume I*. Kluwer, March 2002.

E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau. Htn planning for web service composition using shop2. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4):377–396, October 2004.

A. Stentz, M. B. Dias, R. Zlot, and N. Kalra. Market-based approaches for coordination of multi-robot teams at different granularities of interaction. In *Proceedings of the ANS 10th International Conference on Robotics and Remote Systems for Hazardous Environments*, Gainesville, FL, March 2004.

P. Stone and M. Veloso. Task decomposition, dynamic role assignment, and low-bandwidth communication for real-time strategic teamwork. *Artificial Intelligence*, 110(2):241–273, 1999.

A. Stroupe, T. Huntsberger, A. Oki, and H. Agahazarian. Precision manipulation with cooperative robots. In *Multi-Robot Systems: From Swarms to Intelligent Automata: Volume III*. Springer, March 2005.

K. Sycara. Multiagent systems. *AI Magazine*, 19(2):79–92, Summer 1998.

F. Tang and L. E. Parker. Peer-to-peer human-robot teaming through reconfigurable schemas. In *AAAI Spring Symposium on "To Boldly Go Where No Human-Robot Team Has Gone Before"*, Stanford University, Stanford, CA, March 2006.

F. Tang and L. E. Parker. Coalescing multi-robot teams through ASyMTRe: A formal analysis. In *Proceedings of the IEEE International Conference on Advanced Robotics (ICAR)*, Seattle, WA, July 2005a.

F. Tang and L. E. Parker. ASyMTRe: Automated synthesis of multi-robot task solutions through software reconfiguration. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Barcelona, Spain, April 2005b.

F. Tang and L. E. Parker. A complete methodology for generating multi-robot task solutions using ASyMTRe-D and market-based task allocation. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 3351–3358, Rome, Italy, April 2007.

F. Tang and L. E. Parker. Distributed multi-robot coalitions through ASyMTRe-D. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Alberat, Canada, August 2005c.

The OWL-S Coalition. Specification of the web ontology language for services. Online at www.ai.sri.com, 2009.

M. Thonnat and S. Moisan. Knowledge-based systems for program supervision. In *Proceedings of the International Workshop on Knowledge-Based Systems for the (re)Use of Programs libraries (KBUP)*, pages 3–8, Sophia Antipolis, France, November 1995.

M. Thonnat and S. Moisan. What can program supervision do for software reuse? *IEE Proceedings - Software. Special Issue on Knowledge Modelling for Software Components Reuse*, 147(5):179–185, 2000.

M. Thonnat, V. Clement, and J. van den Elst. Supervision of perception tasks for autonomous systems: the OCAPI approach. *Journal of Information Science and Technology*, 3(2):140–163, January 1994.

I. J. Timm and P-O Woelk. Ontology-based capability management for distributed problem solving in the manufacturing domain. In *Multiagent System Technologies – Proceedings of the First German Conference, (MATES 2003)*, pages 168–179, Erfurt, Germany, September 2003. Springer Verlag.

A. E. Turgut, H. Celikkanat, F. Gokce, and E. Sahin. Self-organized flocking in mobile robot swarms. *Swarm Intelligence*, 2(2-4):97 – 120, 2008. ISSN 1935-3812.

D. Vail and M. Veloso. Multi-robot dynamic role assignment and coordination through shared potential fields. In A. Schultz, L. E. Parker, and F. Schneider, editors, *Multi-Robot Systems*. Kluwer, 2003.

R. T. Vaughan, K. Støy, G. S. Sukhatme, and M. J. Matarić. Whistling in the dark: cooperative trail following in uncertain localization space. In *AGENTS '00: Proceedings of the fourth international conference on Autonomous agents*, pages 187–194, Barcelona, Spain, June 2000. ACM Press.

L. Vig and J. A. Adams. Coalition formation: From software agents to robots. *Journal of Intelligent Robotics and Systems.*, 50(1):85–118, 2007. ISSN 0921-0296.

L. Vig and J. A. Adams. Issues on multi-robot coalition formation. In *Multi-Robot Systems: From Swarms to Intelligent Automata: Volume III*. Springer, March 2005.

L. Vig and J. A. Adams. Multi-robot coalition formation. *Robotics, IEEE Transactions on*, 22(4):637–649, August 2006.

R. Vincent, M. Thonnat, and J. C. Ossola. Program supervision for automatic galaxy classification. In *Proceedings of the International Conference on Imaging Science, Systems, and Technology (CISST'97)*, Las Vegas, NV, June 1997.

Z-D. Wang, Y. Takano, Y. Hirata, and K. Kosuge. A pushing leader based decentralized control method for cooperative object transportation. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1035 – 40, Sendai, Japan, September-October 2004.

D. S. Weld. An introduction to least commitment planning. *AI Magazine*, 15 (4):27–61, 1994.

B. Werger and M. J. Matarić. Broadcast of local eligibility for multi-target observation. In L. E. Parker, G. Bekey, and J. Barhen, editors, *Distributed Autonomous Robotic Systems*, pages 347–356. Springer Verlag, 2000.

R. Zlot and A. Stentz. Complex task allocation for multiple robots. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Barcelona, Spain, April 2005.

R. Zlot and A. Stentz. Market-based multirobot coordination for complex tasks. *International Journal of Robotics Research, Special Issue on the 4th International Conference on Field and Service Robotics*, 25(1):73–101, January 2006.

Index

Publications *in the series*
Örebro Studies in Technology

1. Bergsten, Pontus (2001) *Observers and Controllers for Tagaki-Sugeno Fuzzy Systems*. Doctoral Dissertation.
2. Iliev, Bokyo (2002) *Minimum-Time Sliding Mode Control of Robot Manipulators*. Licentiate Thesis.
3. Spännar, Jan (2002) *Grey Box Modelling for Temperature Estimation*. Licentiate Thesis.
4. Persson, Martin (2002) *A Simulation Environment for Visual Servoing*. Licentiate Thesis.
5. Boustedt, Katarina (2002) *Flip Chip for High Volume and Low Cost – Materials and Production Technology*. Licentiate Thesis.
6. Biel, Lena (2002) *Modeling of Perceptual Systems – A Sensor Fusion Model with Active Perception*. Licentiate Thesis.
7. Otterskog, Magnus (2002) *Produktionstest av mobiltelefonantenner i mod-växlande kammare*. Licentiate Thesis.
8. Tolt, Gustav (2004) *Fuzzy-Similarity-Based Low-level Image Processing*. Licentiate Thesis.
9. Loutfi, Amy (2003) *Communicating Perceptions: Grounding Symbols to Artificial Olfactory Signals*. Licentiate Thesis.
10. Iliev, Boyko (2004) *Minimum-time Sliding Mode Control of Robot Manipulators*. Doctoral Dissertation.
11. Pettersson, Ola (2004) *Model-Free Execution Monitoring in Behavior-Based Mobile Robotics*. Doctoral Dissertation.
12. Överstam, Henrik (2004) *The Interdependence of Plastic Behaviour and Final Properties of Steel Wire, Analysed by the Finite Element Method*. Doctoral Dissertation.
13. Jennergren, Lars (2004) *Flexible Assembly of Ready-to-Eat Meals*. Licentiate Thesis.
14. Li, Jun (2004) *Towards Online Learning of Reactive Behaviors in Mobile Robotics*. Licentiate Thesis.
15. Lindquist, Malin (2004) *Electronic Tongue for Water Quality Assessment*. Licentiate Thesis.
16. Wasik, Zbigniew (2005) *A Behavior-Based Control System for Mobile Manipulation*. Doctoral Dissertation.

17. Berntsson, Tomas (2005) *Replacement of Lead Baths with Environment Friendly Alternative Heat Treatment Processes in Steel Wire Production*. Licentiate Thesis.
18. Tolt, Gustav (2005) *Fuzzy Similarity-based Image Processing*. Doctoral Dissertation.
19. Munkevik, Per (2005) *Artificial sensory evaluation – Appearance-Based Analysis of Ready Meals*. Licentiate Thesis.
20. Buschka, Par (2005) *An Investigation of Hybrid Maps for Mobile Robots*. Doctoral Dissertation.
21. Loutfi, Amy (2006) *Odour Recognition using Electronic Noses in Robotic and Intelligent Systems*. Doctoral Dissartation 2006.
22. Gillström, Peter (2006) *Alternatives to Pickling; Preparation of Carbon and Low Alloyed Steel Wire Rod*. Doctoral Dissertation.
23. Li, Jun (2006) *Learning Reactive Behaviors with Constructive Neural Networks in Mobile Robotics*. Doctoral Dissertation.
24. Otterskog, Magnus (2006) *Propagation Environment Modeling Using Scattered Field Chamber*. Doctoral Dissertation.
25. Lindquist, Malin (2007) *Electronic Tongue for Water Quality Assessment*. Doctoral Dissertation.
26. Cielniak, Grzegorz (2007) *People Tracking by Mobile Robots using Thermal and Colour Vision*. Doctoral Dissertation.
27. Boustedt, Katarina (2007) *Flip Chip for High Frequency Applications – Materials Aspects*. Doctoral Dissertation.
28. Soron, Mikael (2007) *Robot System for Flexible 3D Friction Stir Welding*. Doctoral Dissertation.
29. Larsson, Sören (2007) *An industrial robot as carrier of a laser profile scanner – Motion control, data capturing and path planning*. Doctoral Dissertation.
30. Persson, Martin (2008) *Semantic Mapping Using Virtual Sensors and Fusion of Aerial Images with Sensor Data from Ground Vehicle*. Doctoral Dissertation.
31. Andreasson, Henrik (2008) *Local Visual Features base Localisation and Mapping by Mobile Robots*. Doctoral Dissertation.
32. Bouguerra, Abdelbaki (2008) *Robust Execution of Robot Task-Plans: A Knowledge-based Approach*. Doctoral Dissertation.
33. Lundh, Robert (2009) *Robots that Helps Each Other: Self-Configuration of Distributed Robot Systems*. Doctoral Dissertation.