Lehrstuhl für Bildverstehen und wissensbasierte Systeme
Institut für Informatik
Technische Universität München

Transformational Planning for Autonomous Household Robots Using Libraries of Robust and Flexible Plans

Armin Müller

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Darius Burschka

Prüfer der Dissertation:

1. Univ.-Prof. Michael Beetz, Ph.D.
2. Univ.-Prof. Dr. Joachim Hertzberg, Universität Osnabrück

Die Dissertation wurde am 29.01.2008 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 10.07.2008 angenommen.

Abstract

One of the oldest dreams of Artificial Intelligence is the realization of autonomous robots that achieve a level of problem-solving competency comparable to humans. Human problem-solving capabilities are particularly impressive in the context of everyday activities such as performing household chores: people are able to deal with ambiguous and incomplete information, they adapt their plans to different environments and specific situations achieving intuitively almost optimal behavior, they cope with interruptions and failures and manage multiple interfering jobs. The investigations presented in this work make substantial progress in the direction of building robots that show similar behavior.

This thesis addresses the problem of competently accomplishing everyday manipulation activities, such as setting the table and preparing meals, as a plan-based control problem. In plan-based control, robots do not only execute their programs but also reason about and modify them. We propose TRANER (*Tra*nsformational Plan*ner*) as a suitable planning system for the optimization of everyday manipulation activities. TRANER realizes planning through a generate-test cycle in which plan revision rules propose alternative plans and new plans are simulated in order to test and evaluate them. The unique features of TRANER are that it can realize very general and abstract plan revisions such as "stack objects before carrying them instead of handling them one by one" and that it successfully operates on plans in a way that they generate reliable, flexible, and efficient robot behavior in realistic simulations.

The key contributions of this dissertation are threefold. First, it extends the plan representation to support the specification of robust and transformable plans. Second, it proposes a library of general and flexible plans for a household robot, using the extended plan representation. Third, it establishes a powerful, yet intuitive syntax for transformation rules together with a set of general transformation rules for optimizing pick-and-place tasks in an everyday setting using the rule language.

The viability and strength of the approach is empirically demonstrated in comprehensive and extensive experiments in a simulation environment with realistically simulated action and sensing mechanisms. The experiments show that transformational planning is necessary to tailor the robot's activities and that it is capable of substantially improving the robot's performance.

Zusammenfassung

Einer der ältesten Träume der Künstlichen Intelligenz ist die Konstruktion von autonomen Robotern, deren Problemlösefähigkeit vergleichbar zu der von Menschen ist. Menschliche Problemlösefähigkeiten sind besonders beeindruckend im Zusammenhang von alltäglichen Aktivitäten wie Hausarbeit: Menschen können mit mehrdeutigen und unvollständigen Informationen umgehen, sie passen ihre Pläne verschiedenen Umgebungen und spezifischen Situationen an, sodass sie intuitiv fast optimales Verhalten zeigen. Sie kommen mit Unterbrechungen und Fehlern zurecht und bewältigen mehrere, sich gegenseitig beeinflussende Aufgaben. Die Untersuchungen, die in dieser Arbeit vorgestellt werden, stellen einen substantiellen Fortschritt in die Richtung dar Roboter mit ähnlichem Verhalten zu bauen.

Diese Arbeit beschäftigt sich mit der Frage, wie alltägliche Manipulationsaufgaben wie Tischdecken und Kochen als planbasierte Kontrollprobleme gelöst werden können. Bei der planbasierten Kontrolle führen Roboter ihre Programme nicht nur aus, sondern sie stellen auch Schlussfolgerungen darüber an und modifizieren sie. Wir schlagen TRANER (*Tra*nsformationspla*ner*) als geeignetes Planungssystem zur Optimierung von alltäglichen Manipulationsaufgaben vor. TRANER plant innerhalb eines Zyklus von abwechselndem Generieren und Testen, bei dem Planrevisionsregeln alternative Pläne erzeugen und neue Pläne zum Zwecke des Testens und Evaluierens simuliert werden. Die einzigartigen Merkmale von TRANER sind, dass es sehr allgemeine und abstrakte Planrevisionen behandeln kann wie beispielsweise „staple Objekte vor dem Tragen anstatt sie einzeln zu manipulieren“ und dass es Pläne erfolgreich so modifiziert, dass zuverlässiges, flexibles und effizientes Roboterverhalten in einer realistischen Simulation hervorgerufen wird.

Diese Dissertation beinhaltet drei Hauptbeiträge. Erstens erweitert sie die Planrepräsentation sodass sie die Spezifikation von robusten und transformierbaren Plänen unterstützt. Zweitens schlägt sie eine Bibliothek von allgemeinen und flexiblen Plänen für Haushaltsroboter vor, bei der die erweiterte Planrepräsentation zum Einsatz kommt. Drittens führt sie eine mächtige und gleichzeitig intuitive Syntax für Transformationsregeln ein, zusammen mit einer Menge von allgemeinen Transformationregeln zur Optimierung von Manipulationsaufgaben in Alltagssituationen.

Die Realisierbarkeit und Stärke unseres Ansatzes wird empirisch in aufwendigen und umfassenden Experimenten dargelegt, die in einer simulierten Umgebung mit realistisch simulierten Aktionen und Wahrnehmungsmechanismen durchgeführt wurden. Die Experimente zeigen, dass Transformationsplanen notwendig ist um Roboteraktivitäten anzupassen und dass es eine substantielle Verbesserung der Leistung des Roboters ermöglicht.

Contents

Chapter 1

Introduction

The demand for technical systems in everyday domains is huge. Who of us has not dreamed of a robot that cleans the bathroom all by itself? Devices supporting humans in their daily activities are slowly finding their way into our lives. Just think of car navigation systems or automatic word detection in mobile phones. But all of these devices provide very limited functionality and are only applicable for specialized tasks. When it comes to comprehensive systems like household robots, there is still a long way to go.

Brachman (2002) proposes cognitive systems as "systems that know what they're doing". He claims that tomorrow's systems must be able to explain what they do and why they do it, learn from their mistakes, be instructed and react intelligently to new situations. This means that such a system must be able to change its own control program and find new strategies for accomplishing its tasks.

In this work, we consider a simulated autonomous kitchen robot that can prepare meals, set the table and clean things away as a typical example of a cognitive system in human-dominated environments (cf. "Toward Flexible and Robust Robots" by Nilsson (Selman et al. 1996) for a similar challenge). This domain is much more complex than the ones addressed in today's systems. The robot must not only be able to navigate safely in a close area, but also execute sophisticated manipulation tasks like grasping objects, stirring container contents and transporting things. The objects in a kitchen are very diverse — a knife must be handled differently from a plate and it even depends on the context, if a knife is to be used for cutting or if it is to be transported.

Planning is an indispensable component of any control program working successfully in a kitchen. A planner could reason about how to get a cup out of a cupboard. If the robot has to take several objects out of the cupboard it can think of an order that simplifies the reaching tasks or it could check whether temporarily moving an obstacle out of the way would help. It could reason about whether it could leave a cupboard door open until it is back or whether it would be safer to close the door in the meantime. The robot could also think about the overall structure of activities such as setting the table. Here, the question

is whether to carry the tableware one by one, whether to stack the plates, or to use a tray. Which of the options is the best critically depends on the robot's dexterity, the geometry of the room furnishing, other properties of the environment, the availability of trays, etc.

Although general-purpose planners have received much attention over the last years, they are still not able to solve these problems (Pollack and Horty 1999). Planners primarily address the problem of generating partially ordered sets of actions that provably achieve some desired goal state. While some of the planners reason about resources and generate resource-efficient plans they do so at an abstract level considering plan actions as black boxes. In contrast, the examples above require much more detailed consideration of resources and situation-dependent resource requirements. While current planners aim at provably correct plans, the most important issue in robot control is in most cases whether one plan is more reliable than another one. Current planners make the assumption that complex activities are sufficiently specified using a set of actions that must be carried out and a set of ordering constraints that prevent negative interferences between the plan steps. For every task plans are computed from scratch. The resulting plans achieve goals under assumptions that idealize reality. As a consequence of this idealization, issues such as flexibility, reliability, successful long term activity, and learning from experience are not addressed sufficiently. In contrast, robot activity requires sophisticated coordination using control structures much more powerful than simple action chaining, for example when the robot gets an object out of the way to pick up another one the obstacle should be put back immediately after the pick up is completed and before the robot leaves its current location.

In contrast, we think that it would be too demanding to develop a robot that works as it is in every kitchen. Different kitchens require different kinds of navigation, the objects are stored in different locations and some kitchens offer possibilities that others don't (e.g. not all kitchens are equipped with a dish-washer). On the other hand, we can make some assumptions in a kitchen that don't hold in a general way. First, we can assume that the environment is non-hostile. No one willingly disturbs the robot during its activity and there is no opponent that tries to reach contrary goals like for example in a game with two players. Secondly, the activities in a kitchen can be assumed to take place over and over again. Tasks like setting the table are routine repertoire and can be executed in a similar way each time they are encountered. Because of the unusual challenges and the assumptions we can make in a kitchen or other everyday environments, our approach is to equip the household robot with plans for standard tasks that work in any kitchen. Instead of optimality, we strive for what Herbert Simon (Simon 1955; 1996) called "satisficing" behavior. When the robot is introduced into a new household it adapts its behavior to the special needs of the kitchen and its inhabitants.

In this work we propose TRANER (TRAnsformational PlanNER for Everyday Activity), a framework for plan-based control, whose strategy is to equip the robot with robust and general default plans and adapt these to special situations and environments by plan transformation.

1.1 Challenges in Developing Household Robots

Today robotic applications are moving more and more to everyday environments like private households. For example, there already are robots for vacuum-cleaning. However, such robots can only perform very limited tasks, usually restricted mostly to navigation without destroying anything. In our research we strive for developing a robot with more diverse abilities that can assist people in a variety of tasks like cooking, cleaning, setting the table, etc. In this section we show how our robot differs from existing vacuum-cleaning robots and the major problems in its development.

As we have already mentioned, a general-purpose household robot is confronted with a wide variety of tasks, which not only involve navigation, but also active manipulation of objects. For the manipulation work we use two arms with 10 DOF each. This changes the 2 DOF problem of pure navigation to a 22 DOF problem when manipulation is to be performed. The computational complexity does not only involve the motion control of the arms, it also gives rise to more sources of optimization as well as failure. For example, the robot can already lift its arm when it approaches the table in order to grip something instead of navigating first and moving the arms afterwards. This optimized behavior, however, can lead to more failures, for example the arms colliding with objects or with each other.

Successful application of industrial robots largely depends on the environment the robot is operated in. The whole surrounding of the robot is usually adapted to its needs. In a household environment, this is of course impossible. The vacuum-cleaning robot faces this problem, too. It cannot depend on a certain size of the room or an arrangement of furniture unless it has operated in this room before. This means that it must explore its environment on its own. For a general-purpose household robot this problem is even more striking. While the world of a vacuum-cleaning robot only consists of obstacles and free space, the household robot must identify objects and use the resources of its environment as best as it can.

The virtue of industrial robots is that they work more precisely and are less error-prone than humans. As we have said, this is largely achieved by assuring certain conditions in its surrounding. In real-world settings, however, we cannot expect a robot to work without flaws. Because the real world is unpredictable and highly uncertain in many ways, failures cannot be avoided completely. Figure 1.1 shows a selection of possible unwanted events in the robot's activities: objects slipping from the robot's gripper, not knowing the positions of objects because they are occluded by other objects, or placing objects at the wrong position. While some of these failures could be avoided by more accurate state estimation or robot control, there will always be cases where failures occur. Therefore it is vital to equip a household robot with the means to detect failures and correct them.

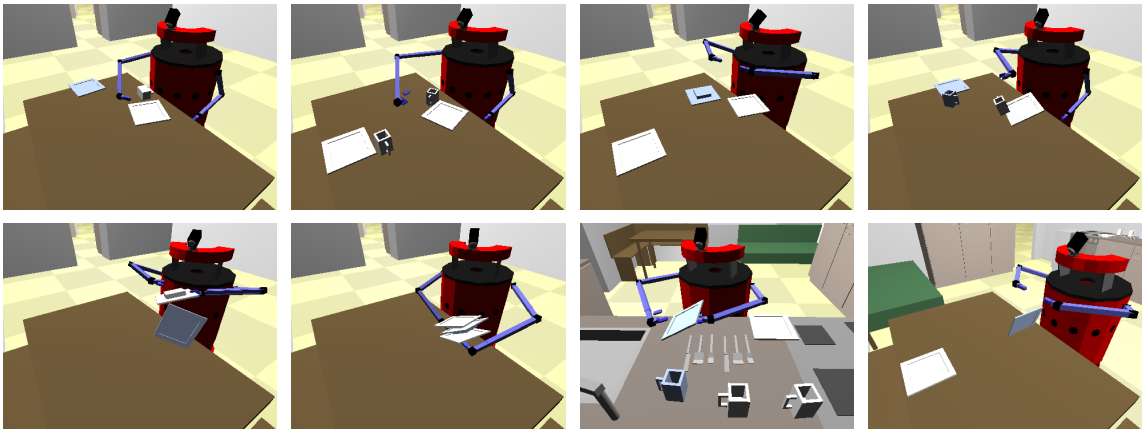
The actions performed by industrial robots or a vacuum-cleaning robot are very limited. The vacuum-cleaning robot has practically only one task: navigate. Industrial robots

often perform amazing manipulation tasks, but these are preprogrammed sequences of motor control signals. In contrast, our household robot has to perform very diverse tasks such as cooking or setting the table. To perform these high-level activities, it must be able to identify sub-activities like moving and handling objects, stirring container contents, opening cupboards and drawers, etc. These more basic actions, which can be used for several high-level activities, must work robustly, also in the light of interruption. Robustness doesn't mean that they must work flawlessly. As we said, we cannot rule out failures completely. Robustness rather means that the actions shouldn't rely on any preconditions to hold in the world and that they detect failures and react appropriately.

We have said much about the problems concerned with controlling the robot. Another aspect of real-world activity is how to describe and identify objects. In industrial settings all objects have well-defined identifiers and can be recognized with complete certainty. In contrast, the vacuum-cleaning robot cannot identify the objects in the room. It only knows that there is a certain area that is occupied by an obstacle. This description fully suffices for not destroying things while cleaning. But in a kitchen environment the matter is more complicated. We could use unique identifiers like in the industrial setting, but this would make the program very inflexible. For example, the robot needs a bowl to mix some ingredients. If the objects are identified with unique numbers, the robot relies entirely on finding this one bowl. A better approach is to describe objects symbolically, for example "a container which is large enough for the ingredients at hand". In this case, the robot can use any other bowl or find another container like a cup if no bowl is available.

Related to the description of objects is the ability to communicate with humans about the course of action taken or ask for advice if necessary. On the one hand, this involves describing objects in a way that humans understand it. It doesn't help complaining that

Figure 1.1 Different failures observed during experiments: dropping objects, not being able to grip objects, and putting objects at the wrong position.



"object 123 wasn't found". But a human can help the robot in finding "an empty cup". Similarly, the robot must have knowledge about the state of its own program. This includes its goals and the actions it has taken to achieve this goals, as well as the degree to which the goal has been achieved and the robot's belief if it is still achievable.

In sum, the demands on a general-purpose household robot differ a lot from those of an industrial robot or a special-purpose robot like a vacuum-cleaning robot. On the one hand, control is harder because of additional degrees of freedom, uncertain and unknown environments, failures, and complex actions. On the state estimation side, rich knowledge representations and object descriptions are necessary to achieve robust behavior. Finally, intelligent robots must be able to communicate about their tasks with humans.

1.2 General Approach and Research Questions

Considering the problems just described, we are convinced that to implement a useful general-purpose household robot, it should be designed a priori to work in any kitchen under all possible circumstances. This means, we don't make any assumptions about the kitchen size or specialized equipment. However, such a robot cannot perform efficiently in all environments, because some kitchens offer the possibility of optimization of the robot's plans, while others don't. For example, the cleaning of dishes should be performed differently in a kitchen equipped with a dishwasher than in a kitchen without one. So our approach aims at a way between complete generality and the restrictiveness of industrial robots. Our robot is to be able to work in any kitchen, but adapts more and more to the specific circumstances of its home kitchen.

Figure 1.2 shows the fundamental idea of our approach. We equip the robot with a set of plans, organized in a plan library. Additionally, we provide a set of plan transformation rules. By applying such a rule to a plan from the plan library, new plans are generated. If the generated plans (or some of them) work better, they are included in the plan library for later use. By doing so, the robot develops from a "general-kitchen" worker to one specialized for a specific kitchen.

The use of a plan library greatly reduces the complexity for activities when contrasted to plan generation. The activities in a household are usually very similarly each time they are executed. Therefore, it is a good assumption to use the plan that worked well last time. On the other hand, the plan transformations add the flexibility necessary in real-world environments, where the environment can change and unexpected events may happen. While the original plans in the library needn't be optimal for a specific environment, they must be robust, so that they produce the desired result under all circumstances, albeit not efficiently. By transforming the plans, the robustness is preserved, but the efficiency gets better, because features of the kitchen at hand can be taken for granted.

Of course, the implementation of the individual plans also contributes largely to meet-

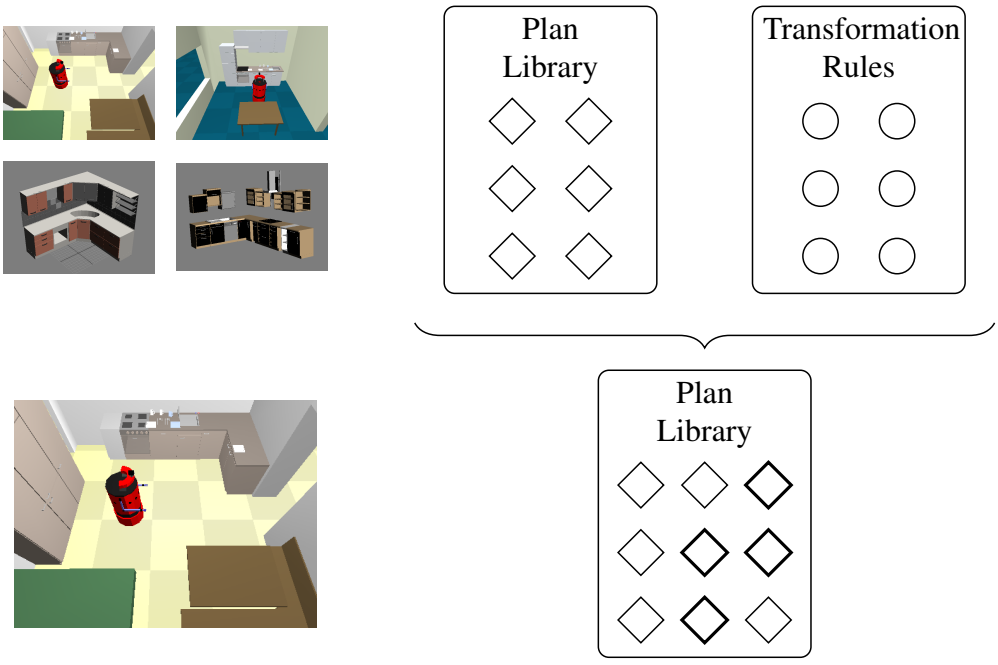
ing the demands from the last section. In all, our approach considers three major interrelated topics:

- the individual plans,
- the plan library and the interaction of plans, and
- plan transformations.

1.2.1 Plans

Plans for intelligent household robots cannot just be sequences of plan steps. As we said, the world is much too complex to rely on flawless execution of these plan steps. Consequently, the plans must constantly check for failures in the execution and try to recover from them. This mechanism must not only work on individual plans, but also involves the interaction of several plans. When a failure is detected in one plan, for example that a certain object cannot be found, this doesn't mean that the plan recognizing this failure is the best candidate for repairing it. The object finding plan cannot do much more than search again. But a higher-level plan, the one that has chosen this object to be looked for, can reconsider its steps and select another object as substitute.

Figure 1.2 General approach of adapting robot plans to a specific environment using plan transformations.



Another important aspect of coding plans is to include semantic information about the purpose of plan steps. Such information is contained in special language constructs that tell for instance that while performing the following steps the robot should stay at a certain location. This approach has several advantages over coding the same with a command to navigate to the location and then execute the following plan steps. First, it makes the plan more robust, because the construct doesn't only move the robot to the desired location once, but also takes care that the robot stays there while performing its subsequent actions. Second, this is a step towards the ability to explain the robot's course of action. Third, it makes plans transformable, an operation needed to adapt the program to the environment it is used in.

A third aspect of plan coding involves the representation of external objects, and also the robot's internal parameters. Sophisticated knowledge representation is necessary to describe objects symbolically, so that the robot can reason about optimally exploiting resources and flexibly replacing one object by another if necessary. This representation also makes communication with humans easier. Similarly, the robot can represent internal variables and parameters to be determined in the same symbolical way. This means that it has notions such as "free space on the table, which is easily reachable from the front side". The advantages of this representation over a fully specified coordinate are the same as for object representations: finding an optimal solution depending on the situation, resolving failures, and enabling communication with humans.

1.2.2 Plan Library

After considering individual plans, we must think about how they are organized in the plan library and how the plans interact when they are executed. Both the sequential and hierarchical interaction of plans have challenging aspects.

A typical sequence of plan steps could be "get the cup out of the cupboard, bring the cup to the table". This plan doesn't say anything about when to open and close the cupboard door. The opening of the cupboard is a precondition for gripping the cup and it can be performed directly before. But when should the robot close the cupboard? The plan doesn't imply any direct necessity of closing the cupboard. But if the robot doesn't close the cupboard, this might make subsequent plans much less efficient or even impossible. When the door is open the robot doesn't have as much room for navigation as it would have otherwise, so some parts of the kitchen might not be accessible to it. Besides, when the cupboard door is left open for long times, the contents collect dust and when the plates and cups are needed, they must be cleaned before they can be used. The simplest way to avoid these long-term risks is to close the cupboard immediately every time the robot has taken something out. This, however can lead to very undesired courses of action where the robot opens the cupboard, gets out a cup, closes the cupboard, opens the cupboard again, gets out the plates, closes the cupboard, etc. Finding the perfect time for performing such

clean-up actions like closing cupboards can hardly be done analytically. The best method is to transform the plan and try empirically which solution is the best.

The second interaction problem of plans lies in their hierarchical composition. We have said that we want to represent internal parameters in a symbolical form. But sooner or later, this symbolic representation must be converted to something more tangible like coordinates. At some point in the hierarchy of plans this decision has to be made. One simple heuristic would be to postpone the decisions as long as possible and let the lowest-level plan choose the parameters. However, this decision is based on very limited knowledge. The lowest-level plan could be a navigation plan. When confronted with the mission to "navigate to a position near the front side of the table", the intention of this assignment is not clear any more. A higher-level plan would know if the purpose is to put something on the table or to clean the table, which might lead to other instantiations of the abstract description. On the other hand, if the parameters are instantiated too high up in the plan hierarchy, the advantages of symbolic descriptions disappear.

1.2.3 Plan Transformations

The plans in the plan library form the basis for our general idea of Figure 1.2, the plan transformation process. Out of robust, general plans the robot is to develop environment specific, optimized plans. This includes adaptations to special opportunities or threats of a specific environment as well as integration of auxiliary goals in the hierarchical interaction of plans such as closing cupboard doors or cleaning dishes.

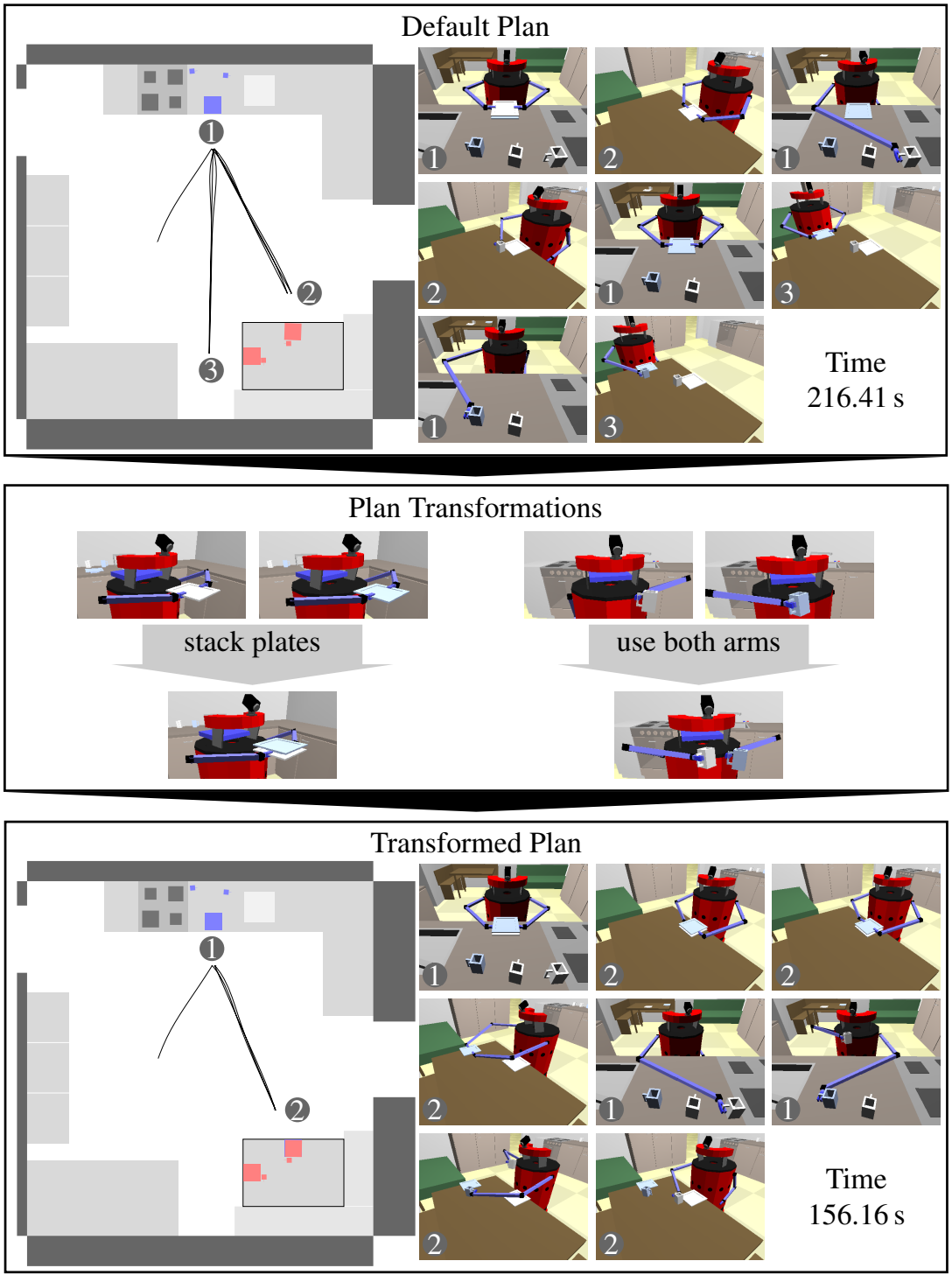
But the transformation process is more than just applying a transformation rule to a plan. It is necessary to know if a transformation is needed at all and if the resulting plan performs better than the old one. This means that each plan has to be assessed through simulated execution and evaluation. Figure 1.3 shows the transformation of one plan in the context of testing and evaluating the plans. The top third of the picture shows a summary of the performance of the original plan. This consists of the navigation path used in setting the table and a set of events during the plan execution. One can see that the robot navigates the same paths several times, an indication that the plan has potential to be improved.

The enhancement of the plan is depicted in the center part of Figure 1.3. It is the straightforward application of transformation rules. In this example, two rules are used — one that changes the plan in a way that the robot stacks plates instead of carrying them one by one, and a transformation rule making the robot use its resources better, in this case by carrying two cups at a time.

The next step is to find out if the new plan is more efficient than the old one. To this end, the plan is executed and monitored in simulation[1]. The result is shown in the bottom

[1] As our robot is only implemented in simulation, the simulation and real environments are identical in our case. In general, a simulation is needed for safely and efficiently testing plans, even though a physical robot is used. For more information see Section 6.1.3

Figure 1.3 Plan transformation including monitoring and evaluation of plans.



part of Figure 1.3. With the transformed plan, the robot doesn't cover the same route over and over again. We can also measure that the time needed to perform the complete table setting task is less than before. This means that the new plan should replace the old one, at least for the specific environmental situation. It is possible that this new plan can be enhanced further by applying the same procedure of transforming it, executing and monitoring it in simulation and evaluating it.

One challenge lies in the definition of general plan transformations, so that they are general enough to work for a wide variety of plans, plans which are unknown at the time the rules are designed. If the rules were very specific, so that they only worked on a limited set of plans, we could implement the transformed plans just as well instead of transforming them. The whole idea of transformation lives from the transformation rules being general, giving the robot the ability to find plans which are strongly adapted to the environment it is operating in. Transformations also allow the robot to adapt its plans when the environment changes or when for example one of the robot's arms breaks.

This observation doesn't only make the design of transformation rules an important issue, but also their representation. It must be possible to specify the necessary characteristics of the input plan to a rule, describe different transformation operations and make use of syntactical as well as semantical information about the plan. This representation language must be very general and flexible while still being understandable and usable by a programmer. After all, the sense of transforming plans is to have less work than designing specialized plans for each environment.

1.3 Technical Challenges

The technical challenges were mainly related to plan construction and transformation, but also include the implementation of a suitable simulation environment.

The first point was how to make plans robust and still easy to transform. To achieve this, the detection and the repair of failures must be separated. Furthermore failure recovery should take place on more than one level of abstraction to ensure overall robust behavior. For example, both the gripping and the put down plans monitor if the object is still in the robot's hand. But a higher-level plan, for example one for carrying an object from one place to another, should watch for the loss of the object, too, because between gripping and putting down the robot has to navigate without losing anything. The failure recovery on the higher level alone might be suboptimal, because often the sub-plans have more information about their current execution status and can handle the failure more effectively.

Then, we had to solve the issue of how to code transformation rules declaratively, how to find a language in which to describe transformations. It should be declarative to be readable and easy to understand as well as general so that it is applicable to a wide

variety of problems. Transformations can be very different. Some might only interchange execution steps, others regroup the usage of resources, still others replace plan steps by others. As the plans are always situated in a higher execution context, this has to be drawn into consideration and be transformed accordingly. Besides being general, the transformation rules must produce plans that are robust and can be transformed in their turn.

Finally, we had to develop a realistic simulation for a kitchen environment. Plan simulations are usually on a conceptional scale, where plan steps are represented as abstract actions such as "go to the table". What we wanted was a physical simulation of the robot and the environment in which we encounter the same problems as in reality, but are able to make simplifications at will. For example we can model the state estimation step as being left out, so that the robot can work with ground truth data, or it can be distorted with different levels of noise. Besides, we have the possibility to furnish the kitchen with equipment that would be very expensive in the real world, e.g. automatically opening doors or an automatic water tap. We made use of the Gazebo simulation environment that integrates the ODE library for physical simulation. But the possibilities were still restricted. There were no kitchen or objects used in a kitchen simulated yet. Our robot arm, too, had to be developed from scratch. Another problem was to model liquids, which are not supported by Gazebo at all. With the sophisticated simulation we have developed we can now simulate the robot's activities from the top-level plan to low-level activities. Because of the common Player interface we will be able to use most parts of our plans on a real robot that is presently under development.

1.4 Contributions

The contributions of this work comprise all the steps of building a running plan transformation system. In particular, they consist of

- ❏ concepts for plan design,
- ❏ implementing a plan library and organizing the plans therein,
- ❏ designing plan transformation rules, and
- ❏ implementing and evaluating a running system.

The first contribution is the introduction of representational constructs to be used for designing robust, general, and transformable plans. This includes special programming constructs that annotate the plan semantically. Besides, we introduce the concept of designators — symbolical descriptions of objects and parameters — to make the plans robust and independent of special environmental conditions. Our plans follow a common structure, which helps to identify their semantics when they are to be transformed.

As a second contribution, this work examines how plans should be organized in a plan library and we offer a fully implemented library for a kitchen robot. Our plan library is roughly categorized in different hierarchical levels that interact to show the required high-level behavior. We point out challenges of plans interacting sequentially or hierarchically and indicate how these can be met with plan transformations.

Developing a general structure of transformations and implementing a set of transformation rules is the third contribution of this work. We introduce a syntactical framework that allows a declarative specification of arbitrary transformation rules, which is intuitive in its usability, but still allows to define complex transformations. We demonstrate the framework with fully functional examples.

The fourth contribution consists of the implementation of a running plan transformation system in a realistic simulation of a household robot. The high-level activities of setting the table and boiling pasta are implemented by a hierarchy of robust plans that are organized in a plan library. We show how the initial behavior can be improved substantially by applying plan transformations.

1.5 Reader's Guide

This work describes the plan transformation system TRANER including plan definition and the organization of plans in a library. The evaluation is performed in a realistic simulation of a household robot.

Chapter 2 motivates the household scenario as a challenging problem of robot control and gives an informal overview of the TRANER system and its components. It further defines the focus and scope of the work.

Chapter 3 addresses the design of robust, reliable and transformable plans. After defining the requirements for designing such plans, we provide guidelines and syntactical support for developing robot plans.

Chapter 4 discusses the classification and interaction of plans in a plan library. The first part of this chapter defines categories of different plan levels. The second part examines challenges in the interaction of different plans in the library.

Chapter 5 deals with plan transformations. It explains the syntax of our transformation rules and presents a complete library of rules together with an illustrative example demonstrating the evolution of plans by applying transformation rules.

Chapter 6 covers the topic of plan execution and evaluation, which is an important component of the TRANER system. It also provides a detailed description of the simulated kitchen and robot.

Chapter 7 evaluates the approach and proves the necessity of plan transformations. It shows that robot plans can be improved significantly by plan transformations and that the quality of a plan depends on the circumstances it is executed in.

Chapter 8 concludes this work by summarizing the main aspects of TRANER and providing a prospect on future enhancements.

Some aspects of former versions of this work have been published. Aspects of plan representation can be found in (Müller and Beetz 2006; Müller, Kirsch, and Beetz 2004; Beetz, Kirsch, and Müller 2004). Work on the plan library is described in (Müller and Beetz 2007) and plan transformations are the topic of (Müller, Kirsch, and Beetz 2007). An overview of the kitchen scenario is given in (Beetz et al. 2007).

Chapter 2

Transformational Robot Planning

This chapter provides an informal overview of the kinds of problems we want to solve and how we intend to solve them. We first introduce the robot we are working with and the kinds of tasks we want it to perform. Then we point out the challenges and characteristics of everyday environments with particular reference to the household scenario. Section 2.3 is a summary of the transformation planning approach we implemented. The most important aspects for this work are pointed out in Section 2.4.

2.1 The Robot and its Household Environment

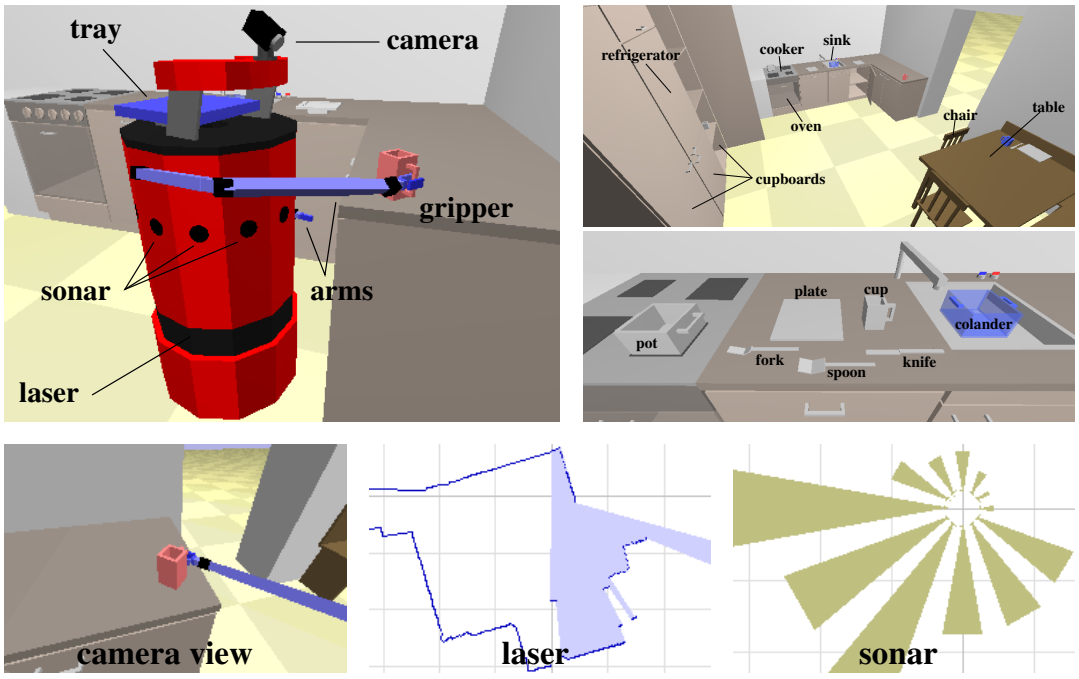
In our work we optimize the behavior of an autonomous kitchen robot in a simulated world. The decision to use a simulator was made, because a kitchen is a complex environment where the robot needs sophisticated actuators, especially arms and grippers. Such equipment, together with the kitchen itself, is very expensive and hard to maintain. The simulation is much cheaper, better available and more flexible concerning different robot hardware and different environments.

We state that the performance of plan transformations relies heavily on the specific environment. This is hardly to be tested with only one kitchen. The simulation gives us the possibility to have different testing environments at hand. Besides, we can adapt the features of the environment according to our research focus. For example, for cooking or setting the table our robot needs to open and close cupboard doors. However, for opening and closing doors in a kitchen, sophisticated motor control and plans are necessary, which we are not particularly interested in. Therefore we added automatic doors to the kitchen, which can be remote-controlled by the robot. This is an assumption that could very well be built into a real kitchen, but the simulation could be implemented with considerably less effort. In our simulation we only make simplifications which would also be possible in the real world, for instance the robot cannot teleport objects.

On the other hand, the danger of a simulation is that interesting aspects of the environment are abstracted away from and the results gotten in simulation aren't applicable to realistic settings. To avoid this danger, we chose the Gazebo simulator, which includes the physical simulation engine ODE. All objects in the kitchen and the robot are composed of solid entities, whose interaction is simulated very realistically by ODE. The interface between our robot program and the simulator is the same as between the program and a real robot. This is possible with Player (Gerkey, Vaughan, and Howard 2003), which provides a device-layer with a network interface to the hardware (or simulated hardware) underneath. This makes it possible to use the same control program in simulation and on a real robot, which we are currently building.

As we aren't concerned with state estimation, we assume that the robot's position (x/y-coordinates and orientation) are given as percepts (which is quite realistic in a known environment and with a laser-equipped robot) and the position of all objects in the robot's field of view can be determined accurately. The simulation is very realistic with respect to non-determinism in the robot's actions. Because there are several processes involved (Gazebo, Player, the robot program) the execution of a robot control program in a given situation in the simulator never causes exactly the same result. This is due to the delays in normal process and network communication and brings the simulation very close to

Figure 2.1 The robot and its kitchen environment. The bottom three images show the camera view and the laser and sonar readings.



reality.

Our robot and one kitchen it works in are depicted in Figure 2.1. The robot is equipped with sonar and laser sensors and with a camera. As an additional feature a tray can be attached to the robot, which it can use to carry objects. For its activity the robot can move in the kitchen, move its arm and grippers and move the camera.

The kitchen is a copy of a real kitchen containing furniture such as cupboards, a table, and a sink. The available objects include cutlery (forks, spoons, and knives), dinnerware (plates and cups), and objects necessary for cooking such as pots, colanders, and wooden spoons. Beside the solid objects, the simulation includes water, which can emerge from the tap and disappear into the sink. Pasta to be cooked is represented as a solid object (even in its finished state).

From a user's point of view our robot masters two high-level activities: setting the table and preparing pasta. Both plans raise interesting research issues. While the table setting is mostly concerned with carrying object around in an efficient manner (e.g. by taking several objects at a time) the pasta boiling plan is a good example of how to deal with time and use idle times efficiently (e.g. by setting the table while the pasta is boiling). The two activities rely on a set of sub-goals, for example for stacking things, grasping objects, filling containers with water, stirring the content of a container, etc (see Chapter 4).

2.2 Aspects of Household Activity

In this section we scrutinize typical activities encountered in household chores. The observations we make are specific to household activity, but many of them apply to other everyday environments such as offices or public utilities as well. Although we don't intend to mimic human behavior, we describe the special demands of household activities from the sight of a human performer, because this shows that the requirements are intrinsic in the environment, not in the robot.

2.2.1 Satisficeability

Many AI techniques for searching, planning or learning strive for optimality. Humans, however, never work optimally. Everything we do is in some way suboptimal. Not every movement we make is necessary when we eat, for instance. Just think of the consequences! If we just used less energy in the eating process by only performing the necessary movements, we could spare some of the food that we consume which again would save energy and time!

But humans don't waste time to find optimal solutions, though they are very skillful in finding a solution that works and finding it quickly. This kind of solution that is good enough, but not perfect is called "satisficing", a term coined by Herbert Simon (Simon

1955; 1996). The time to contemplate alternatives would be disproportional to the gain we could achieve by it.

On the contrary, we don't even think about how to do most of our daily activities, because we have done them often before. Maybe the activity could be enhanced in the situation at hand, but it is much faster to take the course of action that we know and have tried before. On the other hand, people can easily detect situations, in which their normal plan does not work. For example, someone has managed to cook a certain dish on Monday. Some weeks later, he does the same thing, but on a Wednesday. Because of his experience, he doesn't even consider to change the way of cooking the dish, only because it's a different day of the week. However, when the person has to prepare the dish for more people than the first time, he realizes immediately that the original course of action must be revised in a way that the resulting food suffices for all the people attending the meal.

Besides not wasting time to optimize plans into the last detail, in the real world there is no such thing as a provably optimal plan. The simple reason is that no one can ever have enough knowledge to decide on the best course of action. In the cooking example, the best choice of plan would be the one that has worked previously as long as there is no knowledge about more people coming than the first time. But if unannounced guests turn up, this plan is not optimal at all. Of course, the cook could have planned for more people in the first place, but then there might be food left, if no guests attend the meal. In any case, there is no perfect, provably optimal plan, because the situation can change at any time.

So instead of provably optimal courses of action, humans seek satisficing solutions. If a course of action works out without evident flaws including inefficiency and failures, it is approved as an acceptable plan that should be repeated in similar situations. Although the environment is very demanding, it has one big advantage: it is benevolent. This means that small errors usually don't have catastrophic effects. If we prepare a dish for the first time and prepare too much of it, the result is still acceptable. Of course, we would recognize that there is potential for improvement. But in most cases, we can try new things or modify known plans without fearing calamities. This is very different in domains like road traffic, where ill-conceived experiments can have disastrous consequences.

Another fact about everyday environments that humans take as given is that failures or errors cannot be avoided completely. It happens that we drop things and break them, that we forget to buy the ingredients we need for cooking, that appliances are out of order, etc. Instead of trying to avoid every contingency, humans can handle such situations when they come up. This makes the standard plans much more efficient and only leads to small losses of efficiency in the rare situations when unexpected events adulterate the original plan.

Another advantage of satisficeability instead of provable optimality is that planning problems stay feasible. A lot of efficiency can be gained by executing tasks concurrently.

While we are cooking we use the time to set the table and clean some of the things away that aren't needed for cooking any more. If we tried to formulate this triple problem as a classical planning problem we would fail pathetically. But humans find an approximation to the optimal solution, because they know how long activities take and if they can be interrupted. With this knowledge, free time in one activity such as cooking can be used for short, interruptable activities like bringing the plates to the table.

Overall, household environments are the wrong place for provably correct plans. The dexterity of humans shows that satisficing plans are the better choice.

2.2.2 Auxiliary Goals

Another characteristic of human activity is the execution of activities that are not crucial for the person's momentary goals. For example, when we take something out of the cupboard, we usually close the door after taking the object out. Why? The goal of the activity is certainly to have the object, so that closing the door can be seen as a waste of time in this context. Of course, we have good reasons to close the door, namely to avoid accidents, to keep the objects inside from getting dusty, etc. But these reasons have nothing to do with the original goal to get hold of a certain object.

Obviously there is more to human activity than the primary activity goals. We seem to have a set of auxiliary goals that should always be fulfilled unless there is a good reason to disregard them, like when we know that we need other objects from the same cupboard, we would leave the door open. Hammond, Converse, and Grass (1995) refers to this kind of activity as "stabilization" of the environment. This aspect of everyday activity isn't represented in current robotic systems at all and it would be very hard to model them in the classical planning scheme.

Here again the concept of satisficeability helps to describe auxiliary goals as part of efficient behavior. Even though the primary plan can be achieved faster without taking care of auxiliary goals, empirically acquired human activity includes underlying behavior to establish a certain state.

Other examples for background activity are wiping up spilt liquids, cleaning away the dishes after a meal, clearing out the dish-washer or arranging ingredients and tools before they are needed. All of these actions don't fulfill a necessary goal, but make subsequent actions easier and more efficient.

2.2.3 Adaptation to Environments

For most activities connected to housework, people have routine plans that they can use over and over again in similar situations. Interestingly, we know when the precast plan isn't appropriate and should be adapted or replaced by a new course of action. An instance of this behavior is the adaptation of actions to new or changed environments.

When you have worked in a job for a longer time, you know all your assignments and handle them skillfully. One day you get a new boss. In principle the tasks don't change much, but at first you might be unsure what specific demands the boss has and you aren't accustomed to her way of handling things. After a short time you feel comfortable again, because you have adapted the way you work to the altered conditions.

This observation is another indication that provably correct plans don't exist in real-world settings. Otherwise, our familiar plans would work identically in all situations. A better approach than planning an abstract course of action from scratch is the adaptation of known plans. In the example you already know how to do your work, so there is no need to elaborate a completely new conduct. All you have to do is to adapt your knowledge and expertise to the demands of the new boss.

In the household domain, we are also well adapted to the specific conditions of our household. When we move house or help other people with their housework, we can do all the things that we are able to do in our own home, but we need more time, because we don't know where things are stored and how time and resources can be saved in the specific environment. When you know that one of your kitchen drawers jams slightly, you avoid frequent opening and closing of that drawer. Instead, you would try to take out all the things you might need in the near future at once and close it only after you have all the things you need. Someone without your knowledge would open and close the drawer more often. After some time, the other person will have adapted to the broken drawer, too. Or she has considered your advice, thereby saving the time to find out for herself.

In sum, the dexterity of people in household activities can be largely attributed to the fact that the activity plans are not optimal, but satisficing in many situations. People can adapt their plans to specific situations and environments. Besides, not only steps that lead to the present goals are executed, but also activities that ease the execution of subsequent, currently unknown goals.

2.3 Approach

For developing successful household robots, we should learn from the way humans handle their daily activities. This doesn't mean that we intend to imitate human behavior, but that we can use the observations from the last section to build robots that are able to perform well in everyday environments. Therefore, we first summarize the main observations of human behavior that should be carried over to robots. We then introduce the TRANER system, which adapts robot plans to specific situations and environments. After that, we show how TRANER contributes to the demands for household robots derived from human behavior.

2.3.1 Demands on Household Robots

As we have seen, the most important concept for humans to work in everyday environments is advisability, which has been developed and is constantly improved by experience. The basic idea of our approach is to obtain similar behavior by transforming existing robot plans instead of generating them from scratch. Let us regard an example of how this concept works for robots.

Consider a robot that is to prepare a meal with a main dish and a dessert. Let's assume that some ingredients for both dishes are in the larder. The robot should realize that the same subtask — go to the larder to fetch an ingredient — occurs in both plans, the one for preparing the main dish and the one for making the dessert. In this case it should combine the two steps into one and only go to the larder once. But possibly the robot cannot carry all the things at once. Then it should transform the plan again in a way that it uses a container for transporting the ingredients. So the resulting plan would be to go to the larder once and get all ingredients at once, possibly by using a container.

But wait, is this really the best plan? Maybe there is no container at hand and the provisioning of the container needs more time than just going to the larder twice. Or the kitchen might be very small and the ingredients for the dessert are in the robot's way while preparing the main dish. So the best plan can actually not be determined without knowledge and experience about the environment. Depending on the robot's dexterity, the size of the kitchen, the way to the larder, the places where possible containers are kept, etc. different plans should be favored.

This robot already has a plan in the outset and it doesn't try to prove the correctness of its plan or analyze its performance a priori. Instead it uses it and observes what happens. In this context it is important to remember that household environments are benevolent. Wrong decisions don't have catastrophic effects. Besides, transformed plans can be tested in simulation before using them in the real world.

By transforming and evaluating plans consecutively the robot adapts its plans more and more to its familiar environment. This doesn't mean that it cannot work in any other environment. We assume that the robot is equipped with robust default plans that work in any environment, for example in all kinds of kitchens. By transforming the default plans, they are adapted to the specific environment the robot works in most of the time, but it is still able to use the original default plans for other kitchens. This is similar to the way humans adapt to their well-known environments.

In the example, only the robot's primary goals were mentioned. But there are other, secondary goals to consider for tasks such as "use a container". The usage includes choosing and fetching a container and putting it away after it has fulfilled its purpose. When the container is only used for one carrying task, the execution of the auxiliary goals should obviously take place just before and after the main plan is executed. But what is the robot to do when it needs to carry a lot of objects and possibly needs the container for future

activities? It might be more appropriate to leave it in a place where it can be retrieved easily, but this depends on the available space and the time until it is used again. We see, for auxiliary goals we have similar problems as for primary goals. Because of that, we use plan transformations also for treating this kind of stabilization activity.

In the case of humans, we have argued that people accept failures to happen. It is of no use to forestall all kinds of possible failures and some are beyond our sphere of influence anyway. We can hardly expect a robot to perform better than humans with respect to failures. Therefore, the plans must be prepared to recognize failures and react to them appropriately. This means for the default plans to include elaborate failure detection and recovery mechanisms, which have to be preserved when plans are transformed.

To summarize, a household robot should show the same characteristics in its execution as humans do: strive for satisficeability, be tolerant towards failures, execute auxiliary goals appropriately, and adapt to the specific environment. In this work we will show how these demands can be achieved by thorough plan design and plan transformations.

2.3.2 The TRANER System

In this section we describe our system TRANER (TRAnsformational PlanNER for Everyday Activity) in more detail. The strategy is to equip the robot with robust and general default plans and a set of general transformation rules. Using the transformation rules the robot can compute more efficient plans by adapting its default plans to specific situations and environments, and to its own dexterity.

As shown in Figure 2.2 TRANER operates in two modes. In the online mode TRANER retrieves plans for the given tasks and executes them. It also measures the performance of the plans by monitoring them with a given cost function that might for example include the time needed to complete tasks and the execution failures that occurred. Based on the measured performance TRANER decides whether or not it should try to improve the respective plans. In idle times, possibly at night, TRANER generates alternative candidate plans and evaluates them in simulation. If the candidate plans achieve better performance than the respective plan in the library, the library plan is either replaced by the best new one or the library is extended with a new plan, depending on the situation of the environment for which the plan has been adapted. In the rest of the section the components of TRANER are explained along the illustration of Figure 2.2.

Plan Library. All plans are stored in a plan library. It contains default plans (surrounded by thin lines) that can be expected to work in any kitchen as well as plans that have already been enhanced by transformations and are adapted to the specific situations and needs of the environment and the robot working in it (depicted with thicker lines).

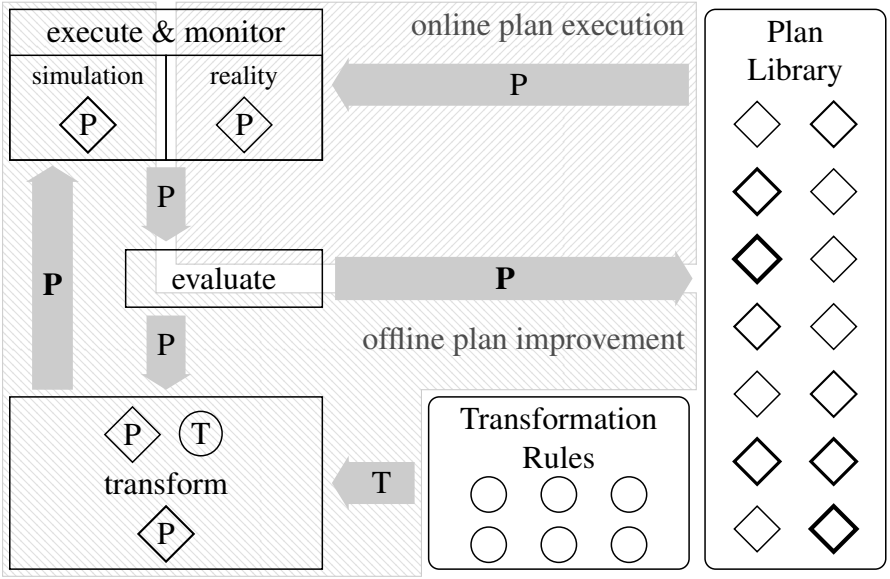
The default plans must be robust and general without making assumptions about the

specific situation they are executed in. For example, a plan telling the robot to get the ingredients from the larder one by one is probably suboptimal for most environments and situations, but it is quite certain to work.

Plan Execution. When the robot tries to reach a goal — be it a user command or some subgoal required by another plan — it selects a plan from the plan library. All the plans in the library are expected to work, so that in a known situation, the chosen plan produces the desired result with high probability. Plans are annotated with situation descriptions they are adapted to. Default plans work in every situation. If possible, an already adapted plan is used, otherwise the appropriate default plan is chosen.

During execution of the plan the robot monitors itself and records an execution trace containing symbolic and quantitative information. For example the execution trace for setting the table as shown in the left part of Figure 2.3 includes the time taken to achieve the plan, the fraction of time used for navigation, turning and arm movements, how often the navigation and turning goal was called, how often an arm was used, the sum of the covered distance of the robot and the arms, the chosen paths for navigation, the time steps when objects were picked up and put down, the arm(s) used for manipulating objects,

Figure 2.2 Overview of TRANER. Libraries (Plan Library, Transformation Rules) are illustrated as boxes with rounded corners. Plans are depicted as diamond-shaped objects, transformation rules as circles. Plans surrounded by thicker lines have been transformed more often than those with thin lines. The boxes mark operations (execution, evaluation and transformation) on data structures like plans, transformation rules or execution traces.

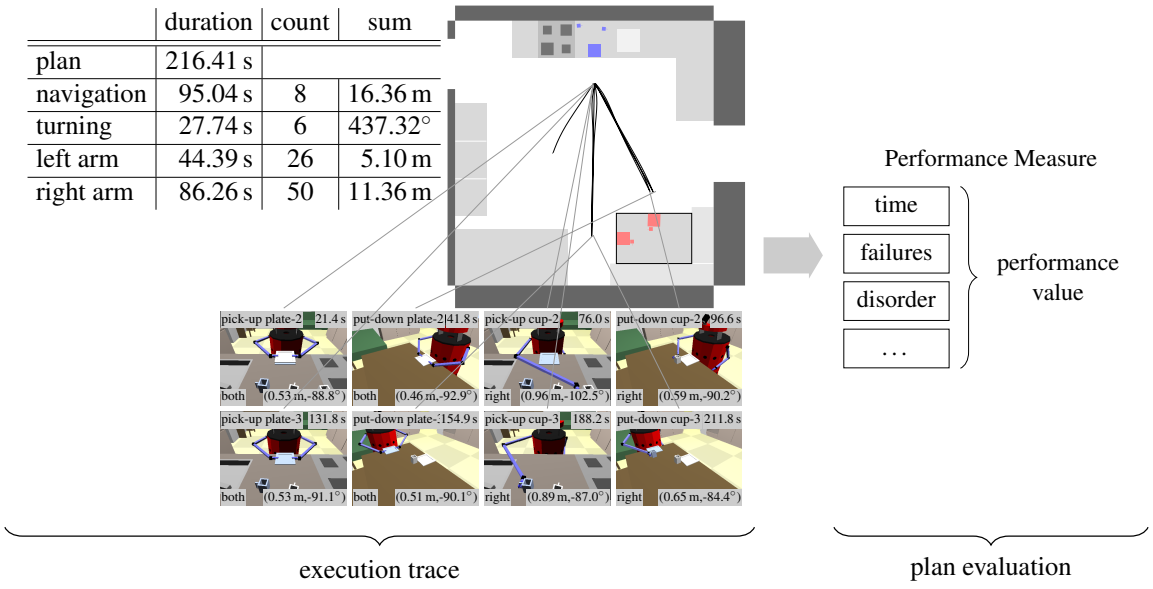


and the relative position of the robot to the manipulated object. Other information that is recorded, but not shown in the figure includes if the plan achieved the goal successfully, how many failures occurred and could be recovered during execution, or how much disorder was induced in the environment.

Plan Evaluation. After the execution of the plan has ended (either successfully or with a failure) the execution trace is evaluated. The evaluation operation computes a performance value of the plan using the execution trace (right part of Figure 2.3). If the performance is unsatisfactory according to some other performance value computed with the same performance measure or if no comparable performance value is available, the plan should be tried to be optimized by transformation. Other performance values can have been obtained from previous executions and transformation cycles of the plan or from observing humans performing the same task[1].

Plan Transformation and Rules. For making plans better, a set of transformation rules is given. A transformation rule accepts a plan with a certain structure and produces several new plans according to the specified rule. For improving plans, a possible transformation

Figure 2.3 The left part shows data of the execution trace monitored during plan execution and on the right how it is evaluated by computing a possible performance value.



[1]The aspect of observing humans for getting performance measures is subject to ongoing research and hasn't been used in this work.

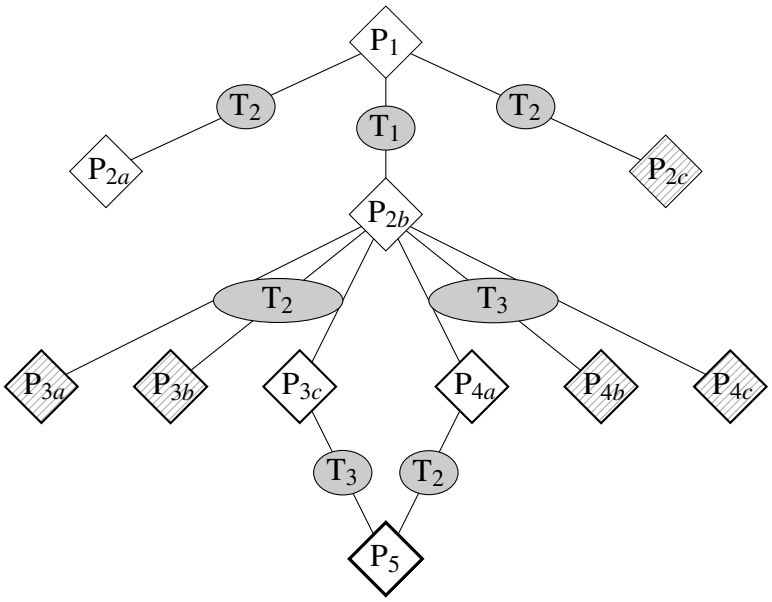
rule is chosen and applied. All the resulting plans are then executed in simulation, monitoring the same quality criteria as in the real plan execution before. Although we expect our environment not to be overly perilous, it is advisable to test transformed plans in simulation, also with regard to time efficiency. It may happen that a new plan is not even syntactically correct, the transformer doesn't check this. But such a plan will not be executed successfully, and will consequently be discarded. All the resulting plans are tested against the original plan and against each other. The best plan is then stored in the plan library.

TRANER performs an exhaustive search in order to find better plans. Plan improvement as just described doesn't necessarily have to end after one transformation step, instead it is an iterative process. If the new plan is better than the old one, but still doesn't fulfill the demands of the evaluation function, it is stored and transformed further. Besides, if the original plan fits the structural conditions of several transformation rules, all possible rules are applied and the resulting plans are evaluated against each other. This procedure corresponds to an unguided search in the space of plans, where plan transformations describe the possible state transitions (see Figure 2.4). To make this search more efficient, especially when many transformation rules are available, the transformation rules provide an applicability condition, which uses the benchmark data observed during plan execution to determine if the transformation rule might be a good choice. For example, if the robot dropped objects when it had to transport several things, a rule that adds the use of a container would be more useful to make the plan stable than one that changes the order in which the objects are handled.

Operation Modes. We should mention explicitly the two working modes of this scheme using online or offline transformations. In the procedure just described, the original plan from the plan library is used without any conditions and only afterwards the performance is evaluated. The transformations needn't take place immediately, but can be scheduled for a later time when the robot has more time to spare. This is the offline operating mode we are mostly concerned with in this work.

Another method, similar to that of *Hacker* (Sussman 1973; 1977) is testing the plan in simulation before it is used in the real world and transforming it at once. Only when it fulfills the performance criteria, it is employed to fulfill the task. In real-world settings with complex plans and several transformation rules matching a plan, this operating mode can seriously influence the robot's overall performance. In the world we operate in, it is not vital to execute plans optimally, but to start executing them at once. The acceptance of a system thinking half an hour about its best course of action cannot be expected to be very high, even if the resulting plan takes some minutes less than the one without transformations.

Figure 2.4 Search Tree for improving the default plan for setting the table. The default plan places a plate and a cup on a table one by one, repeating these steps for each person attending the meal. Each transformed plan is executed and monitored in simulation.



Transformation Rules:
T_1: reorder plan steps
T_2: stack objects
T_3: carry objects using both arms

Failed Plans:
P_{2c}: stack plates on cups
P_{3a}: stack plates, stack cups
P_{3b}: stack cups
P_{4b}: carry plates in parallel
P_{4c}: carry plates and cups in parallel

Successful Plans:
P_1: set the table (default plan)
P_{2a}: stack cups on plates
P_{2b}: all plates, then all cups
P_{3c}: stack plates
P_{4b}: carry cups in parallel
P_5: stack plates and carry cups in parallel

Best plan after evaluation: P_5

2.3.3 Benefits of the Approach

This approach mirrors the idea of satisficeability. Instead of trying to build a plan from scratch and try to prove its correctness and optimality without any reference to the environment, we let the robot find out what works by trial and error. This is similar to what people do. The assumption that there exist default plans from the outset is no restriction to this approach, but a strength, because we can rely on different sources for these default plans. One possibility would be to use a classical planner and let it generate a plan from

scratch. Another option is generating the plan from natural-language descriptions drawn from websites like `ehow.com` or ontologies like Cyc. Another future idea would be to have the robot be instructed by humans and generate rough default plans from these directions.

Furthermore, failure handling can be integrated in a natural way. The default plans should work in a way that they succeed in most cases, but they still might fail. One approach could be to make the agent more cautious and include more failure recognition, avoidance and fixing. But this produces agents that check the fuse every time before they turn on the cooker. In contrast, plan transformations produce situation and environment specific plans, so that only the necessary failure treatment is included without sacrificing stability.

Our approach also allows to integrate auxiliary goals into the robot's daily activities. Without representing them explicitly, the robot will sooner or later realize that it is acting in a suboptimal way. When the high-level plans are tested, for example setting the table, the robot soon sees that when it leaves the doors of the cupboards open it comes into situations where it cannot move properly in the kitchen any more. Over time the plans will be adapted in a way that the desired behavior of keeping certain assumptions in the world intact is attained.

Finally, the underlying principle of the approach is that agents are adapted to a particular environment. This seems quite restrictive at first, one might prefer agents that can work in any environment. However, we have shown that this adaptation to a special environment is one of the strengths of human conduct. People can get along in unknown situations, too, because they have default plans for known situations and can adapt them, but they are more efficient in familiar settings. The same should apply to robots. Our default plans are designed in a way that they can be executed in any environment, but don't necessarily work efficiently. With the adaptation the robot can develop much more sophisticated skills that resemble human expertise.

2.4 Research Focus

We have sketched out how autonomous service robots should develop and enhance their plans in order to produce sophisticated environment and situation specific behavior. This approach has received very little attention so that there is still a lot of work to be done. In this section we point out the focus of this work and how it contributes to the overall goal of plan transformation in autonomous robots.

2.4.1 Plan Design

One important part of the whole approach is how to represent and develop the default plans so that they are robust and general on the one hand and ready to be transformed on

the other hand. The default plans must be robust and general without making assumptions about the specific situation they are executed in.

One aspect for making plans better understandable are designators, logical descriptions of entities in the world. Instead of addressing objects or locations by unique identifiers they are described by the demands in the situation. For example, instead of specifying "Bring me the object with ID 0815" we would demand "Bring me a book that has adequate proportions for stabilizing the jiggling table". This way of describing things is more expressive and gives the robot much more freedom in transforming plans.

Moreover, the plans must be structured in a way that potential transformations can be recognized. In order to make plans more self-explaining, we introduce macro structures like `at-location`, `with-designators` or `with-auxiliary-goals`. On the one hand, these constructs ensure that the action is performed under certain conditions, for example the `at-location` macro warrants that the robot doesn't move while it is performing the action. On the other hand, the plans become more structured and therefore easier to transform.

Also failures must be represented explicitly in the plan structure. By introducing the construct `with-failure-handling` in our framework, it is easy to see which are the functional parts, the monitoring tasks, and where the failure recovery is performed. Without this knowledge failure recovery might get lost during transformations and the plan structure would be more opaque, so that the applicability of the transformations rules would be more difficult to ascertain.

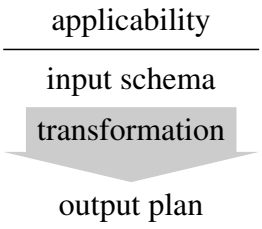
2.4.2 Plan Library

The plan library contains all kinds of plans, the whole range from high-level activities invoked by user commands to basic skills such as moving joints. We classify plans according to some structural criteria and examine the interaction of these plans.

One kind of interaction is between sub- and super-plans. Here the question arises, which plan complements specific information about the plan execution, for example in the gripping plan, which hand to use. Another interaction of plans arises in their sequential execution. Here we must decide how to distribute auxiliary goals. Both questions are hard to answer in a general way and largely depend on the specific situation.

2.4.3 Transformation Rules

Beside the representation of the plans and their organization in the plan library, we have to consider the structure of plan transformation rules. All the rules in this work are represented in the following way:



The *input schema* is a structural demand on the original plan to be transformed by this rule. For example, when the rule reorders plan steps that are executed at the same location, it is necessary that at least two `at-location` constructs be contained in the plan, otherwise the plan cannot be transformed by this rule. Besides determining if the plan is transformable, the input schema serves for structuring the input plan in a way so that the transformation can be performed, i.e. certain plan parts are bound to variables. The *transformation* part of a transformation rule describes how aspects of the input plan must be modified in order to receive the desired output. This operation is described by first-order logic rules. The *output plan* specifies how the resulting plan is composed from the transformation results.

The *applicability* is the heuristic about when the transformation rule can be applied most effectively. This hint makes use of observations made during the execution of the plan. For example, we might have a transformation rule that reorders plan steps when some of the steps are executed at the same location, but are interrupted by other plan steps in the original plan[2]. This transformation rule is a good choice when the robot has observed that several plan steps are executed at the same location. In other cases, for example when the robot realizes that it drops things frequently, this transformation rule doesn't help much, even if the input schema matches.

2.4.4 Plan Execution and Evaluation

Finally, we regard the execution and evaluation of robot plans. For testing transformed plans a realistic, accurate simulation environment is needed. Therefore, we use the Gazebo simulator, which provides a very good physics engine and realistic, non-deterministic simulation. In our current system the "real" and simulated environment are identical. This ensures that when we have a real robot also a realistic simulation is available.

When the simulated or real execution has ended (either successfully or with a failure) the result is evaluated. For this purpose, an execution trace is observed during plan execution, for example the time and resources needed or the number of occurred failures, the fraction of them that could be repaired etc. Then a cost function is applied to this data, calculating a performance value for comparing plans.

[2]This transformation might lead to conflicts with the interleaved plan steps, but the robot will find that out when testing the plan.

The topic of automating monitoring and evaluating plan execution, storing and retrieving situation specific annotated plans and the transformation process is very wide. This work concentrates on the basic procedure, more automation and more refined search strategies for transformations is subject to future work.

2.5 Summary

In this chapter we have first given an overview of our experimental setting, which consist of a simulated autonomous mobile robot equipped with two arms, an optional built-in tray, a camera, and laser sensors working in a fully equipped kitchen including furniture and cooking devices.

We have then analyzed key aspects of human everyday activity and identified three critical observations: (1) Humans do most of their activities suboptimally, but good enough to succeed. (2) Besides achieving primary goals, there is a wide variety of auxiliary goals to be met like closing cupboards or cleaning things away, which don't contribute to any primary goal directly, but make the execution of primary plans much more efficient. (3) Successful household activity demands adaptation to changing environments.

These requirements led us to our plan transformation system TRANER, which enables a robot to meet all those challenges. The basic idea is to define a library of default plans and enhance these plans by plan transformation. This iterative process leads to behavior that is adapted to a specific environment and performs reasonably well, although not provably optimally.

Finally, we have pointed out the research focus of this work, which is primarily aimed at the design of robust, reliable plans, their organization in a plan library, the design and implementation of plan transformation rules, as well as plan execution and evaluation.

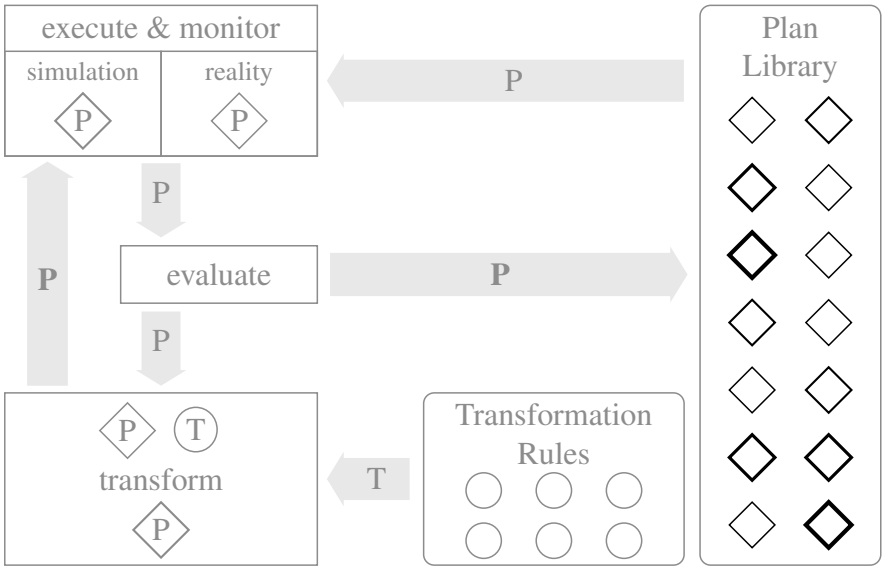
Chapter 3

Plan Design

The central concept in TRANER are plans (Figure 3.1). This is why our first focus is on how to structure and implement the plans, which are then to be transformed. The initial plans must work reliably in every conceivable situation and be tolerant with respect to failures, but still must have an explicit, declarative structure so that they can be transformed.

After indicating the important aspects for plan design in the light of transformational planning, we explain the plan structure in more detail. One important aspect of the plan representation is the description of objects in the world and how to represent free plan parameters adequately.

Figure 3.1 Part of TRANER described in this chapter.



3.1 Motivation and Design Issues

Plans for the blocks world and of many other AI planning applications often contain actions such as *pick-up(?x)* and *put-down(?y)*, where at execution time the variables are bound to *"A"* and *"B"*. *A* and *B* denote blocks in the real world. Knowing the identity of the object and the satisfaction of some high-level conditions such as the hand being empty and the block being accessible from the top suffices to carry out the action successfully.

Unfortunately, when implementing a robot performing household chores this view of actions and plans is much too simplistic. A robot acting in the real world doesn't have symbolic constants that denote objects in the real world. Rather it must parameterize its actions with object descriptions generated by the robot sensors, which are incomplete, inaccurate, ambiguous, and even faulty. Instead of relying on exact quantitative values or pure symbolic descriptions, the robot should have a qualitative idea of an object or a parameter (like a location) in higher-level plans to make plan revisions possible and quantitative values on lower levels to make an execution possible.

Also, while picking up objects in the blocks world is an atomic action, picking up objects in a kitchen requires very complex sophisticated robot behavior: some objects have to be picked up using a pinch grasp while others require a wrap grasp. Some objects can be picked up with one hand while others require two hands, and even others require tools such as spoons. Picking up objects from a cupboard might require complex reaching trajectories and even temporarily removing obstacles. Also, in order to achieve smooth and high performance behavior, activities must not be limited to be sequences of discrete action. A skilled robot will start reaching for an object on the table *before* it has arrived at the desired position. This means for a robot to show such behavior it needs a sophisticated plan representation allowing complex interactions of actions and the means to react appropriately to the situation at hand. The latter can be achieved by transforming plans without the need to develop an optimal procedure by hand for all conceivable circumstances.

Another aspect that is totally missing in the blocks world, but crucial in a kitchen, are failures. When gripping objects, they can slip from the gripper. The robot might collide with objects or not be able to find a specific tool. It is important to recognize such situations as failures and to react to them appropriately.

The bottom line is that we need a rich language that can represent plans for dealing with the intricacies of real-world environments. We can summarize these requirements as robustness, generality, transformability and the enabling of cognitive capabilities.

A robot has *cognitive capabilities* when the structure of its plans allow it not only to execute them, but also reason about and explain them. *Robustness* includes the awareness of, monitoring of, and recovery from failures. *Generality* of plans means that they work in typical situations and environments without making assumptions of the specific circumstances they are facing. Plans are *transformable* when their structure allows to manipulate them with transformation rules.

3.1.1 Enabling Cognitive Capabilities

The plans we develop are for robots working in human environments. This means that they strongly interact with humans and must behave in a comprehensible way towards people, who expect cognitive behavior. Brachman summarizes cognitive systems in the following way:

> A truly cognitive system would be able to learn from its experience — as well as by being instructed — and perform better on day two than it did on day one. It would be able to explain what it was doing and why it was doing it. [...]
>
> Brachman (2002)

One component for achieving cognitive behavior in robots is to use an appropriate representation language for plans on all levels of abstraction. A robot should, for example, be able to tell what it is doing, which goals it must achieve in order to achieve a higher-level goal and recognize that tasks have failed. This is why McDermott (1992a) considers robot plans to be any robot control program that cannot only be executed but also reasoned about and manipulated. These are the basic ideas of his plan language RPL, which we use as our plan representation language (see Section 3.2).

Beside the possibility of manipulating plans, the reasoning should be facilitated by structuring the plan along semantic aspects. For example, a plan might tell the robot to go to a certain position, grasp the wooden spoon, and stir the content of a pot with it. This plan is correct, but it doesn't say explicitly that the robot should stay where it is while stirring. When a failure occurs because the robot has moved, it has no hint what went wrong. In contrast, we could tell the robot to perform the actions of picking up the wooden spoon and stirring at a certain position. This, too, would make it navigate to the desired position first, but then it would know that staying at the position is vital for the program execution. We will introduce several such constructs that provide more semantic information about the plan.

3.1.2 Robustness

The environment our robot works in is a very close simulation of a real-world kitchen. In such a complex environment failures are unavoidable. Figure 1.1 on page 4 shows a small variety of failures that occur while the robot is handling objects. The robot sometimes cannot grip the object reliably, things slip from the grippers, or the robot places an object at a position, where another object already is (either because its state estimation is inaccurate or because it has placed one of the objects at the wrong position). These failures are sometimes caused by the lacking dexterity of the robot, sometimes by the uncertainty of the environment. In any case, it is impossible to avoid them completely.

To achieve robustness, this means first of all that the robot must have certain expectations about what should not happen during the execution of a plan and the means to check whether such an unwanted situation has occurred. Secondly, there must be mechanisms that allow to react from the failure by retrying the plan execution, fixing the problem, or failing, hoping that the next process in the hierarchy finds a solution.

For example, if the robot detects that it was unable to grip a cup, the low-level grip plan would try to grip again. If after several trials the cup still couldn't be reached, the grip plan fails and passes the failure description to the higher-level plan. This plan might try to grip with the robot's other gripper or from another location. If that still doesn't help, a higher-level plan might decide to get a different cup that can be reached more easily.

3.1.3 Generality

By generality we mean that plans work in all household environments under almost all circumstances. In particular, a plan should not assume a certain structure of the environment, it should not make assumptions about certain objects to be present in the world and it should be able to adapt parameters to environmental conditions without prior assumptions.

We only require the robot to work in *almost* all circumstances, because exceptional situations can occur that we don't want the robot to check for every time it executes the plan. One example of such a situation is when the robot needs to use the cooker, but the fuse has burned out. In this situation, our default plan will not work. Of course, the plan could contain a step to check for the fuse before using the cooker. But the cases with an intact fuse occur much more frequently than the failure case. Therefore the plan would become very inefficient without much gain in robustness. In the rare situation of such failures the plans can be adapted by plan transformations.

One aspect of achieving generality is to describe objects symbolically and bind them to existing objects during execution. Also free parameters like a good position for the robot or the trajectory of the arm for gripping an object are defined symbolically, but the instantiation happens by heuristics, learned functions or specialized modules (e.g. motion planners). In this way, the assumptions about existing objects are reduced as far as possible. Again, there are some limits to generality. In a kitchen, we must assume that there is some area where the robot can put things like a work top or a table and that there are containers and cutlery.

Unfortunately, the generality of plans is not consistent with the need for high performance plans. A plan that has to work in any environment cannot take advantage of favorable opportunities in its specific surrounding. But this is the job of the plan transformations — to adapt general plans to specific environments. The manual adaptation of a robot to each specific environment isn't necessary.

However, having general plans is a crucial precondition for successful plan transformation. The search space on possible transformed plans can get huge with only a few transformation rules and as all the plans have to be tested, the search should only consider very few well-chosen plans. Therefore, the default plan from which the transformation starts must be extremely reliable, robust and general. If these criteria are fulfilled, the transformational planner needn't care about fixing bugs in the default plan. Instead, it can use the restricted resources to optimize an already valid plan and adapt it to specific situations.

In sum, to achieve generality, plans should be described only by a set of subgoals to be fulfilled, but not how to reach them. In the same way, objects and parameters should not be described by identifiers or coordinates, but rather by symbolical descriptions that can be adapted depending on the environmental conditions.

3.1.4 Transformability

In the context of TRANER a vital characteristic of default plans is that they can easily be transformed with transformation rules. This means that the plans must have an explicit, clear structure so that they match the appropriate transformation rules. The purpose of plan steps must be identifiable to enable meaningful transformations.

The requirement of transformability is closely related to that of cognitive capabilities. For both purposes the plan structure must be explicit so that the robot can reason about and explain its plans. The main difference lies in how this information is used. For displaying cognitive behavior, the robot makes inferences about its current goals and their execution status in order to be able to explain its doings and to comprehend the overall situation. In the case of transformations, the plan must be understood mainly on a syntactic level.

3.2 Plan Representation Language

We use an extension of the Reactive Plan Language (RPL) for representing our plans. In the following we first give a brief overview of reactive planning and its development. Then we introduce RPL, focusing on the features necessary for this work. For more information on RPL see McDermott (1990; 1991; 1992b; 1993).

3.2.1 History of Reactive Planning

Research on planning in its early stages assumed that generating plans is the difficult part, whereas executing the plans is much easier (Gat 1997). But when real-world applications, particularly autonomous robots, were regarded, it turned out that plan execution is a big

problem. It was not as simple as expected, since plan steps can fail or produce unexpected outcomes and it might be necessary to react to unforeseen events during the execution.

To overcome these problems, more powerful plan execution models called "reactive planning" (Firby 1987; Agre and Chapman 1987; Payton 1986) were developed. Reactive planning systems monitor plan execution and are able to react to failures and unexpected events by changing the course of the lower-level commands.

Reactive planning was embedded on the execution layer of three-layer architectures (Gat 1997; Bonasso and Kortenkamp 1995). This execution layer was on the middle layer, making it possible to execute abstract plans generated from the planner on the top layer using basic skills from the bottom layer. The plans on the top layer were very abstract actions like *pick up object X* that didn't match any skill implemented on the bottom layer. The middle layer therefore was the broker in the form of a reactive planning (or rather reactive execution) system to enable a successful robot behavior.

Interestingly when developing these three-layer architectures the top (planning) layer was often largely neglected and for the success of autonomous robot control mainly the execution layer was responsible. This shows that the execution plays an important role when building autonomous systems. One of the best-known reactive planning systems is *RAP*s (Reactive Action Packages) developed by Firby (1989), which were originally developed to be used as execution layer in three-layer architectures.

*RAP*s are the immediate ancestor of the Reactive Planning Language (RPL) (McDermott 1993; 1992b), which originally implemented the same ideas. However, McDermott (1990) rejects the need of an additional planning layer, which abstracts away from execution details. The main idea of RPL can be summarized as follows:

> This language embodies the idea that a plan is simply a program for an agent, written in a high-level notation that is feasible to reason about as well as execute.
>
> (McDermott, Cheetham, and Pomeroy 1991)

3.2.2 Basic Concepts

RPL is a concurrent reactive control language. It provides conditionals, loops, program variables, processes, and subroutines as well as high-level constructs (interrupts, monitors) for synchronizing parallel actions. To make plans reactive and robust, it incorporates sensing and monitoring actions, and reactions triggered by observed events. It makes success and failure situations explicit and enables the specification of how to handle failures and recover from them. RPL is implemented as an extension to LISP, but is a pure procedural language.

One feature of RPL is the reaction to external events. This feature is implemented by so-called fluents, which will be introduced in Section 3.3.1. Besides, RPL offers sophis-

ticated control over the interaction of plans like sequential execution (`seq`) and several modes of parallel execution (`par`, `pursue`, `partial-order`), differing in the way failures in subplans are treated.

Moreover, RPL includes well-known programming constructs such as `when`, `if`, `cond`, `unless`, `let`, and `let*`. These have the same name and functionality as the LISP constructs, but work in the context of plans. Besides, RPL includes the LISP functionality of defining macros, which makes it possible to specify the language extensions (explicit designator definitions and semantic annotations) described in sections 3.3 and 3.4. Arbitrary LISP functions can be used inside plans, too.

The original RPL doesn't support goals as an explicit concept. We are of the opinion that an explicit representation of goals makes plans better understandable and is necessary for enabling cognitive capabilities. Therefore, goals are represented as objects and a plan is a means to achieve a given goal. So in contrast to the original RPL we don't apply plans directly, but set a goal, which the program must achieve in whichever way. This makes it possible to use different plans for achieving the same goal in different contexts.

For setting goals, we introduced the construct `achieve`, which expects a goal specification as input. When several similar goals are to be achieved, `for-all` works like a mapper, achieving the same goal multiple times with different parameters. A special kind of goal is one where the robot doesn't manipulate the environment, but tries to identify objects. The `perceive` construct achieves special perception goals in order to transform a symbolic description of an object to an object reference. Note that this can involve changing the environment (for example by opening a cupboard), but the change is not the primary goal.

3.2.3 RPL Code Tree

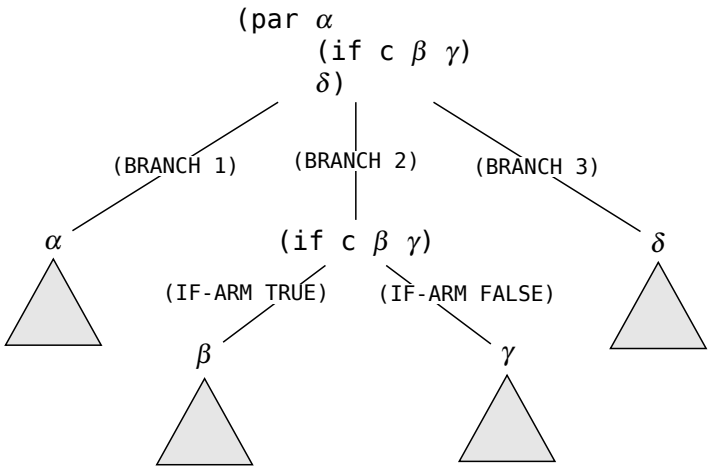
Internally, RPL represents plans in the form of code trees, an example of which is shown in Figure 3.2. This representation is not just used for interpreting plans, it is present inside the program and can be modified at run time.

This feature is essential to making plan transformations work. Because the internal plan structure is available for manipulation, the transformation can replace parts of the code tree by others or exchange branches of the tree.

Each node in the tree can be identified uniquely by a description of the path leading from the root to the node to be identified. For example, the subtree β in Figure 3.2 is identified by the path `((branch 2) (if-arm true))`. Another way of identifying parts of the tree is by giving them a name (called "tag" in RPL). This feature makes the addressing of tree nodes more comfortable.

The code tree concept is not only a prerequisite for executing plan transformations. It is also needed to specify the paths in order to identify the parts of the input plan for the transformation steps.

Figure 3.2 RPL code tree of the plan `(par` α `(if c` β γ `)` δ`)` indicating the path names of the different subtasks.



3.3 State Representation

One important detail of defining plans is how to represent the robot's belief with respect to its own situation (velocity, position, arm movement) and objects in the world. In the following we describe the concept of fluents and how they represent values that describe the state of the world. Then we introduce designators — functional descriptions of objects — to make the plans general, robust, transformable, and enable it to show cognitive capabilities.

3.3.1 Fluents and Fluent Networks

Successful interaction with the environment requires robots to asynchronously respond to events and sensor data arriving from the state estimation as well as to internal program state changes. To handle these requirements, RPL provides a special type of variable called *fluent*. The values of these variables change over time and every change can be detected by special RPL constructs in other parts of the control program.

Fluents are best understood in conjunction with the RPL statements that respond to changes of fluent values. The RPL statement `(whenever fluent body)` is an endless loop that executes `body` whenever the value of `fluent` gets the value "true". Besides `whenever`, `(wait-for fluent)` is another control abstraction that makes use of fluents. It blocks the execution of the following plan steps nonrecurringly until `fluent` becomes "true".

The information provided as fluents by the state estimation includes the robot's estimated position and velocity, the arms' positions, joint angles and velocities, moving

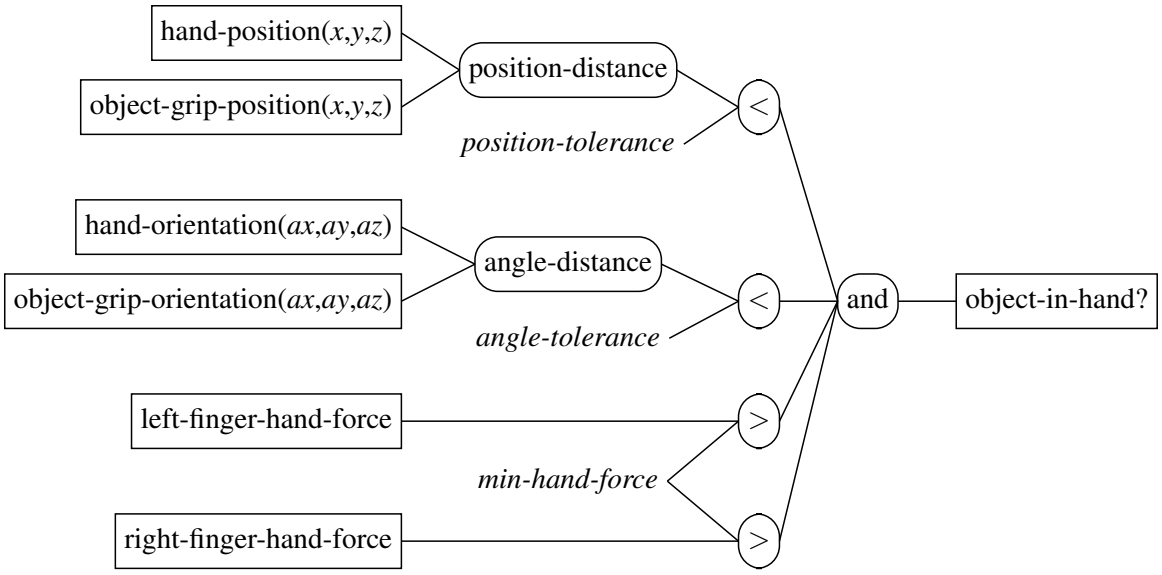
status, the positions, velocities and force values of the gripper fingers and the camera pan, tilt and zoom status.

Even more interesting than single fluents is the possibility to build networks of fluents[1]. A composite fluent is calculated from the current values of other fluents. When one of the constituent fluents changes its value, the resulting fluent is recalculated.

For example, Figure 3.3 shows a fluent network computing the output fluent `object-in-hand?`, which is true if and only if the distance between the hand pose (position and orientation) and the object's gripping pose is smaller than the allowed tolerances (`position-tolerance` and `angle-tolerance`) and the hand-force sensors of both fingers are above a certain minimum. The inputs `object-grip-position` and `object-grip-orientation` are fluent networks by themselves, calculated by the current object pose and the relative gripping position. The `object-in-hand?` fluent can be used for checking if the robot has lost an object, which it is carrying and can thus detect failures.

By using fluents, especially for monitoring the program for failures or unexpected situations, the plan becomes robust and reactive. Fluents contribute to plans on all levels of abstraction, so that even higher-level plans have knowledge about what is going on in

Figure 3.3 Fluent network computing the fluent `object-in-hand?`. The fluent network receives the hand and object's gripping poses (positions and orientations) and the hand-forces of the fingers as its input fluents. Fluents are depicted as rectangles, functions are denoted by ovals, and parameters are written in italics.



[1] Fluent networks are an extension to the original RPL.

the world. By building fluent networks, the plans can also work with abstract knowledge representation instead of pure quantities delivered by the state estimation.

3.3.2 Object Designators

In everyday situations, most goals involve the handling of objects. Usually objects are addressed by unique identifiers and typical goals are "put object 0815 onto object 0955". This is feasible in worlds with a small number of well-defined objects. However, in a household or other real-world scenarios, this kind of description is not enough to make plans robust and general and poses several problems: (1) Such environments often contain a great variety of objects that can hardly be enumerated, just think of tea bags. (2) Objects might cease to exist or come into existence during the robot's actions, for example when the robot opens an egg, the egg doesn't exist any more, but the content can be regarded as a new object. (3) The robot might not be able to relate the object identification to an object in its perceptual descriptions (Agre 1988).

For example, if object 0815 is a cup and object 0955 the kitchen table, the goal "put object 0815 onto object 0955" might fail, because the cup cannot be found or the table is littered with other objects and there is no room for the cup. Depending on the high-level goal, the robot might still be able to fulfill its task. For example, if the overall goal was to prepare tea, the robot can use a different cup and put it on the worktop instead of the table. With absolute object descriptions, this kind of failure recovery is not possible.

Designator Concept

To overcome these problems, RPL offers the concept of designators, which are developed further in TRANER. A designator is a symbolic representation of an object, possibly including the object type, structural properties (e.g. its color, is it clean, or has it a flat surface) and functional properties (e.g. can it contain liquids). The goal in the example would then be to "put a clean cup onto a flat surface", which gives the robot much more freedom for optimization and failure tolerance. Of course, on some level of execution — usually on lower level plans — the abstract descriptions must be instantiated to well-identified objects. It is possible that the cup will be associated with object 0815, but it might also be identified as some other cup. The following designator describes an object, which is a currently unused and clean plate a specific person likes to use:

```
(some entity (type plate)
             (status unused)
             (status clean)
             (preferred-by person))
```

Designator Instantiation

For using objects described by designators, the robot has a goal `perceive`, which tries to establish a connection between a designator and an object in the world using a perceptual plan. The strategy for the plan achieving the perceive goal is first to check whether the designator already has a reference to an object in the world. If this is the case, the robot has to check if this reference is still valid, i.e. if the object is still at the position where it was last perceived. Therefore, the robot moves to the supposed position and if it finds any objects there it compares their properties with the ones described in the designator.

When no reference is available or the robot decides it to be invalid, it generates a list of possible objects matching the description and then proceeds as in the case of already instantiated objects, that is, look for the object at the last known position. When no matching object can be found, the plan to achieve the perceive goal fails.

Once an object in the real world matching a designator has been found, this specific object is added as a *reference* to the symbolical designator description. From then on the coordinates and specific properties of the reference object can be used in the program.

With designators, a plan gets more general in that it doesn't rely on specific objects to be available in the world. The use of designators also enhances failure tolerance, because when one object isn't available another object can be found with similar properties. Besides, plan transformations have more flexibility in improving the plan by changing descriptions of designators. Because designators are functional descriptions of objects, they contribute to the goal of showing cognitive capabilities by providing information about the object properties and why a specific object was used in a situation.

3.4 Plan Structure

RPL already provides many aspects for having a plan representation that enables the writing of plans with the characteristics described in Section 3.1. In this section we present the language extensions we added (enlargement of the designator concept and semantic annotations of the plans) for writing robust, general and transformable plans. We then define a general framework of all the plans in the plan library.

3.4.1 Parameter Designators

In Section 3.3.2 we introduced designators as a means of specifying objects by giving their properties instead of unique identifiers. The concept of designators can be seen in a more general scope when they are also used to describe parameters needed for execution. Instead of preprogramming default values, the parameters to be set are described by functional aspects and instantiated at execution time.

Consider a plan that is to specify how the robot is to pick up a cup. Clearly, which hand to use, which kind of grip to apply, how to position itself to grasp the cup reliably, which gripper and arm trajectory to take, and where to hold the cup after picking it up cannot be determined before seeing the cup. The optimal choice also often requires to take the situational context, expectations, and foresight into account. In our kitchen scenario, the hand to be used, the pose of the hand during transportation, and the pose used to grip the cup decide on the reliability and efficiency with which the tasks of the subplans are executed.

Figure 3.4 shows three typical examples of parameter designators in our plans. The first one returns a collision free arm trajectory that is possible to follow with the given hand (a parameter designator) and that enables the robot to grip the specified entity (an object designator) by closing its fingers afterwards. The second example describes a hand that is currently unused and a certain entity can be gripped with. The last returns a location for an entity on the table where a person prefers to sit and which matches the relative position of the entity to a cover.

At the time of execution, the abstract descriptions of the designators must be converted to concrete values of the parameters, that means that they are instantiated. Our plan interpretation calls specific arbitration mechanisms that decide on the appropriate value of the parameter. The choice of these values depends on the environment situation and the robot's abilities. As it is infeasible to foresee every possible situation and optimize it, one has to find appropriate heuristics. The most efficient and flexible way is to learn the heuristics automatically using the experience the robot makes when executing the plans (Kirsch and Beetz 2007), or to use specialized modules like motion planners.

Like object designators, parameter designators make plans very flexible, robust and transformable. Plan Transformations can add or remove constraints of the designators. Moreover, they provide a declarative structure of what conditions the parameter value must fulfill. This again makes cognitive behavior possible. When a robot is to justify why it put an object at a certain position, it can describe its choice in terms of abstract concepts like "I needed a place where to put the object safely near the sink".

Figure 3.4 Typical examples of parameter designators.

```
(some trajectory (collision-free)
                 (gripable-at-end entity)
                 (possible-with hand))
(some hand (status free)
           (gripable-with entity))
(some location (on table)
               (at (preferred-seating-location person table))
               (matches (entity-cover-location entity)))
```

3.4.2 Semantic Annotation of Plans

Plan transformations work mostly on the syntactic structure of plans. But to describe sophisticated plan transformations often more information is necessary. Therefore, we extend the RPL language by introducing specific constructs using the macro mechanism offered by RPL. The new constructs annotate plans in a way that more semantic information about the doings of the plans can be transmitted to the transformational planner. The plans are also better structured and more readable, thus enabling cognitive capabilities. These constructs make some assumptions about the robot, e.g. that the robot is mobile and that it manipulates objects. These assumptions, however apply to any robot acting in scenarios like a kitchen.

In the following we present the new constructs `with-failure-handling`, `at-location`, `with-designators`, and `with-auxiliary-goals`. Listing 3.1 on page 48 shows an example using these constructs. The example will be described in detail in Section 3.5.

Behavior Monitoring and Failure Recovery

Robustness is one of the design issues we wanted for our plans. To achieve it, it is necessary to (1) monitor the robot's behavior, (2) signal failures, (3) catch failures, and (4) recover from failures.

Behavior monitoring is best accomplished by running the monitoring and the execution parts in parallel. RPL already offers a rich set of constructs for parallel execution, like `par`, `pursue`, `try-all`. With all these control structures it is possible to monitor the behavior, but they differ in how they implicitly handle failures. `par` succeeds if all actions succeed or fails if one fails. `pursue` succeeds if one action succeeds or fails if one fails. `try-all` succeeds if one action succeeds or fails if all fail.

All constructs don't check for and react depending on the type of failure. Failure handling is implicitly encoded in the control structure. This makes it difficult for the robot to act appropriately depending on the failure type and for the transformational planner to identify the purpose (behavior monitoring, failure handling) of the code pieces.

We achieve an effective integration of execution monitoring, failure generation, failure catching and failure recovery by introducing `with-failure-handling` as an additional control structure of the plan language RPL:

```
(with-failure-handling failure
  recover-vars
  (recover <recovery-code>)
  (monitor <monitoring-code>)
  (perform <body>))
```

This control structure executes `<body>` and `<monitoring-code>` in parallel. Every

time `monitoring-code` detects an undesired behavior it signals a failure. This failure or a failure thrown from a lower level plan is bound to the local variable `failure` and `<recovery-code>` is executed. `<recovery-code>` checks for the type of the failure and whether a handler for this type is specified, otherwise the failure is propagated to higher level plans. For storing information about previous recover tries, `<recovery-code>` uses the local variables defined in `recover-vars`.

The control structure `with-failure-handling` makes the purpose of code pieces meaningful for the planning mechanisms. If the code pieces produce goal achieving behavior they are located in `<body>`, if they monitor the robot's behavior and signal failures they are part of `<monitoring-code>`, and if they are to catch and recover from signaled failures they must be specified as `<recovery-code>`.

In sum, the `with-failure-handling` construct enables the development of robust plans that react appropriately to failures. At the same time, the code is annotated semantically, so that plan transformations can be performed and the robot can show cognitive behavior, for example by explaining an action as a means to recover from a previous failure.

Location of Execution

Often a robot needs to make sure that it is located at a certain position in order to fulfill a manipulation task such as gripping an object. The construct `(at-location location subplan)` ensures that the robot moves to the location before it starts the execution of the subplan. During the plan execution, it monitors constantly if the robot leaves that location. If this is the case, it takes care to move the robot back into the specified position. This construct is important for manipulating objects. It contains a specific failure monitoring and recovery, using the construct `with-failure-handling`, to make the execution robust.

Specifying Designators

An interesting characteristic of a plan — both for plan transformations as well as producing cognitive capabilities — is which object and parameter designators are used and instantiated in the plan. Using the construct `(with-designators (desigs) subplan)` the transformational planner is informed that the specified designators are used in the plan. This knowledge allows optimization of common resource usage of plans and provides information of whether plans can be executed in parallel or compete for the same resources. Knowing that a parameter designator is used in a plan allows optimization of parameter instantiation for the plan.

Formulating Auxiliary Goals

Not every action taken by an intelligent robot is an explicit step towards its goal, for example cleaning up or opening and closing cupboards. Some of these actions must be taken to achieve a goal, but it is not specified when they have to be taken, like opening a cupboard. Others aren't really necessary, but ensure the reachability of subsequent goals, such as the closing of cupboards. The construct `with-auxiliary-goals` indicates that an action should be taken, but doesn't contribute directly to reaching the goal:

```
(with-auxiliary-goals local-vars
  (prepare <prepare-goals>)
  (perform <body>)
  (clean-up <clean-up-goals>))
```

The control structure separates the auxiliary goals (`prepare-goals` and `clean-up-goals`) from the `body`. Additionally local variables needed only for the execution of the auxiliary goals can be defined in `local-vars`.

Because it is often extremely difficult to find good heuristics for such secondary tasks, `with-auxiliary-goals` indicates appropriate places for improving plans by transformations, for example when the robot needs several objects from the same cupboard (which can be determined using the construct `with-designators`), it can consider leaving the cupboard open in the meantime.

Since the plans are required to work in almost all circumstance, we include in our plans only those steps and conditions, which are highly probable to occur and which we humans normally also take and check. For example if we grip a object, we check if it is inside a cupboard or not, but we don't check if the fuse is working before turning on the cooker.

3.4.3 General Plan Framework

All default plans in the plan library make extensive use of these additional constructs. They make it possible that plans follow a rough common structure separating the principal plan from failure monitoring, failure recovery, or the execution of auxiliary goals. A typical plan structure is illustrated in Figure 3.5 and Figure 3.6 shows as an example the plan for picking up an object[2]. This structure is not mandatory for plans, but it has proved to be useful for many kinds of plans.

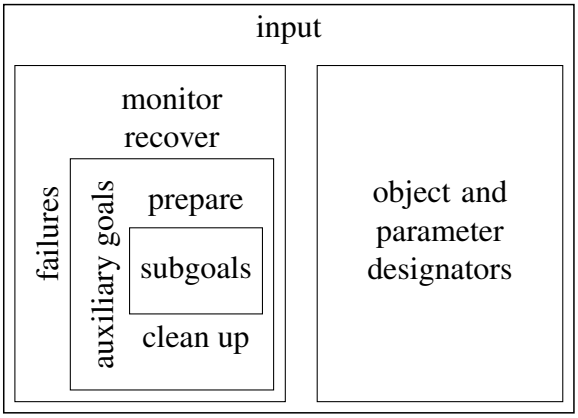
Usually, the outermost declaration is `with-designators` declaring the designators (both for objects and parameters) used in the plan. Note that the designators are not instantiated at this time. The construct only defines the designators symbolically and says that these designators will be used in the plan.

[2]Detailed explanation and the complete code (Listing 3.1) follow in the next section

The next step is usually a `with-failure-handling` construct. Since most steps in a plan are prone to failures, the failure handling is best located as far outside the actual plan steps as possible. For recovering from failures, one common reaction is to execute the body of the plan a second time, either directly or after some repairing steps have been performed. For this method to work, the plans are required to find out for themselves which steps have been performed successfully before the failure occurred and which are still to be achieved. For example, when the robot is to carry two cups to the table and drops one of them. When restarting the plan, the first goal is to pick up the first cup. But this cup is already in the robot's gripper, since it has only lost the second one.

Inside the failure handling construct the body of the plan is executed. This usually involves using the `with-auxiliary-goals` and `at-location` constructs. The nesting of these constructs depends completely on the purpose of the plan, so that it is hard to make a general statement about their usage.

Figure 3.5 Typical structure of the plans in the library.



3.5 Example

Now we have described all the relevant aspects of our plans. As an example, Listing 3.1 shows the plan for achieving the goal of picking up an entity and Figure 3.6 summarizes the important aspects of the plan.

The plan gets as inputs `entity-desig`, a designator describing which object should be picked up, and `designators`, containing a list of object and parameter designators which are merged with the designators defined in lines 2–7 by the construct `with-designators`. The control structure checks if a designator is already defined in the variable `designators` and in the true case it merges the conditions provided by the calling plan and the current plan, otherwise a new designator is created. For example `hand-desig` (line 2) has no

constraints in this plan, but a top-level plan could have added the condition only to use the left arm, and the sub-goal for gripping the object (lines 41–42) will add more constraints, like the hand being free and that the object can be gripped with this hand.

Another designator defined is `pick-up-loc-desig` (lines 3–4), which describes a location where the robot should position itself so that it is possible to grip the object from this location. The designators specified in lines 5–7 set maximum values for how often certain failures are tried to be recovered before forwarding the failure to the calling plan.

For easier understanding we explain the rest of the plan from the inside to the outside. The goal achieving steps are listed in lines 27–29 and 38–47. These steps specify that in order to achieve the state of holding an object described by `entity-desig` the robot has to (1) find the object described by the designator and if the object is not already in its hand (2) position itself at a position such that the object is in the working space of one of its arms and ensure that the robot stays there (lines 38–39); (3) get the object in its gripper;

Figure 3.6 Plan for picking up an object described by the structure of Figure 3.5.

entity-picked-up

inputs: (1) entity

designators: (1) pick-up location
(2) carry tries
(3) grip tries
(4) at location tries

recover: (1) entity-lost-failure
(2) grip-failure

monitor: (1) entity-lost-failure: when lifting entity

prepare: (1) enclosing-container-opened
(2) supporting-entity-extended

subgoals: (1) robot-at-pose: achieved by `at-location`
(2) entity-gripped
(3) entity-lifted

clean-up: (1) supporting-entity-retracted
(2) container-closed

description: Search entity described by designator. If entity is already gripped do nothing. Otherwise grip and lift entity at a suitable location.

Listing 3.1 Plan for picking up an entity.

```
(define-plan (achieve (entity-picked-up ?entity-desig) !?designators)
  (with-designators ( (hand-desig '(some hand))
                      (pick-up-loc-desig
                       '(some location (gripable-from ,entity-desig)))
                      (carry-tries-desig '(some value (maximum 3)))
                      (grip-tries-desig '(some value (maximum 3)))
                      (at-loc-tries-desig '(some value (maximum 3))) )
    (let ( (object-in-hand? nil) )
      (with-failure-handling failure
        ( (carry-tries-count (value carry-tries-desig))
          (grip-tries-count (value grip-tries-desig)) )
        (recover ( (typep failure 'entity-lost-failure)
                   (handle-plan-failure carry-tries-count
                     :entity entity
                     :do-always ( (signal-entity-gone entity-desig) )
                     :do-retry ( (recover-arm-pos hand-desig) )) )
                 ( (typep failure 'grip-failure)
                   (handle-plan-failure grip-tries-count
                     :entity entity
                     :do-retry ( (recover-arm-pos hand-desig) )) ))
        (monitor (scope (begin-task carry) (end-task carry)
                   (whenever (not object-in-hand?)
                     (fail :class entity-lost-failure
                           :entity entity-desig
                           :hand hand-desig))))
        (perform
          (:tag find-entity (perceive entity-desig))
          (unless (or (desig-eq entity-desig (left-hand-entity))
                      (desig-eq entity-desig (right-hand-entity)))
            (with-auxiliary-goals ( (enclosing-entity nil)
                                    (supporting-entity nil) )
              (prepare
                (setf enclosing-entity
                  (achieve (enclosing-entity-opened entity-desig)))
                (setf supporting-entity
                  (achieve (supporting-entity-extended entity-desig))))
              (perform
                (at-location ( pick-up-loc-desig
                               :max-tries at-loc-tries-desig )
                  (:tag grip
                    (achieve (entity-gripped entity-desig)
                      :hand-desig hand-desig)
                    (setf object-in-hand?
                      (get-object-in-hand-fluent-net
                        entity-desig hand-desig)))
                  (:tag carry
                    (achieve (entity-lifted entity-desig)))))
              (clean-up
                (achieve (entity-retracted supporting-entity))
                (achieve (entity-closed enclosing-entity)))))))))))
```

and (4) lift the object into a position that is safe for subsequent plans (normally the object is lifted a few centimeters).

These are the steps, which the robot has to perform definitely for achieving the goal. In lines 32–36 the prepare steps and in lines 48–50 the clean up steps of auxiliary goals are specified. Prepare steps are to open the enclosing container (e.g. a cupboard door) if the object is inside a container and to extend the supporting entity (e.g. a board[3]) the object is placed on. Clean up steps include retracting the supporting entity and closing the container. The container and the supporting entity are stored in local variables specified at the beginning of `with-auxiliary-goals` (lines 30–31)

The plan is made robust by wrapping it inside the `perform` part (line 26) of `with-failure-handling` (line 9). Failure monitoring is done in lines 21–25. Only while lifting the object (the failure monitoring is restricted by the `scope` construct) the plan signals an `entity-lost-failure` whenever the fluent `object-in-hand?` becomes `false`. The `object-in-hand?` variable is defined in line 8 and a fluent is assigned to it in lines 43–45. The separation of the definition and the assignment is necessary, because the variable has to be known in the monitoring code, but can only be set to the correct fluent after the object is gripped. Before the robot hasn't gripped the object the hand to be used is unknown and the according designator `hand-desig` is not instantiated.

Failure recovery is done in lines 12–20. The plan catches two types of failures, other failures are propagated to the calling plan. In the case of an `entity-lost-failure` the plan is retried, but only if the value of `carry-tries-count` is greater than zero. `carry-tries-count` is initialized in line 10 with a maximum value specified with the designator `carry-tries-desig`. When an `entity-lost-failure` has occurred the internal knowledge about the entity is always updated (`signal-entity-gone`) and in the case of retrying the plan the arms are moved to a position, in which the plan can safely be executed again. The plan also recovers from a `grip-failure` not monitored by itself. This failure is signaled by the subplan for gripping the object. Again, the plan is retried if `grip-tries-count` is greater than zero and before retrying the plan the arms are moved to a safe position.

3.6 Related Work on Plan Representation

Many ideas on designing routine plans originate from the work on *XFRM* (McDermott 1992b) and *SRC*s (Structured Reactive Controllers) (Beetz 2000; 2002a). Beetz tries to make some of the implicit design practices of *XFRM* explicit and proposes them as design principles. Transformable plans were the subject of the work by Beetz (2002b), who extended RPL to create plans that can automatically be revised. The language presented

[3] We made the boards movable, because it's easier for the robot to manipulate objects inside cupboards if the boards are extended.

in this chapter offers an even richer language for plan representation including semantic annotations. Besides, our household domain is a lot more complex as opposed to the abstract simulation and the office robot used by Beetz.

Research on *PRS* (Ingrand, Georgeff, and Rao 1990; Ingrand, Georgeff, and Rao 1992; Ingrand et al. 1996; Myers 1997), *TDL* (Simmons and Apfelbaum 1998) and the *Architecture for Autonomy* (Alami et al. 1998) focus on the design and implementation of plan execution languages that incorporate failure monitoring and recovery as well as concurrent execution. Applications include fault diagnosis and control of spacecraft (Georgeff and Ingrand 1989) and mobile robots (Georgeff and Lansky 1987). However, the representation doesn't take into account the transformability, because these systems don't make use of transformational planning.

Williams et al. (2003) developed the Reactive Model-based Programming Language (*RMPL*) to represent reactive plans. *RMPL* includes model-based programming features for making the plans more robust. The language was mainly employed in spacecraft applications. Again, plans that are transparent and explicit to the planner in the way that our representations are, is not a focus of William's research.

An early approach to monitor failures is presented by Fikes and Nilsson (1971), where tables assigning plan execution states to expected world states are kept and compared to the happenings in the environment.

Another approach to include reactivity, failure handling and situation dependent actions are universal plans (Schoppers 1987). Universal planning integrates goal-directed planning with situation-driven reaction by precomputing a plan that has a set of relevant actions for each possible world state and goal.

Robustness was also a main issue in the *RAP*s system (Firby 1989; Firby, Prokipowicz, and Swain 1995), one of the best-known reactive planning languages. By formulating expectations about the outcome of plans, the system could compare the desired and actual behavior of its activities.

Classical plan languages like *PDDL* (McDermott 2000), the language used in the annual planning competition are also evolving into the direction of representing uncertainty and temporal relations (Fox and Long 2006). Still the approach is very different to the reactive planning approach, which uses richer control structures, failure monitoring, and parameter representations.

3.7 Summary

In the beginning of the chapter we have defined the design issues we should bear in mind when developing a language for plan description and coding the plans: robustness, generality, transformability, and enabling cognitive capabilities. Later we have presented several extensions of the plan language RPL that contribute to achieving these demands.

Showing cognitive capabilities includes the ability to explain the course of action taken and that the robot's activities are comprehensible to humans. The symbolic representation of objects using object designators contributes largely to this goal. On the one hand, the robot can explain what kind of object it is dealing with on a level that humans can understand. On the other hand, the robot shows intelligent behavior by not giving up a task just because a certain object cannot be found, but adroitly looks for alternatives. Besides, the semantic annotations of plans help to make the robot understand what it is doing and why. So if a human asks why the robot has closed a cupboard door, it can answer that this is an auxiliary goal, which facilitates subsequent actions.

Robustness requires the robot to recognize failures and react intelligently. The construct `with-failure-handling` offers both components and enables sophisticated failure handling that keeps track of former recovery trials. The failure concept of RPL allows to treat failures on different levels of abstraction in the program. Additionally, with the concept of designators, the robot is free to reconsider former decisions. For example when an object is to be placed at a certain position and this location is already occupied by another object, a higher-level plan can deduct another position that fulfills the high-level designator description.

Generality is the quality of a plan to work in all instances of a certain class of environments, for example in any kitchen. Here again designators are a vital concept to describe activities on an abstract level, to be replaced by more tangible values when executed in a certain environment.

Finally, the plans must be represented in a way that they can easily be transformed. We have introduced a general plan framework that makes it possible to develop general transformation rules that are applicable to a wide range of plans. The constructs providing semantic annotations are especially important for understanding the purpose of plan parts. In particular, the failure handling construct clearly separates failure recovery from the main activity and thus ensures that failure recovery code is not destroyed in transformations.

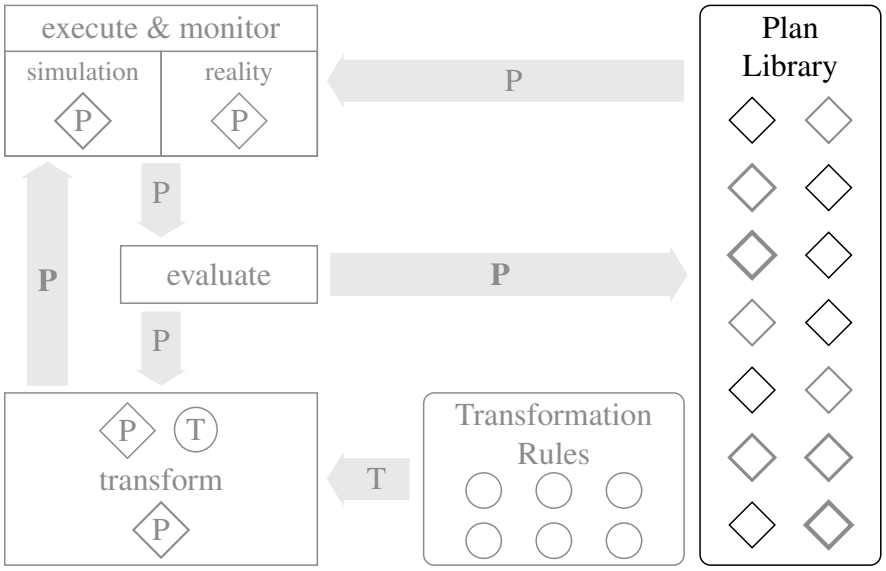
Chapter 4

Plan Library

For the whole plan transformation system to work, the initial plan library (Figure 4.1) must contain general plans that work in any kitchen and without assumptions about the situation they are executed in. The general structure of such plans has been described in the last chapter. Based on this structure and other characteristics of plans we now present how the plans are organized in the plan library.

We first introduce the hierarchical structure of the library based on a detailed example. Then we explain the hierarchy levels at greater length. The combination of plans both in a hierarchical structure by calling subplans and in a sequential order poses many problems

Figure 4.1 Part of TRANER described in this chapter.



in real world scenarios that haven't been subject to extensive research yet. We point out these issues and hit at how they can be solved by plan transformations. Finally, we provide links to related work and give a summary of our plan library.

It is worth mentioning here, that we are only aware of one research project that aimed at the development of a library of general plans applicable to different contexts of the cleanup office task of the IJCAI robot competition 1995 — the *Chip* robot of the Animate Agent Project (Firby et al. 1996). In this project, however, plan revision was not investigated.

4.1 Plan Hierarchy

For an illustration of what a plan library for everyday activity must comprise, let us have a closer look at activities of the household robot. From the view of a user, it must be able to accept abstract commands like "set the table!" or "prepare pasta!". These represent the most abstract class of goals[1] contained in our library, named *activities*.

Now let's try to fulfill the user's desire to prepare pasta. From a high-level perspective we would describe the activity by actions like "put the pot onto the cooker" or "fill the pot with water". We expect the robot to be able to perform these plans in any situation without special preconditions. These are what we call *higher-level manipulation* plans. This class of plans also includes less abstract plans. For example, to fill a pot with water, the robot must be able to bring the pot from one location to another, which involves picking up the pot, carrying it, and putting it down. All of these plans are general enough to work in any situation in the kitchen. For bringing an object from one place to another, for example, the robot may already be holding the object or not even know where it is located.

The higher-level manipulation plans rely on specific plans to manipulate objects, like gripping, lifting or releasing them. These are less robust with respect to different environment situations than higher-level manipulation plans. For example, when gripping an object, the object must have been identified before and the robot must be at a location from where it can reach the object. Otherwise, the plan fails. This class of plans is called *basic manipulation* plans.

On the lowest level of the hierarchy are *basic goals*, which constitute the connection to the robot and the environment. These goals control the robot's base, arms, grippers and camera with position commands or translational and rotational velocity commands to the joints. The goal values are sent as commands to the hardware[2].

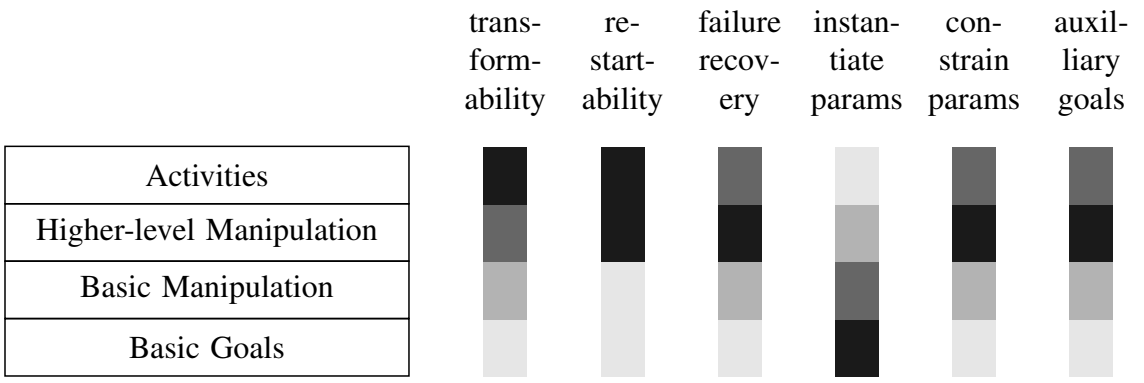
[1] We use the terms "plan" and "goal" interchangeably. In fact, a plan is necessary to fulfill a goal. Most goals have only one plan, which achieve them. When we mention goals, we usually mean the plan that is to achieve that goal. If a goal can be fulfilled by more than one plan, we explicitly mention it.

[2] Actually commands are sent to Player, which provides a hardware abstraction interface (see Section 6.1).

A summary of the levels and their criteria of differentiation is shown in Figure 4.2. All these levels of plans are represented by the same plan representation language RPL, so that all operations on plans, including plan transformations, can be applied to them. However, the transformation of plans is usually more convenient on the higher levels, whereas the lower-level plans can be optimized by classical learning approaches. We have already mentioned one criterion for splitting plans into different classes: the robustness with respect to preconditions. Plans in the activities and higher-level manipulation levels are restartable since they don't have preconditions. They also tend to be more robust in the light of failures. As they have more information about the overall goals and plan execution, they can apply more sophisticated failure recovery methods, including retrying the plan. The job of the lower levels is usually to instantiate parameter designators with concrete values like coordinate positions or the arm used for gripping, whereas the more informed higher levels add constraints to the designators in order to calculate a better informed value (for an example of constraining and instantiating designators see Section 4.3). Auxiliary goals are most commonly found in the higher-level manipulation plans and the activities.

This classification of plans is not a strict separation. But as we represent all plans in a common way, it provides a good indication of their characteristics. As indicated in the introductory example, the plans are not restricted to only have subplans from lower-level plans of the hierarchy. The calling structure usually moves from abstract goals downward, where often several plans from the same category are involved. In principle, even a basic goal could have an activity as a subgoal.

Figure 4.2 Plan library levels and their characteristics. Darker colors indicate that a criteria is more common in a level.



4.2 Plan Categories

In the following we describe the categories of our plan hierarchy in more detail and give examples of the plans pertaining to each class. At the end of each category follows a list of plans belonging to the category with short descriptions. For every goal the necessary input designators are given. Optional designators, which are collected in the variable `designators` and merged by the construct `with-designators` as explained in Section 3.5 are omitted. But it is important to note, that these optional designators enable the plans to constrain designators in higher-level plans and instantiate them in lower-level plans at execution time (see Section 4.3).

4.2.1 Basic Goals

In our robot architecture, which is described in Chapter 6.1, we use an abstract hardware interface provided by the Player open source project. In this abstraction layer basic skills such as moving joints, controlling the robot's velocity, navigation or inverse kinematic calculations are implemented. The lowest level of plans is almost a straightforward mapping of the goals to the commands provided by the Player interface.

These goals can be divided into goals for moving the robot's base, arms, grippers and camera. Base goals are turning to an angle and navigating to a position either with a specified orientation or not. Arm goals comprise setting the angle of each joint separately, moving the arms end effectors to a three dimensional position (including the orientation) either synchronously or separately. Goals for the grippers are setting the position and velocity of the fingers. For the camera changing the pan, tilt or zoom is possible.

All these basic goals get many parameter designators, like the goal position of the robot/arm, the maximum turning and driving velocity, intermediate waypoints for navigation, or trajectory points for moving arms. The designators are instantiated at this level and the resulting values are sent as commands to the hardware abstraction layer.

Summary of Basic Goals:

(`robot-set-vel` *`velocity`*) Set the *`velocity`* of the robot.

(`robot-at-angle` *`angle`*) Turn the robot to the given *`angle`*.

(`robot-turned` *`degree`*) Turn the robot by the specified amount of *`degree`*.

(`robot-at-point` *`point`*) Move the robot to the given *`point`* ignoring the orientation.

(`robot-at-pose` *`pose`*) Move the robot to the given *`pose`*, including x- and y-coordinates, and the orientation.

(`hand-at-pose` *`side pose`*) Move one arm (determined by *`side`*) of the robot to the given *`pose`*. The *`pose`* is a three-dimensional position and orientation.

(`hand-fingers-set-vel` *`side velocity`*) Set the *`velocity`* of the fingers of the arm given by *`side`*.

(`hand-fingers-at-pos` *side position*) Set the *position* (opening distance) of the fingers of the arm given by *side*.

(`arm-joint-at-angle` *side joint angle*) Turn a *joint* of one arm (specified by *side*) to the given *angle*.

(`camera-ptz-at-angle` *pan tilt zoom*) Set *pan*, *tilt* and *zoom* of the camera.

4.2.2 Basic Manipulation

Using the basic goals, we implemented a set of basic manipulation skills serving as subgoals for several higher-level plans. Such goals include very simple activities like opening and closing the gripper and turning the wrist. More complex plans in this category are moving the arm to a save position for navigation, which we call "idle pose" and depends on the currently gripped object, navigating the robot to a position where it is save from a specified cupboard door when opened, and object handling plans like moving, gripping, lifting, dropping, and unhanding objects.

Obviously, these plans perform tasks of different complexity. The main characteristics to differentiate them from higher-level plans are (1) that they depend on certain preconditions to hold and (2) that they supplement most free parameter designators for execution in the real world. This means that they are more specific and less flexible than higher-level plans. They operate on a local view of the execution context. In many cases, especially the ones for handling objects, the goals are achieved by different plans depending on the object. For example some objects must be gripped with both hands, for others one hand suffices.

In contrast to the higher-level plans to achieve the goals on this level of abstraction, certain conditions in the environment must hold. For example, the grip goal only makes sense when the robot has already found the object and is at a suitable position to grip it. The plans for gripping don't take care of looking for objects, they fail if the object is unknown. The identification of objects must be done in higher-level plans. In general, these basic manipulation plans have only few possibilities to react to failures. They monitor their execution and throw failures, but most failures have to be dealt with on more abstract levels.

The goal `entity-gripped` is a typical representative of a basic manipulation goal. There are three plans available for achieving the goal. One plan grips objects with both hands, another one uses only one hand, and the last one performs a handing over which is explained shortly. Which plan is used depends on the object to be gripped. Figure 4.3 shows the abstract description of the plan for gripping an object with one hand. Every designator in the example depends on the entity to be gripped. Since objects in a kitchen are very diverse there are many different (learned) heuristics available to instantiate the designators. The arbitration mechanism chooses the appropriate heuristic and calls it.

The *gripping hand side* designator chooses an arm to use for gripping, depending on the kind of object to be grasped, the position of the object and the robot, and if the arm is currently unused. A cup can be gripped at the handle (front or back), at the cup's base or from the top. Which position is used is specified by the designator *gripping pose*. Which trajectory the arm should take to get from the current position to the *gripping pose* is determined by the designator *gripping trajectory of arm*. For the trajectories we use manually implemented heuristics, but we are on the way to replace these with a motion planner. The last two designators *closing velocity of grippers* and *gripping end force* decide how fast the grippers should be closed and what force should be applied afterwards to hold the object in the robot's grippers.

The last five basic manipulation goals described in the following summary have the same preconditions. They assume that the `entity` has already been searched and identified and that the robot is at a position where it can manipulate the object or reach the given pose with its arms. These goals have at least two plans fulfilling them, either handling objects with one or both arms. A third plan, handing over objects, exists for the goals

Figure 4.3 Plan for gripping an object with one hand. The description is based on the typical plan structure presented in Figure 3.5.

entity-gripped-with-one-hand

inputs: (1) entity

designators: (1) gripping hand side
(2) gripping pose
(3) gripping trajectory of arm
(4) closing velocity of grippers
(5) gripping end force

recover: (1) grip-failure: call goal hand-at-recover-pose and return failure

subgoals: (1) hand-at-pose
(2) hand-gripping

description: For gripping the given entity move one hand (designator 1) to a suitable gripping position (designator 2) by following a trajectory (designator 3). Then close the gripper with a certain velocity (designator 4) and ensure a necessary gripping force (designator 5) at the end. If a grip failure occurs move the hand to a safe recover pose and return the failure to the calling plan. This plan assumes that the object is already identified and that the robot is at a location where it is possible to grip the object.

`entity-gripped` and `entity-dropped`. For instance the handing over plan is necessary when a wooden spoon is needed for stirring (see Figure 4.4). If the grasp mode wouldn't be changed the wooden spoon is lost with high probability during the stirring activity. Particularly the higher-level manipulation goals `entity-picked-up`, `entity-put-down` and `entity-turned` use these last five goals.

Summary of Basic Manipulation Goals:

(`hand-gripping` *`side`*) Close fingers of arm described by the *`side`* designator until the contact sensors of both fingers are activated, otherwise throw a `grip-failure`.

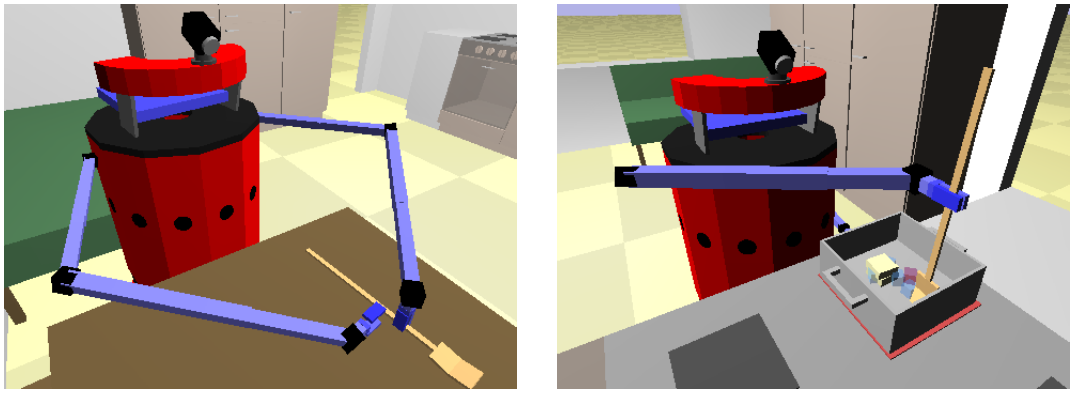
(`hand-closed` *`side`*) Close fingers of specified arm *`side`* completely. If the grippers can't be closed because of a collision detected by the contact sensors, throw a failure.

(`hand-at-idle-pose` *`side`*) Move arm *`side`* to an appropriate idle pose for safe navigation. The idle pose depends on the currently gripped object and is also different if no object is in the hand.

(`hand-at-recover-pose` *`side`*) After for example a `grip-failure` has occurred, open the fingers and move the arm *`side`* to an idle pose, but stop executing as soon as the hand has no contact with an object any more. This ensures a safe state for following plans, otherwise it is possible that with the next arm movement the object is knocked over.

(`hands-at-safe-navigation-pose` *`pose`*) If the robot has to cover a distance greater than a certain tolerance to reach the given *`pose`*, the arms are moved to the idle pose by calling the goal `hand-at-idle-pose` for every side. This ensures that the arms do not collide with other objects while the robot is moving.

Figure 4.4 The robot has to change the kind of grip used for the wooden spoon, in order to be able to stir the content of a pot.



(`wrist-turned` *side degree*) Turn the wrist joint of an arm given by *side* by the specified amount of *degree*

(`robot-outside-door-range` *door*) Ensure that the robot and its arms are outside the *door* range when afterwards the *door* is opened or closed.

(`entity-gripped` *entity*) Grip *entity* either with one or both hands. For details see Figure 4.3

(`entity-lifted` *entity*) Lift the gripped *entity* to ensure that following plans don't lose it again when the arm is moved and friction prevents the object to move along smoothly.

(`entity-dropped` *entity pose*) Put down the *entity* at the given *pose*, but still keep the object gripped[3]. Throw a failure if the object is lost while setting it down.

(`entity-unhanded` *entity*) Open the fingers to release the already set down *entity* and move the arm to a safe position where following plans don't knock over the object.

(`entity-moved` *entity pose*) Move the already gripped and lifted *entity* to the given *pose* in three-dimensional space. For example this goal is called by the higher-level manipulation goals `container-content-decanted` and `container-content-stirred`.

4.2.3 Higher-level Manipulation

More abstract goals can be achieved by using the basic manipulation goals and embed them in higher-level manipulation plans. Goals on this level don't have preconditions. The plans take care to analyze the circumstances first and then choose appropriate subgoals. They make extensive use of auxiliary goals, perform failure recovery, are restartable and constrain parameter designators. Thus plans in the higher-level manipulation level fulfill all the design issues discussed in Section 3.1.

Low-end examples of goals in this level are picking up and putting down objects, opening and closing entities such as cupboards, switching on and off devices such as hot plates or the oven, and turning the tap. More sophisticated skills include taking an object from one position to another, filling containers with water, decanting the content of a container into another container, and stirring the contents of containers.

Although the plans fulfilling these goals constrain designators, they usually rely on the lower-level goals to add concrete constraints and instantiate them. But there are other considerations involved in the execution of plans in real-world environments like the integration of auxiliary goals. This is mainly done on this level of higher-manipulation plans.

[3]The name `entity-dropped` is in some way misleading, since the robot doesn't just open its fingers and let the object fall down, but the name `entity-put-down` is reserved for a higher-level manipulation goal.

The plan for picking up objects is an instance of a typical higher-level manipulation goal, where several failures are monitored and recovered from and auxiliary goals are used. This plan is discussed in depth in Section 3.5. The plan for placing an entity at a location is another example of a higher-level manipulation plan and is shown in Figure 4.5.

Summary of Higher-level Manipulation Goals:

(`entity-picked-up` *`entity`*) Grip and lift *`entity`*. For details see Section 3.5 and Figure 3.6 on page 47.

(`entity-put-down` *`entity location`*) Find a suitable position for the robot, put down the *`entity`* at the given *`location`* and unhand it. If the object has the wrong orientation resulting from restrictions of the arms' dexterity, turn the object. Auxiliary goals include opening and closing cupboard doors and extending and retracting boards. Throw a failure if object is lost during navigating to a suitable position.

(`entity-turned` *`entity`*) This goal is achieved by different plans. Every plan first searches the *`entity`*. If a freely movable object is involved, like cups or plates, the *`entity`* is picked up, turned and put down. For doors, hot-plate knobs, water knobs and the tap the functionality is adapted to the object. In the case of doors it means opening or closing them and for knobs turning them in order to operate some device.

(`entity-at-pose` *`entity pose`*) Pick up *`entity`* and move it to the given *`pose`* still holding the object. For instance the goal `container-content-decanted` uses this goal.

Figure 4.5 Plan for placing an object at a location based on the plan structure described in Figure 3.5.

entity-placed-at-location

inputs: (1) entity
(2) location

designators: constrain used arm

recover: entity-lost-failure

subgoals: entity-picked-up, entity-put-down

description Search entity described as a designator. If entity is already at the requested location do nothing. Otherwise pick up the object and place it at the location. Also, if possible add a constraint to the designator choosing the used arm for gripping by giving a hint what the final goal location of the object will be.

(entity-placed-at-location *entity location*) Pick up *entity* and put it down at the given *location*. For details see Figure 3.5.

(entity-on-entity *top-entity bottom-entity*) Place *top-entity* on top of *bottom-entity* by calling entity-placed-at-location. Optionally a location designator describing where the top entity should be placed on the bottom entity can be given. This designator is forwarded to the subgoal. If the location designator is omitted then a default location designator is created.

(entities-stacked entities-list) Stack the objects given in the list of entity designators. The plan determines an order of how to achieve a stable stack and returns a newly created stack designator. The plan recognizes whether the objects are already completely or only partly stacked and skips stacking of these objects.

(entities-unstacked *stack*) Unstack the given *stack*. Optionally a list of goal locations of the objects in the stack can be given, otherwise a suitable location is calculated.

(hot-plate-temperature *hot-plate temperature*) Turn the knob of the *hot-plate* to set the specified *temperature*.

(container-content-decanted *current-container goal-container*) Decant the content of *current-container* into *goal-container*. First *current-container* is picked-up and moved to a decanting pose. In a clean-up step the container is then put down at the original position.

(container-content-stirred *container*) Find a suitable tool for stirring, pick it up, stir the content of the *container* and in a clean-up step put the tool down at the original position.

(container-filled *container content amount*) Fill *container* with the specified *amount* of *content* (e.g. water, pasta). The plan for filling a pot with water places it under the tap, turns the tap and fills the pot with water by opening and closing the tap. In the case of pasta[4] the objects are picked up and put down in the container by calling the goal entity-placed-at-location.

(door-at-angle *door angle*) Turn the *door* to the given *angle*.

(door-opened *door*) Open the *door* by calling door-at-angle and specifying the angle to be set to the maximum possible value.

(door-closed *door*) Close the *door* by calling door-at-angle.

(board-at-pos *board position*) Slide the *board* to the specified *position*

(board-extended *board*) Call board-at-pos to extend the *board* completely.

(board-retracted *board*) Call board-at-pos to retract the *board* completely.

[4]Pasta is modeled as solid cuboids with a certain weight.

(`enclosing-entity-opened` *`entity`*) Used as an auxiliary goal of `entity-picked-up` and `entity-put-down`. Determines the enclosing entity of *`entity`* and calls the appropriate opening goals. The newly created designator *`enclosing-entity`* is returned.

(`supporting-entity-extended` *`entity`*) Checks if *`entity`* is on a supporting entity (e.g. a board) which can be extended, calls the appropriate goal and returns a new designator *`supporting-entity`*.

(`entity-closed` *`entity`*) Closes the specified *`entity`* by calling the appropriate goal. *`entity`* is normally the returned designator of `enclosing-entity-opened`.

(`entity-retracted` *`entity`*) Retract the specified *`entity`* by calling the appropriate goal. The goal `supporting-entity-extended` normally returns the designator *`entity`*.

4.2.4 Activities

Using the higher-level goals, we can implement sophisticated household activities. At the moment, our robot can boil pasta and set the table. From the view of a user, these are the goals or abstract commands the robot should be able to accept and perform. In these activities a great variety of plans on all levels of abstraction are used. Figure 4.6 shows the hierarchy of goals involved in the plan for setting the table. When a goal can be fulfilled by more than one plan, the appropriate plans are mentioned inside parentheses. The goal `table-set` uses `for-all` to repeatedly place the objects on the table, for every person separately. The code of the plan is depicted in Listing 5.3 on page 88 and explained in Section 5.4.1 as running example of the transformation rule library.

The execution of the pasta boiling plan is depicted as a slide show in Figure 4.7 that demonstrates some of the higher-level manipulation plans used in the activity. Additionally, for the pick up and put down tasks, the relative position of the robot towards the object and the arm used for manipulating it are indicated. When gripping the wooden spoon, it slips from the robot's hand. This failure is discovered immediately, so the robot tries a second time and succeeds.

The activities for setting the table and boiling pasta make use of the 22 higher-level manipulation and the 12 basic manipulation plans presented. The hierarchy is 7–9 levels deep, the lowest level consisting of the basic goals provided by the Player interface. In all, the plans used in the activities monitor eight kinds of failures, supplement 27 parameters by hand-coded heuristics or learned functions and set five auxiliary goals.

Activity Goals:

(`table-set` *`persons`*) Set the table for the given *`persons`*. The plan places the objects (plate, cup, cutlery) one by one on an optionally specified *`table`* (otherwise the

Figure 4.6 Subgoal hierarchy of goal `table-set`.

table-set
- for-all
  - entity-on-entity
    - entity-placed-at-location
      - perceive
      - entity-picked-up
        - perceive
        - enclosing-entity-opened
          - door-opened
            - door-at-pos
              - robot-outside-door-range
              - entity-turned *(hinge)*
        - supporting-entity-extended
          - board-extended
            - board-at-pos
        - at-location
          - entity-gripped *(with-one-hand, with-both-hands)*
            - hand-at-pose
            - hand-gripping
          - entity-lifted *(with-one-hand, with-both-hands)*
            - hand-at-pose
        - entity-retracted
          - board-retracted
            - board-at-pos
        - entity-closed
          - door-closed
            - ...
      - entity-put-down
        - perceive
        - ...
        - at-location
          - entity-dropped *(with-one-hand, with-both-hands)*
            - hand-at-pose
          - entity-unhanded *(with-one-hand, with-both-hands)*
            - hand-fingers-at-pos
            - hand-at-pose
          - entity-turned *(with-one-hand)*
            - ...
        - ...
  - ...

normally used table in the household is used). These steps are repeated for every person that will attend the meal.

(`pasta-boiled` *amount*) Boil the specified amount of pasta. The steps necessary include placing a colander in the sink, filling a pot with water, placing the pot onto a hot plate, switching on the hot-plate, boiling the water, adding pasta, waiting until the pasta is boiled, switching off the hot-plate, decant the water and the pasta into the colander and placing the pasta on a serving object.

Figure 4.7 Plan for boiling pasta showing some higher-level manipulation plans as a slide show.

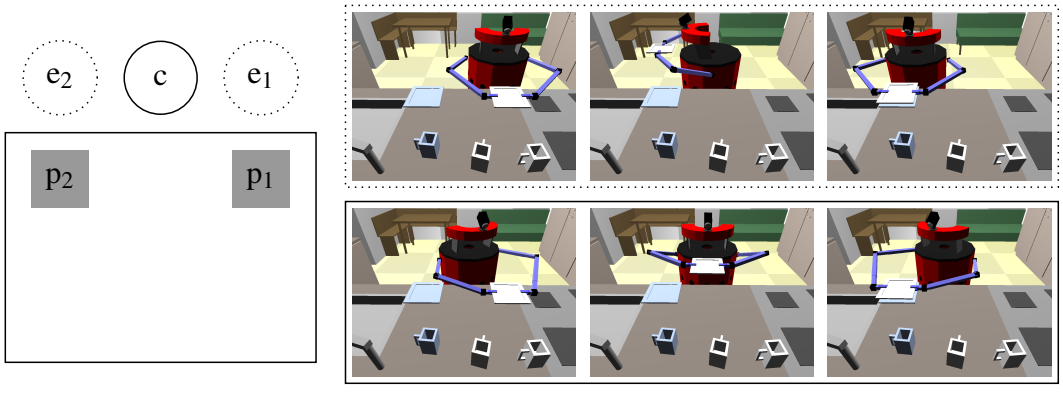
4.3 Combining Plan Steps

We have described the hierarchy of plans needed for sophisticated activities. However, the overall plan performance can not only be explained by the plans involved without having a deeper look at how they work together. The interaction takes place in two ways: (1) the hierarchical decomposition of plans into subplans and (2) the sequential execution of plans. The first gives rise to questions about which level of plans is responsible for constraining and instantiating designators. When executing plans sequentially, the pursuit of auxiliary goals should be rethought. In both cases, the issues raised here haven't been studied in depth, so that no analytical methods exists to solve these problems. To our knowledge, plan transformations are the most general and reliable approach to handle these issues.

Hierarchical Interaction. The first interaction to be mentioned is within the hierarchy. We have required the lower-level plans to replenish the parameters that higher-level plans abstract away from. However, the basic manipulation plans have a very narrow view on the job to be done. They don't have any information what happens before or after they perform their task and can therefore only make local decisions. Often, better results can be obtained when the higher-level plans suggest concrete parameterizations or impose restrictions on the designators describing the parameters, because they have a wider view of the overall goal.

One example of this phenomenon is the choice of the arm the robot should use to grip an object with. The latest point for deciding on the arm is the `entity-gripped` plan. This

Figure 4.8 The top three images show the robot placing a plate from position p_1 to p_2 by picking up the plate at position e_1, driving to position e_2 and putting down the plate. In contrast in the bottom three images the position c is chosen to perform picking up and putting down the plate.



plan bases its choice on the knowledge if an arm is already used and on the object position relative to the robot. What it doesn't know is where the object is to be carried and put down afterwards and if a specific arm is needed for later activities performed before the object is released. Therefore, the grip plan might choose the wrong arm when seen in the overall context. The higher-level plan moving an entity from one place to another is much better informed about additional constraints and can select the hand based on the original object position as well as the target location.

A similar situation arises when the robot chooses the position where to stand in order to grip an object and put it down (see Figure 4.8). When the choice is postponed to the last possible plan, unwanted effects arise when the object's original and target locations are close. In this case, the robot should select a position from where it can reach the object and put it down without the need to navigate in between. The information about the two positions for the object is only available in the plan that carries an object from one location to another.

On the other hand, higher-level plans can more easily be manipulated and are more flexible when they don't have to bother about the low-level details of execution. It is an important research issue to find a good balance between too much detail at higher abstraction levels, uninformed choices on the lower levels, and how transformations can modify plans to add more constraints to the designators to make the plans more robust.

Sequential Composition. Similar trade-offs are necessary for sequential goals. In our plan library, goals are defined by a predicate describing the situation when the goal is reached. For example, the grip goal is reached when the robot's touch sensors detect an object and this object is the one to be gripped. But often, the borderline between different plans is fuzzy. One instance is when an object is put down (see Figure 4.9). From an abstract point of view the goal is reached as soon as the object is sensed at the target location. If the next plan performs another manipulation action with another object, it will probably fail, because the arm knocks over the first, recently put down object.

Figure 4.9 The goal placing a cup at a location is reached from an abstract view as soon as the cup is at the given location. But the robot fails to grip another cup, if it doesn't leave a clean state.

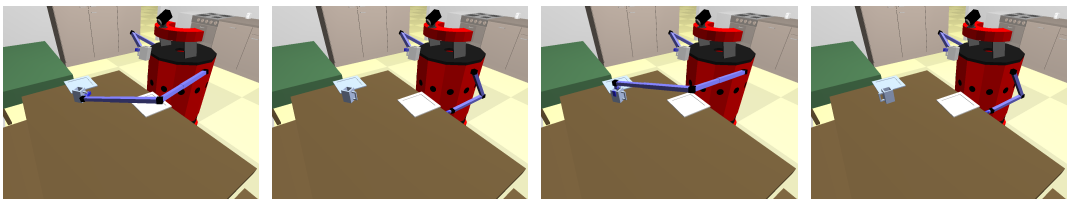
Obviously, we need auxiliary goals that ensure that the next plan has a chance to succeed after a goal has been reached. These actions include moving the arms to the idle pose, closing doors, put back objects that have been shifted to enable the first plan, etc. An even wider view of such actions is described by Hammond, Converse, and Grass (1995). They claim that long-term activity can only be successful when the robot performs "stabilization" tasks like cleaning dishes and storing them away. These activities aren't explicit goals, but must be performed somewhere in between the primary plans.

Like the hierarchical coupling of plans, the questions arising in sequential performance aren't trivial. One possible approach is to ensure after every plan that any subsequent plan finds a set of prespecified conditions, e.g. that all cupboard doors are closed and the robot's arms are in the idle pose. This ensures that all plans can rely on uniform conditions and don't fail just because the previous plan has left the environment in a mess. However, it often gives rise to funny sequences of actions (see Figure 4.10), for example the robot carries a cup, puts it down, moves its arm to the idle pose, moves the arm towards the cup in order to turn it to its target orientation, and moves the arm back into the idle pose. Because the plan putting down the cup is eager to leave a good starting situation for the next action, it makes the whole process very inefficient, because the hand is needed at the cup for the next action.

A similar situation occurs when the robot takes two cups out of a cupboard (see Figure 4.11), one in each hand. It opens the cupboard, takes out the first cup, closes the cupboard, opens the cupboard, takes out the second cup and closes the cupboard again. As we have pointed out, leaving the cupboard open in general is not a good solution as it might hinder the robot to perform later actions. Another solution would be to integrate the opening and closing activities in a higher-level plan, in this case the one that tries to take both cups. But this doesn't solve the problem in general. There might be other things to be taken from the same cupboard, so that the closing should be performed on an even higher level. On the other hand, when there are people working in the kitchen, it might make sense to close the door more often, in order not to hinder others in their activities.

It is extremely hard to find a general solution to this problem and the more abstract the cleaning up activity, the more difficult it gets to integrate it into the overall sequence

Figure 4.10 Sequence where the arm is moved too often to the idle pose. Between putting down the cup and turning the cup, the idle pose isn't needed.



of actions. In our plans, we differentiate between the robot's state and that of the environment. Each plan takes care that the robot is in a state that makes the execution of the plan possible before it is started. For example, when navigating the robot checks on the distance it is going to move and decides if the arm should stay in the position it currently is in or should be moved to the idle pose (for longer distances). In contrast, the situation in the environment is stabilized after a goal has been reached. So when the robot takes something out of the cupboard, it closes it afterwards. This is described with auxiliary goals and for the default plans this is a desired behavior, because it ensures that the robot can use them in every environment. By using plan transformations the default plans are then adapted to find optimal times to perform auxiliary goals and to omit unnecessary plan steps (see Chapter 5).

Figure 4.11 Sequence where the cupboard door is closed and opened again between taking two cups of the cupboard

4.4 Related Work on Plan Libraries

As already mentioned in the beginning of the chapter, our plan library is similar to the work by Firby et al. (1996), who describes the components and the general plan library of the robot *Chip*, whose task was to clean up trash in an office environment. They use Firby's (1989) *RAP*s language for their plans, which is a direct predecessor of RPL, but doesn't enable plan transformations.

Sussman's (1973; 1977) *Hacker* system uses a similar approach of defining plans in a library and transforming them in the blocks world domain. His library contains plans that have already been used successfully.

The hierarchical execution of plans at different layers of abstraction is also the basic idea of the prominent 3T architectures (Bonasso et al. 1997; Firby 1989; Pell et al. 1997). Other than our system, where the different plan layers are implemented in the same language and the calling structure is not restricted, 3T architectures assume three independent levels of execution, where higher levels pass commands to lower levels. Optimization can only take place in each level, thereby omitting necessary higher- or lower-level information.

Zöllner et al. (2005) build hierarchical task representations in a kitchen environment by observing humans. In contrast to our complete plan library, they focus on the level of basic manipulation skills.

The problem of parameter instantiation is addressed by Agre (1988). His solution, called *deictic representation*, symbolically describes objects and their relation to the agent. The representation of such objects can be performed with designators. However, Agre assumes that no instantiation with low-level values is necessary for the agent to choose its action. This is a feasible approach in Agre's Pengi world (a simple computer game), but too abstract to be used on a robot acting in the real world.

The work on lifeworld analysis performed by Agre and Horswill (1997) offers ideas how to instantiate and, more importantly, how to interchange designator descriptions. The use of predictive models for optimizing partially specified parameters is also demonstrated by Stulp and Beetz (2005).

Besides, the problem of auxiliary goals has been studied under the term "stabilization" of environments by Hammond, Converse, and Grass (1995) as a problem in household domains without offering an implementable solution.

4.5 Summary

In this chapter we have presented our plan library for performing sophisticated household activities. To achieve the desired behavior, we need plans on different levels of abstraction. These abstraction layers cannot be seen as fixed categories, but include plans that are similar with respect to failure monitoring and recovery, constraining and instantiating parameter designators and the use of auxiliary goals.

Although lower-level goals can diagnose failures, they have few means to recover from them. They have to complement the abstract problems with values for execution parameters, which are needed for the execution on the system level. Higher-level plans are more concerned with repairing failures and to ensure that they can be executed robustly at any time without preconditions. An important concept here are auxiliary goals, which are necessary for reliable execution, but are hard to integrate efficiently into the plan.

We have especially pointed out the challenges in combining plans, both sequentially and hierarchically. We discussed the problems of shifting the responsibility for the low-level parameters to basic plans, because they often lack the information needed for an appropriate choice. We have also explained the intricacies of defining when a plan has ended and what the next plan in a sequence may expect from the world state. Plan transformations become a vital tool in making robot activity feasible in the real world.

With the use of auxiliary goals and failure recovery our plans show very robust and reliable performance. They are represented in a uniform way, so that they can be enhanced by plan transformations.

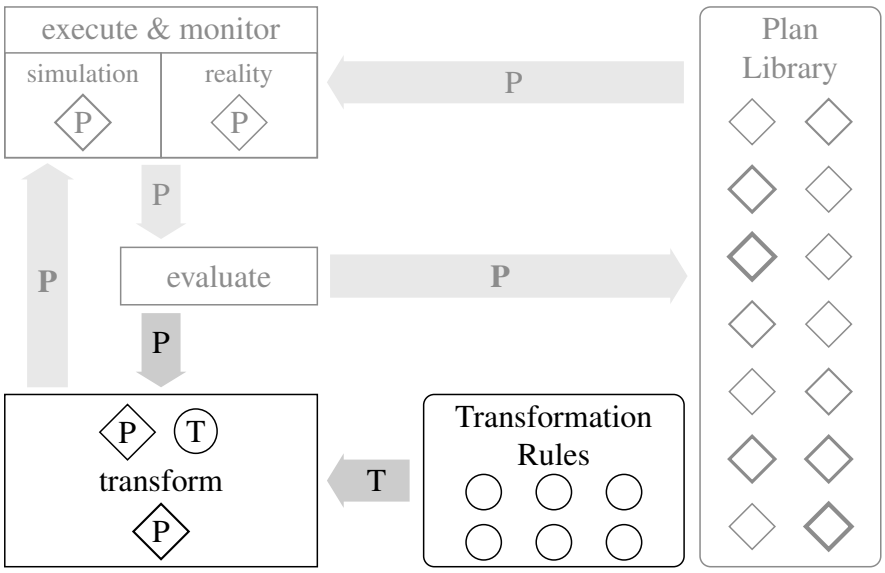
The most abstract activities in our library are currently setting the table and boiling pasta. The higher-level and basic manipulation plans are very general, they are used for both activity plans equally.

Chapter 5

Plan Transformation

We have presented how plans are designed and organized in the plan library. The eponymous component of TRANER, however, is the transformation of such plans. In Chapter 2 we have explained the role of transformation rules in the complete plan execution and transformation process. Now we introduce the concept of transformation rules in detail and show how a plan and a transformation rule result in new plans. The chapter concludes with a description of our complete set of transformation rules.

Figure 5.1 Part of TRANER described in this chapter.



5.1 Motivation

Figure 5.1 highlights the components of TRANER we explain in this chapter: the transformation rules and the procedure of transforming a plan to new ones using transformation rules. First we motivate why a special representation for transformation rules is necessary and what the requirements are. Second we look at the challenges of processing transformation rules.

5.1.1 Rule Representation

Similar to plans, transformation rules must follow a determined syntactical structure, because like plans they must be understood by TRANER to execute the transformation. The transformation rules should be able to perform complex modifications on plans in order to change the course of activity of and the actions taken by the robot significantly. For instance, transformations can suggest to use tools (e.g. a knife or a pair of scissors) or appliances, to leave cupboard doors open or to use containers. These modifications have to be performed on the concurrent reactive plans with complex control structures presented in Chapter 3. This is only possible by using a special transformation language for the rules. Whereas plans are represented in a procedural way, we use a logical notation for transformation rules.

Let's first have a look at some examples. For instance a transformation rule should be able to express things like *if* a plan places objects one by one on another object (e.g. a table), *then* stack the objects first, bring the stack to the goal object and unstack the objects. Another possibility would be to use a container (e.g. a tray) instead of stacking the objects, which requires to add steps for finding and getting the container. A third possibility is, if the type of the object allows it, to carry two objects (e.g. cups) in parallel. Another rule is *if* a plan opens and closes doors multiple times, *then* leave the doors open, but make sure they are closed at the end of the plan.

These examples show that a transformation rule needs a condition (*if*) under which it can be applied and an output part (*then*) specifying the new plan. As the transformation rules must be general enough to apply to a wide range of plans they must specify exactly which assumptions they make about the plan to be transformed without restricting the set of matching plans more than necessary.

Furthermore, transformation rules should not be tailored to a certain robot. They should be general enough to be carried over to several platforms and different problems. To make the rules understandable both to the transformation mechanism and to the programmer, a declarative rather than procedural specification is necessary.

No matter how general the transformation rules are, it is inevitable to rely on certain assumptions about which plans exist in the system and what they do. For example, a transformation rule for stacking objects relies on the existence of the plans `entity-on-entity`

and `entities-stacked` and that they do indeed what their names imply.

The transformations are not as simple as formulated in the examples. For instance in the example where a plan places objects on another object one by one, the transformed plan stacks the objects before carrying them. When now for instance the objects are four plates, should all four plates be stacked, or only three or two? If three or two plates are stacked, which of the four should be chosen? Which combination results in the best plan can only be found out by testing all alternatives and the number of subsets can get large. A transformation rule has to support the generation of multiple alternatives.

To get a better feeling what transformation rules should look like, the left hand side of Figure 5.2(a) shows a simplified version of a plan for setting the table[1]. This plan first binds the necessary designators (*desigs*), like the plates, cups and the table. Then a sequence of steps follows, where each step places either all plates or all cups on the table[2]. A possible improvement of the plan is shown on the right side of Figure 5.2(a) where instead of placing the plates on the table one by one, all plates are stacked first, then the stack is placed on the table and afterwards it is unstacked.

The problem is now, how to get from the left plan to the right plan. A direct approach would be to write rules where the original input plan is replaced with the output plan. When implementing rules this way, we could write all the plans by ourselves, because the generality is missing. Therefore it is necessary that a transformation rule accepts more than one input plan by checking a condition and is able to identify important parts of the input plan. The top part of Figure 5.2(b) shows the code tree[3] of the left plan in Figure 5.2(a). In the example the transformation rule has to match the `(STEP 1)` branch and transform it to the branch shown at the bottom part of Figure 5.2(b). The new branch contains three new subbranches.

Often it is necessary not only to modify one part of the input plan, but also some other parts. The top part of Figure 5.3 shows a simplified code tree of the plan for setting the table. When a transformation rule optimizes the opening and closing of doors, it has to identify the subplans in the tree where the plan closes doors. This happens three times in the example (indicated by circles). The output code tree is depicted at the bottom part of Figure 5.3. Here the closing operations are removed and new steps at the beginning and the end of the plan are added. One parallel step (indicated by a rectangle) is included, which monitors the plan for doors opened by the robot. The second new step closes all opened doors in the end.

Beside being general, transformation rules have to express complex transformations.

[1] The plan is already transformed and not the default plan contained in our plan library. The default plan is shown in Listing 5.3 on page 88 and will be explained in Section 5.4.1. The transformed plan is chosen to simplify the example.

[2] `(achieve (eoe p→t))` is the short version of achieving the goal `entity-on-entity` by placing a plate on the table

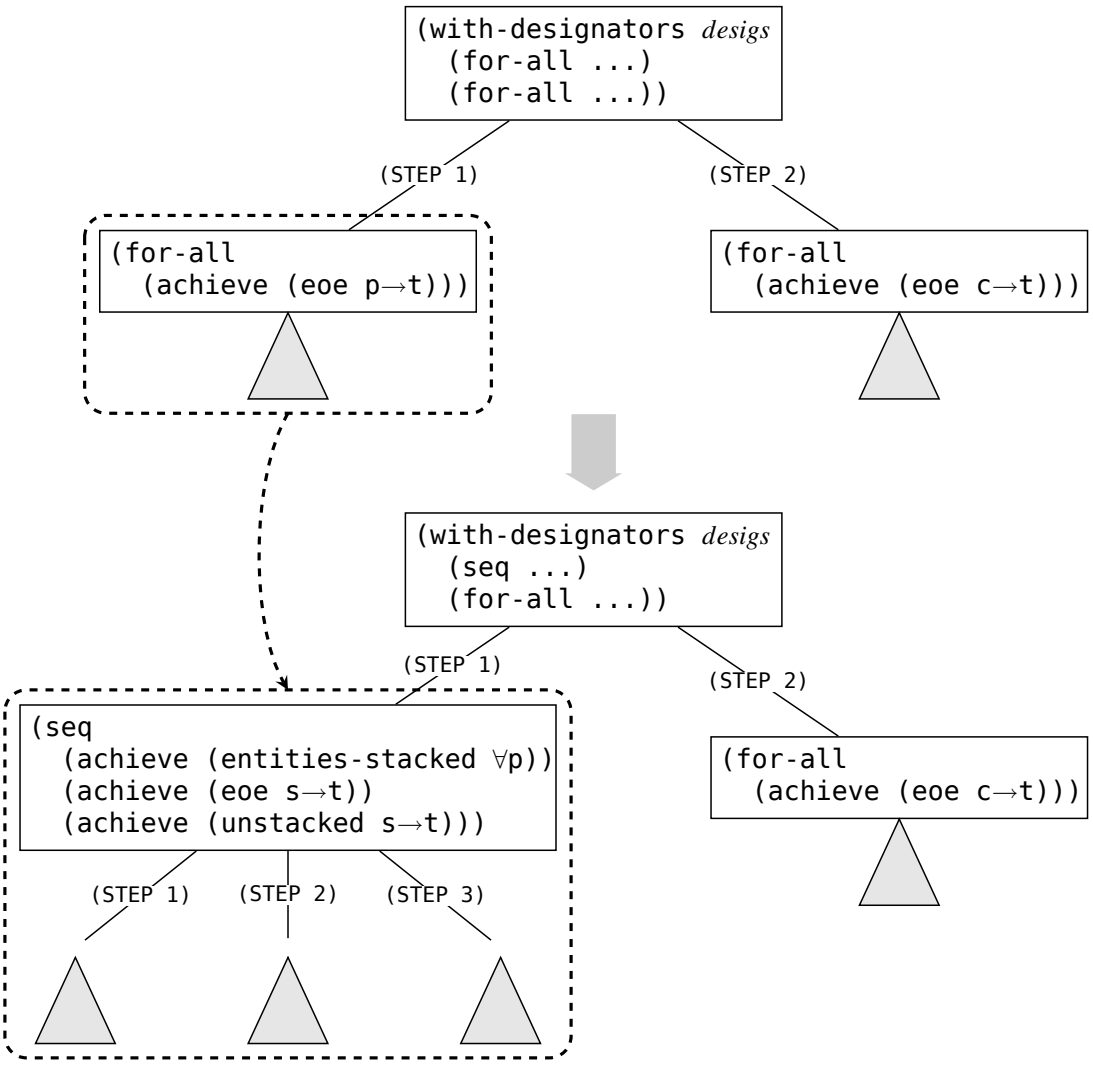
[3] Code trees are explained in Section 3.2.3 on page 37

Figure 5.2 Illustration of a plan transformation.

```
(with-designators desigs
  (for-all
    (achieve (eoe p→t)))
  (for-all
    (achieve (eoe c→t))))
```

```
(with-designators desigs
  (seq
    (achieve (entities-stacked ∀p))
    (achieve (eoe s→t))
    (achieve (unstacked s→t)))
  (for-all
    (achieve (eoe c→t))))
```

(a) Changes necessary to a plan.

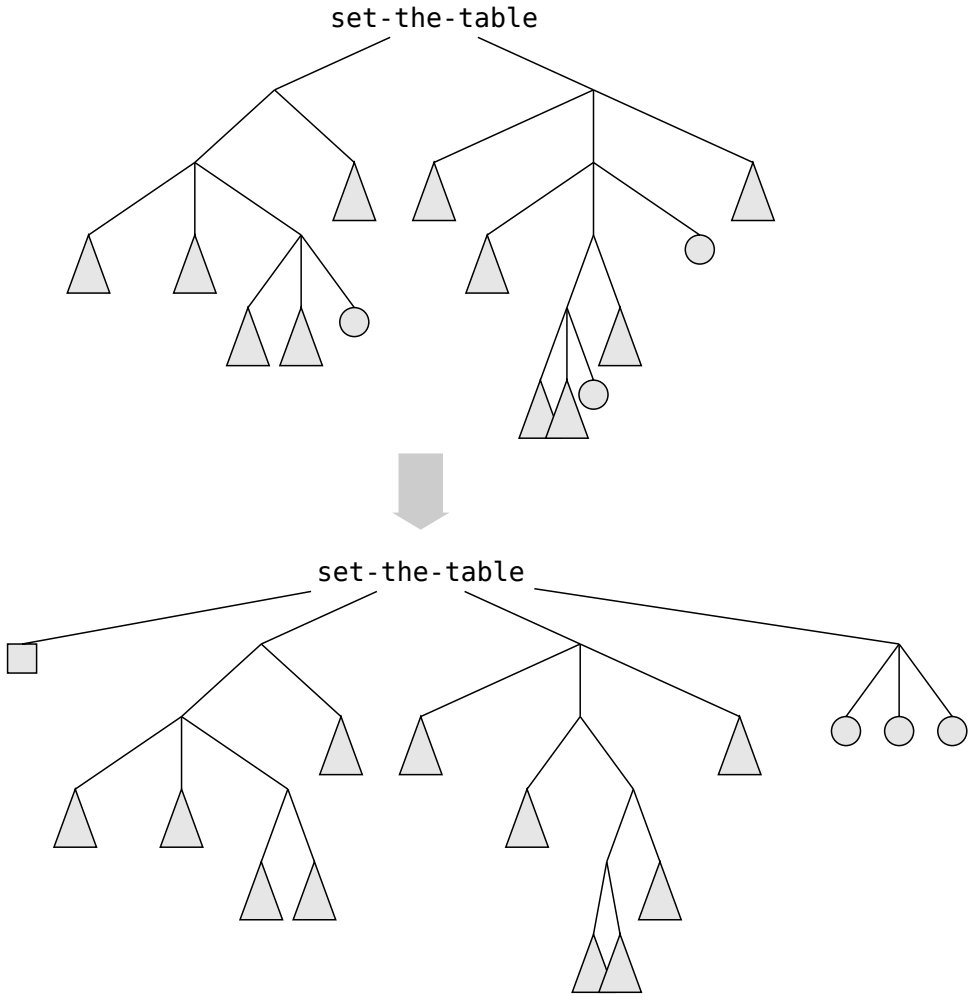


(b) Replacement in the code tree.

They need to describe how the input plan is converted to an output plan by only modifying some parts of the a priori unknown plan and maintaining the functionality of the rest of the plan.

In sum, a transformation rule has to be able to modify the right subplans. This means, that it must be possible to specify all relevant parts of the *input* plan, the *transformation* of the identified parts and the assembly of the *output* plan by replacing the matched parts

Figure 5.3 The top code tree depicts a simplified view of the plan for setting the table. In this plan a cupboard door is closed three times (indicated by circles). In the output plan (bottom code tree) the transformation rule has identified and removed these branches. Additionally a branch monitoring the opened doors (rectangle) is added and at the end plan steps are added to close the opened doors.



with the transformed parts. Because the situations a robot might encounter are not known at the time when the transformation rules are implemented, they must be general enough to comprise a great variety of situations. Currently, we develop transformation rules carefully by hand. For the future we plan to generate transformation rules by comparing the robot's performance to that of humans and extracting possible transformations.

5.1.2 Transformation Procedure

Beside the representation of transformation rules, the processing of the plans by the rules is another important research question. When a plan has shown to work suboptimally, an adequate transformation rule must be selected from the set of available rules. The first criterion is simply if the rule (condition) matches the plan. Each transformation rule assumes a specific syntactic structure of the plan, for example that the plan contains two concurrent subplans. Only transformation rules for which the plan is syntactically valid can be considered. After the matching test there might still be several applicable transformation rules.

In principle, all these rules could be applied to the plan and the resulting plans can be executed and evaluated in simulation. This procedure results in a search tree as shown in Figure 2.4 on page 26. However, the branching factor of the search for a good plan is very large when all valid transformation rules are used. Another criterion is necessary to narrow the search. All rules contain an applicability condition telling in which situations this rule is particularly useful. The condition is based on the execution trace observed when the plan was executed. As an example, consider two transformation rules assuming the plan to be composed of two subplans to be executed in parallel. The first transformation rule transforms the plan in a way that the two subplans are executed sequentially in the resulting plan. The second transformation rule intensifies the parallel execution so that not only parallelizable actions are executed concurrently, but when one plan contains idle times, these are used by the other plan to execute plan steps. Both transformation rules can be applied to plans that contain two concurrent subplans. But the first rule should be applied when the observed data indicates that the parallel actions thwart their goals mutually, whereas the second rule can be used to enhance the robot's performance.

Not only several applicable transformation rules lead to different resulting plans. We have seen that by applying only one transformation rule a plan can be transformed to several other plans by generating alternatives. For instance this happens when only a subset of the objects should be stacked. In this way, one transformation rule is used like a set of transformation rules and this has to be supported when transformation rules are processed.

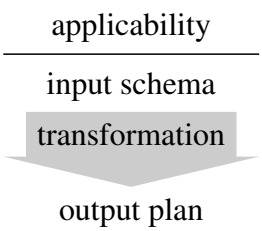
5.2 Transformation Rules

In the following we introduce the representation of our transformation rules first informally in a structural way, then formally by providing the syntax of the rules, each time together with an example. The language is based on logical concepts, but includes the full expressiveness of LISP.

5.2.1 General Structure

We have argued that a sophisticated transformation rule should consider both the observations on the robot's behavior and the syntactical structure of the plan. Therefore our transformation rules, whose structure is depicted in Figure 5.4, are split into a reasoning component and a rule body describing the syntactic operations on the plan.

Figure 5.4 Graphical illustration of transformation rules.



Syntactical Declarations

The syntactic component is subdivided into three parts: the input schema, the transformation, and the output schema. The input schema determines if a plan matches the transformation rule syntactically. This means, it expresses how a plan must be structured in order to be transformable by the rule. For example, a rule might rely on the plan containing subplans or including a sequential execution of several plan steps. A rule whose input schema matches the plan structure is called a *valid transformation rule* for the plan.

An input schema is not just a condition on the validity of the rule, but binds parts of the input plan to variables by pattern matching. As the plans contain subplans, which might have to be transformed in their turn, the whole transformation gets a hierarchical character entailing a more complicated structure of the pattern matching.

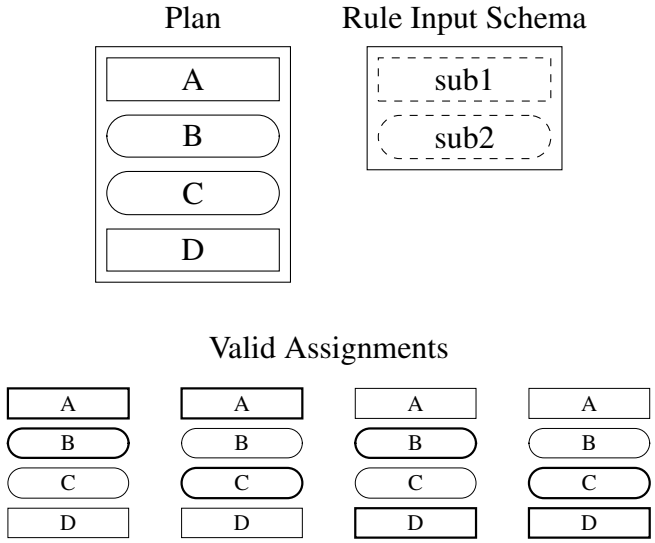
Figure 5.5 illustrates how a transformation rule can be valid for a plan in several ways, because there are multiple valid assignments for the variables `sub1` and `sub2`. Depending on the semantics of the rule, all possible assignments or only a subset might be reasonable

for the transformation. Our syntax enables the specification and restriction of multiple matches.

The matched variables are the input for the transformation part of the rule. The declarations here specify how the code is to be transformed before being reassembled in the output schema. For very simple rules, the transformation part can be omitted, for example when the rule only changes the order of two commands of the original plan. But in most cases, more sophisticated operations are needed, for example to ensure correct variable bindings, adjust designator specifications, and not to lose failure recovery methods.

The result of a transformation rule is determined by the output plan of the rule. It assembles the resulting plans by using variables bound in the previous parts and arranging them to the desired plan structure.

Figure 5.5 A rule input schema matched against a plan, resulting in multiple valid assignments for `sub1` and `sub2`. If the order of the subplans in the input schema is important, only the first two assignment are valid.



Reasoning Component

Pure syntactic transformations lead to a huge search in the space of possible plans, whose branching factor is determined by the number of transformations applied to the plan. For keeping this number small and only trying promising transformations, we don't use all valid transformation rules to produce new plans, but introduce the concept of *applicable transformation rules*. The applicability of a rule is a condition on the execution trace, which was observed when the plan was executed. Such conditions might include the

number and kind of objects manipulated, the traveled distance, or the number of failures during execution. A rule can only be applicable, if it is valid for a plan, that is that it matches the input schema.

This deliberation doesn't only consider if the rule is applicable at all, but also restricts the possible matches of the input schema. Conditions on the observed robot behavior determine which input matches are the most promising ones.

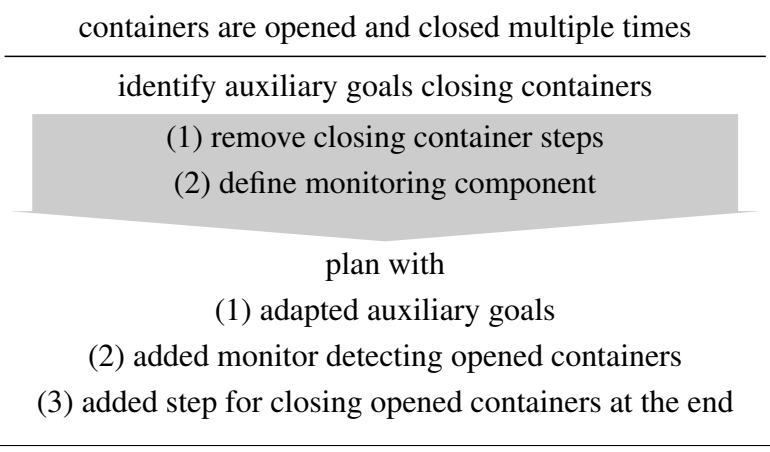
Example

For illustrating the structure of a transformation rule, let us regard a rule which eliminates unnecessary execution of auxiliary goals as shown in Figure 5.6.

The applicability condition states that this rule is only useful for plans, during whose execution the robot has opened and closed containers several times. However, there is no guarantee that the transformed plan will behave better than the original one. First, the applicability doesn't require the containers to be equal, which means that all containers that are opened and closed can be disjunctive. Second, it might not be possible to leave a container open for a longer time, for example because the content might deteriorate or there is not enough room for other activities if containers are left open or the family cat likes to jump into cupboards. On the other hand, the rule discards the transformation of plans that contain auxiliary goals, but the handling of containers is already done efficiently.

The input schema analyzes the plan and identifies auxiliary goals that close containers. These closing operations are then removed by the transformation part. By these operations alone, all containers would be left open in the end. Therefore, the rule defines a new program part monitoring which containers are opened during execution.

Figure 5.6 Transformation rule for optimizing the opening and closing of containers specified as auxiliary goals.



The output plan is constructed by using the original auxiliary goals without the commands for closing cupboards. Then the monitoring component is added, and finally a new auxiliary goal is added at the end of the plan execution, closing all opened containers at the end.

Note that the transformation didn't try to eliminate multiple opening of containers. This is due to the fact that the plan for opening containers is a higher-level manipulation plan, which can operate in any situation. When the container is already open, the plan realizes this and terminates immediately.

5.2.2 Rule Representation

For representing transformation rules in our plan language, we defined a special programming construct `def-tr-rule` along the concepts introduced in the previous section. The syntax specification for transformation rules is given in Figure 5.7 and an example is shown in Listing 5.1. Using the example, in this section the syntax is explained and in the next section we describe how the parts of the transformation rules play together and are executed.

The Basics

Our transformation rules are based on *XFRML* (*XFRM* Language) (Beetz and McDermott 1997; Beetz 2000; 2002a), the transformation language of the planning system *XFRM* (McDermott 1992b), which allows the declarative specification of general transformation rules. *XFRML* itself is based on a Prolog-like Horn clause language with fifty to sixty built-in predicates, specifying relations on plan code and execution traces. With this language it is possible to retrieve information from plans and execution traces and test properties of them by writing Prolog-like queries.

Beetz (2000) describes *XFRML* as "the usual Prolog-in-LISP combination: parenthesized prefix notation, with variables indicated with '?' and segment variables by '!?'." Basic general predicates provided are `(true)` and `(false)`, unification with `(unify` e_1 e_2`)`, arithmetic equality and inequalities (`=`, `!=`, `<`, `<=`, `>`, `>=`), the logical operations `(and` $pred_1$ … $pred_n$`)` and `(or` $pred_1$ … $pred_n$`)`, as well as set operations like `(member el list)` and `(set-of arg pred set)`. The last predicate succeeds if every `arg` in `set` satisfies `pred`.

Besides it is possible to call LISP-code with the predicates `(eval exp val)` and `(lisp-pred pred` x_1 … x_n`)`. The predicate `eval` is true if `exp` evaluates to `val` in LISP and the predicate `lisp-pred` is successful if `(pred` x_1 … x_n`)` returns a value other than `nil` in LISP. New predicates can be defined with `(<- p (and` q_1 … q_n`))`, where `p` is an atomic formula.

Basic Syntactical Elements

As described in Section 5.2.1 a transformation rule has four parts: (1) applicability, (2) input-schema, (3) transformation, and (4) output-plan. This structure is reflected in the construct `def-tr-rule` as shown in Figure 5.7(a). Every definition part is a list of at

Figure 5.7 Syntax specification of transformation rules.

```
(def-tr-rule ⟨name⟩
  :applicability (⟨predicate⟩+)
  :input-schema (⟨input match⟩+)
  :transformation (⟨predicate⟩+)
  :output-plan (⟨rpl code⟩+))
```

(a) Transformation rule definition.

⟨*predicate*⟩ → (⟨pred-name⟩ ⟨*pred-arg*⟩*) |⟨*plan identification*⟩
⟨*pred-arg*⟩ →⟨*expr*⟩ |⟨*predicate*⟩
⟨*expr*⟩ →⟨*variable*⟩ |⟨symbol⟩ | (⟨*expr*⟩+)
⟨*variable*⟩ →⟨symbol⟩ starting with '?' or '!?'
⟨*input match*⟩ → (⟨*plan identification*⟩ ⟨*control command*⟩)
⟨*plan identification*⟩ → (match-plan
:at ⟨*path*⟩
:plan ⟨*variable*⟩
:bind-path ⟨*variable*⟩
:cond ⟨*predicate*⟩)
⟨*path*⟩ →⟨*variable*⟩ | (⟨*path-spec*⟩*)
⟨*path-spec*⟩ →⟨rpl-path⟩ |⟨*variable*⟩
⟨*control command*⟩ →⟨*branch*⟩ |⟨*for-each*⟩
⟨*branch*⟩ →:branch (:generate (⟨lisp-function⟩ ⟨*variable*⟩*)
:unify ⟨*expr*⟩
:cond ⟨*predicate*⟩)
⟨*for-each*⟩ →:for-each ⟨*variable*⟩ :unify ⟨*expr*⟩

(b) Explanation of the expressions of part (a).

⟨*expression*⟩	An expression that is defined later or has been defined.
⟨lisp expression⟩	A construct from LISP or RPL giving a short specification.
a → b	'a' is of the form 'b'.
a | b	Alternative construct, either 'a' or 'b'.
a^+	→ a | aa^+.
a^*	either nothing or a^+.
optional	An optional expression is illustrated in gray.

(c) Explanation of the notation used in part (a) and (b).

least one specification and each list has the same length[4]. When a transformation rule is executed the elements of the lists are traversed in a certain order as shown in Figure 5.8 on page 87 and explained in Section 5.3.

The specification of the applicability condition (lines 2–6 in Listing 5.1), the transformation part (lines 23–28), and the output part (lines 29–42) are straightforward. Each list element of the applicability is a predicate returning true or false. The rule is only used when each applicability predicate returns true. If a list element has to perform several operations, they can be combined as conjunctive predicates. The transformation is also given as a list of Prolog predicates, performing modifications on the matched parts of the input plan. The output plan is a plan where some parts are represented by the variables bound in the input schema and the transformation part. More sophisticated operations on plan parts must have been performed in the transformation part.

Input Schema Matching

The remaining part of a transformation rule — the input schema (lines 7–22) — is slightly more intricate in its definition. The reason is that our plan language is quite complex and therefore a sophisticated language for pattern matching and identifying interesting subparts of plans is required. The most basic possibility to define the input schema is when the interesting parts of the plan can be specified by a known path. Then the specification is `(match-plan :at <path-spec> :bind-path ?path-var :plan ?plan-var)`, where `<path-spec>` is the known path specification. The arguments `:bind-path` and `:plan` are optional. If they are provided, the path in the task tree and the plan found at this path are bound to the respective variables.

The situation is more complicated when the exact path specification is unknown, but some feature of the plan is required, for example the part in the plan where `with-failure-handling` is called. In this case, an additional condition specified as a predicate can be given. The condition can be combined with a complete or incomplete path specification. If a complete path specification is provided, the condition only checks if the plan at this position in the task tree can be transformed by the rule. But it is also possible to give an incomplete path specification and then a search for a plan matching the condition is performed in the subtask tree of the given path. An incomplete path specification is characterized by containing unbound variables.

When working with conditions and incomplete paths, it may happen that several subplans match during the search. If more than one subplan is accepted by the condition, then the first matching plan is used. This behavior is not always desired. In the example transformation rule all occurrences where containers are closed in clean up steps should be removed. To achieve this `match-plan` itself is defined as a predicate, which can be

[4](length applicability) = (length input-schema) = (length transformation) = (length output-plan)

Listing 5.1 Implementation of the transformation rule for optimizing the opening and closing of containers specified as auxiliary goals.

```
(pl:def-tr-rule :containers-closed-at-end
  :applicability
    ( (and ...                                                    ; ap-1
           (> ?containers-opend-count 1)
           (> ?containers-closed-count 1))
      (true) )                                                    ; ap-2
  :input-schema
    ( ((match-plan :at () :plan ?plan                             ; is-1
        :cond (set-of (?path ?bdgs ?prepare ?perform ?in-clean-up)
                      (match-plan :at ?path :plan ?aux-goal
                        :cond
                          (and (unify (with-auxiliary-goals ?bdgs
                                        ?prepare
                                        ?perform
                                        (clean-up ?in-clean-up))
                                      ?aux-goal)
                               (lisp-pred tr-rules-has-goal
                                 ?in-clean-up 'container-closed)))
                      ?aux-goal-vars)))
      ((match-plan :at ?path)                                     ; is-2
       :for-each ?aux-goal-vars
       :unify (?path ?bdgs ?prepare ?perform ?in-clean-up)) )
  :transformation
    ( (transformed-plan                                           ; tr-2
        ?in-clean-up
        (reduction (:achieve (container-closed ?_)) (no-op))
        ?out-clean-up)
      (true) )                                                    ; tr-1
  :output-plan
    ( (with-auxiliary-goals ?bdgs                                 ; op-2
        ?prepare ?perform (clean-up ?out-clean-up))
      (with-auxiliary-goals                                       ; op-1
        (perform
          (with-constraints
            (roll:acquire-experiences
              (getgv :experience 'observe-containers-opened-exp))
            (:tag OBSERVE-CONTAINERS-OPENED
              ?plan)))
        (clean-up
          (for-all
            (lambda (container) (:achieve (container-closed container))
            (get-observed-containers-opened ?tag))))) ))
```

used inside the condition. In combination with `set-of` all occurrences can be collected (lines 9–19).

The first input schema (`is-1`, lines 8–19) binds the whole plan (path is empty) to the variable `?plan` and searches for all subplans with the command `with-auxiliary-goals` containing a call to the goal `container-closed` in the clean-up part. The interesting variables of the search are collected in a set and bound to the variable `?aux-goal-vars` (line 19).

This variable contains now several subplans and each should be transformed. The control command `:for-each` ⟨*variable*⟩ `:unify` ⟨*expr*⟩ provides the ability to loop over each element and unify its content, which then can be used by `match-plan` (`is-2`, lines 20–22). For each iteration the appropriate transformation predicate (`tr-2`, lines 24–27) is called. In this case `(:achieve (container-closed ?_))` is replaced with `(no-op)`. At first, the new output plan is a copy of the input plan and then the transformed subparts are replaced with the code specified. In the example output plan two (`op-2`, lines 30–31) replaces the command `with-auxiliary-goals` with the new version. Output plan one (`op-1`, lines 32–42) wraps `with-auxiliary-goals` around the original, but already in subparts modified, plan. In the `perform` part a monitoring component observing opened containers is added and in the `clean-up` part steps to close these containers in the end are added.

So far the transformation rule always generates exactly one new output plan. But when several subplans match, it is not always clear if every subplan should be transformed. In a later example we will see a transformation rule searching for subplans using the construct `for-all` and expanding it to a sequence. Should every occurrence of `for-all` be expanded now or only occurrence two and six, for instance, or any other combination? The control command ⟨*branch*⟩ allows to specify a branching, where each branch generates a separate output plan in the end. The example in Listing 5.2 generates the power set of all `for-all` goals found, unifies each element of the power set with the variable `?for-all-goals-set` and the branch is only accepted if the length of the subset is at least one. A transformation rule containing this branch would generate $2^n - 1$ output plans, where n is the length of the variable `?for-all-goals`.

Each input schema can have either one ⟨*branch*⟩ command or one ⟨*for-each*⟩ command. The difference is that ⟨*branch*⟩ generates new output plans, whereas ⟨*for-each*⟩ performs the transformations on the same output plan.

Another example of an input schema is presented in Section 5.4.3 on page 94.

Listing 5.2 Example of using `:branch` inside a input schema definition.

```
:branch (:generate (power-set ?for-all-goals)
         :unify ?for-all-goals-set
         :cond (> (length ?for-all-goals-set) 0))
```

Summary of Contributions

We have mentioned *XFRML* as the predecessor of our plan transformation rules. However, the syntax for our system allows the specification of more sophisticated transformation rules in several ways.

First, the structure of the rules clearly separates the syntactical plan matching and semantical conditions of the applicability from the transformation part. The separation of applicability and transformation commands doesn't only add clarity to the transformation rule, but makes nested and branching transformation rules possible. In contrast, *XFRML* doesn't distinguish between applicability conditions and transformation. Instead, it relies on specifying complicated Prolog rules in what is called the condition part of the transformation rule.

The second main difference to *XFRML* lies in the specification of the input schema. With the sophisticated pattern matching syntax, our plan transformation rules can handle cases of matching a plan in several ways and allow to specify the desired outcome by either applying the rule to one or several matches or producing more than one output plan. Also our representation allows incomplete paths or unknown path positions.

Finally, the transformation rules of *XFRML* were designed for an abstractly simulated world, where a robot is to move objects from fields in a grid to other fields. The plans necessary for our kitchen robot are more complex. Only the additional power of the TRANER transformation rules allows sophisticated transformation procedures and the specification of very general rules that can be applied to a wide variety of concurrent reactive plans.

5.3 Transformation Procedure

As mentioned in the previous section every definition part (applicability, input-schema, transformation, and output-plan) of a transformation rule is a list of the same length. If the length of each list is exactly one, then the basic transformation procedure is straightforward and in accordance with the structure of the rules. In the first step the plan is matched to the input schema and the rule variables are bound to the respective plan parts. Before starting the transformation, the applicability condition is checked, optionally binding new variables. Only when the rule is applicable for the valid input match the transformation is performed using the previously defined variables. The output is then composed by accessing all the variables, particularly those set in the transformation step.

A moot point in this procedure is whether to apply the applicability condition or the input matching first. Both are needed to decide if the rule can be used for a given plan. From a declarative point of view this question should be unimportant, because the rule is applied if both the syntactic input matching and the behavioral applicability condition hold. From the implementational point of view the order of execution can affect the effi-

ciency of the whole transformation. In our current implementation the syntactical check comes first and the applicability is tested afterwards.

This simple procedure gets more complicated when nested plans and transformations on the subplans are involved. One intricacy is the treatment of subplans with respect to the overall plan: Particularly the order of how each element of the definition lists is processed is a little bit tricky. The left hand side of Figure 5.8 shows an abstract transformation rule having n definitions in each list. Two input schemas have a control command included. A `:branch` command is specified in $\texttt{is}_2$, and in $\texttt{is}_{n-1}$ a `:for-each` command is declared. The right side of the figure shows the order of how each element is called and how branching takes place.

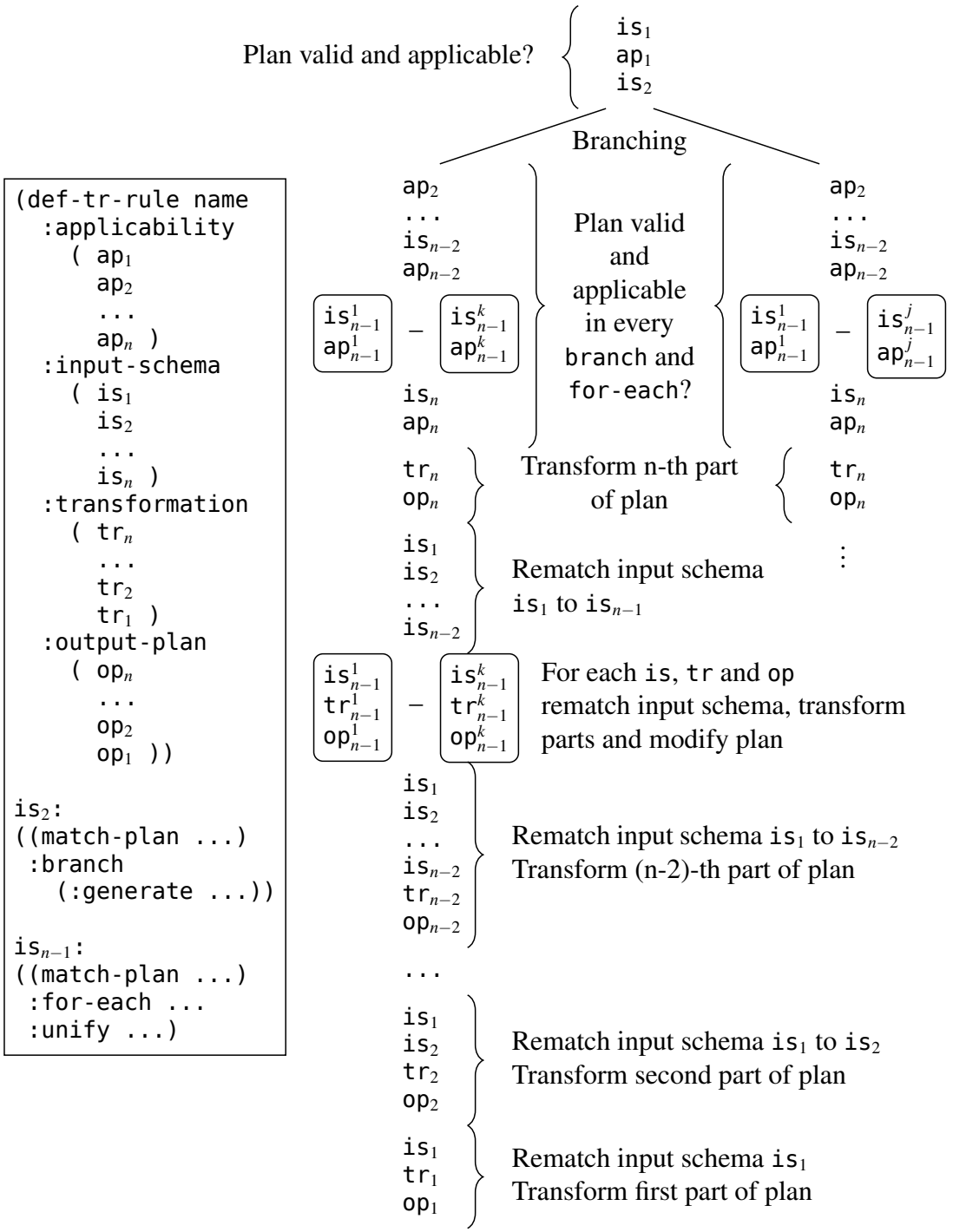
For the input schema and the applicability, the matching works alternating from the top-level plan towards subplans. This is necessary, because subplan matches can often be defined in terms of parent plans. For the composition of the transformation result, however, the subplans must be assembled first, so that their transformation results are included into the final plan and are not discarded by using their original versions. This reverse order is reflected in the definition of transformation rules. Input schema and applicability define their elements from 1 to n, whereas transformation and output plan elements go from n to 1.

If an input schema defines a `:for-each` command, some special treatment is necessary. Usually following input schema and applicability definitions can access the variables bound in previous definitions. But since a looping is performed over the content of the variable specified, it is not clear from which iteration a binding should be kept. For this reason all new variables in every iteration are local and can't be accessed by other iterations or following definitions.

Another subtle point is the transformation of one plan to several output plans by applying one rule. If a branch command is declared new branches are generated, which are completely independent. The following matching and transformation steps are executed for each branch. If a later input schema defines another branch, then in every branch new branches are defined which leads to many new output plans at the leaf nodes of the transformation tree. Following `:for-each` commands can have different lengths, since the length depends on previously defined variables. In Figure 5.8 the left branch length of the `:for-each` command is k, whereas the right branch has length j.

The last difficult aspect to note is when the transformation starts. First the last input match is transformed by $\texttt{tr}_n$ and the part of the plan specified by the path in the input schemas is replaced with the resulting output plan $\texttt{op}_n$. Since some variables may be bound incorrectly due to the modification of the plan a rematch of the previous input schemas ($\texttt{is}_1 \ldots \texttt{is}_{n-1}$) is necessary. During a rematch every `branch` command (branches are already defined) and `for-each` command (binds only local variables) is ignored. After the rematch the next transformation and plan modification is performed. In the case of a `for-each` command, in every iteration the rematching, the transformation and the

Figure 5.8 The left side shows the definition of a transformation rule, having n elements in each list. A `:branch` is defined in $\texttt{is}_2$ and a `:for-each` in $\texttt{is}_{n-1}$. The right side shows the processing order of each element of the definition lists.



replacement are performed as a group. Sometimes it is possible that transformations of subplans modify the plan in a way that during rematching previously successful conditions return false. To detect if a rematch is performed inside a condition, there is a predicate `rematch-p`[5] returning false for the first match and returning true for every rematch. With this predicate it is possible to write the condition appropriately.

5.4 Transformation Rule Library

Similar to plans, the transformation rules are organized in a library. In the following we describe several transformation rules, which are categorized in six general classes. Some instance rules of these classes are introduced on the basis of a running example, the plan for setting the table.

Listing 5.3 The default plan for setting the table. An example for calling the plan is `(achieve (table-set '(:alvin :simon :theodore)))`.

```
(define-plan (achieve (table-set ?persons))
  (for-all
    (lambda (person)
      (with-designators
          ( (table '(some entity (type table)
                                 (used-for meals)))
            (seating-location `(some location (at ,table)
                                              (preferred-by ,person)))
            (plate `(some entity (type plate)
                                 (status unused)
                                 (status clean)
                                 (preferred-by ,person)))
            (cup `(some entity (type cup)
                               (status unused)
                               (status clean)
                               (preferred-by ,person))) )
        (achieve (entity-on-entity plate table)
          :location `(some location (on ,table)
                                    (matches (entity-location ,plate))
                                    (matches ,seating-location)))
        (achieve (entity-on-entity cup table)
          :location `(some location (on ,table)
                                    (matches (entity-location ,cup))
                                    (matches ,seating-location)))))
    ?persons))
```

[5] Listing 5.4 on page 96 shows an example usage of the predicate `rematch-p` and the listing is explained on page 95.

5.4.1 Example Default Plan

The default plan for setting the table is shown in Listing 5.3. Lines 4–16 define the important designators of the plan. First the `table` is described as an entity of the type table which is used for meals. Second the designator `seating-location` describing the location where a person prefers to sit at the table is defined. The last two designators specify the `plate` and the `cup` to be placed on the table.

Lines 17–24 specify the steps the robot has to take to reach the goal. By calling the goal `entity-on-entity` the plate and the cup are placed on the table. The position on the table is given by additional designators describing the location on the table where to put the object depending on the type of the object (`entity-location`) and the `seating-location` of the person.

Section 4.2.3 described that `entity-on-entity` calls `entity-placed-at-location`, which itself first calls `entity-picked-up` and then `entity-put-down`. The pick-up plan searches for the object described by the designator, navigates to a location where it can grip it and finally picks it up. Putting down the object includes navigating to a suitable put down position and placing it on the table. These plans are implemented in a robust way and can recover from local failures, like losing the object during navigation.

Normally more than one person attends a meal. By using the construct `for-all` (line 2) the steps of defining the appropriate designators for a person and placing the needed objects on the table are repeated for every person.

Most of the transformation rules presented in the following operate on this plan. For an easier illustration of the modifications a transformation rule performs on the plan, the simplified representation on the right is used. $P_{default}$ gives the name of the plan, `w-desigs` means `with-designators`, and `eoe` $e_1 \rightarrow e_2$ is the short version of calling the goal `entity-on-entity` for placing entity e_1 on entity e_2. Instances of e_1 and e_2 are `p`, `c`, `t` and `s`, meaning plate, cup, table and a stack.

$P_{default}$

```
for-all
  w-desigs
    eoe p→t
    eoe c→t
```

5.4.2 Transformation Rule Classes

The plan for setting the table shows potential for improvement, since bringing the necessary objects to the table one by one for every person takes a lot of time. By applying a small number of transformation rules, the behavior can be enhanced significantly. In the following we introduce the set of transformation rules we have defined for the kitchen robot and group them with regard to their effects.

Although our running example is setting the table, the transformation rules presented here are not designed specifically for this tasks. They apply equally to the preparing pasta scenario and can be expected to be useful in optimizing other household tasks.

First of all, we give an overview of the classes of transformation rules we have identified and introduce some instances of these classes. Figure 5.9 on page 93 depicts how the application of the transformation rules affects the performance of the table setting plan and shows the search tree for the best plan. This example will be explained in more detail in the next sections, but we already provide references to the identifiers of the rules used in the figure.

Based on their effects, we group our transformation rules into different classes. In this section we explain the general effect by which a class is identified and give examples of transformation rules belonging to it. The groups we are using are the following:

- syntactical transformations,
- reordering of plan steps,
- usage of external resources,
- usage of robot resources,
- modification or introduction of auxiliary goals,
- optimization of free time.

Syntactical Transformations are ones that don't affect the robot's behavior, but only change the syntax of the plan. Here we differentiate between two subclasses of rules: code beautifiers and code restructuring rules. Examples of code beautifiers include the replacement of `seq` commands by implicit sequencing, for example inside a `let` construct. Similarly, a sequence of several `seq` commands can be combined to one `seq` containing all subinstructions. Removing `no-op` instructions is another instance of this class. Code beautifiers are applied on already transformed plans. Plan transformations sometimes generate code that works, but lacks compactness and readability. For example, in the local context of a plan transformation rule it might make sense to add an extra sequencing command, but in the context of the whole plan, this command can be superfluous and should be removed to make the plan better readable and therefore more easily transformable by other rules. As the transformation rules rely on the syntactic structure and assume a sensible coding style, code beautifiers play an important role to make chains of plan transformations possible.

The second class of syntactical transformations — code restructuring rules — alter the syntax of a plan without changing its behavior in order to enable subsequent plan transformations. One such rule (named T_7 in Figure 5.9) converts a `for-all` instruction to a sequence of instructions. This gives subsequent transformations more freedom in deciding on the order in which all the steps are executed. On the other hand, the plan gets more specific with respect to the number of instances for which the plan is executed. Whereas `for-all` can be parameterized for which objects the subplan is to be achieved, the sequence is expanded into a fixed number of steps. This is like replacing a loop command in a normal program by a sequence of instructions. Another code restructuring rule (T_1)

moves designator bindings outside the main plan code. This means that instead of binding designators inside a `for-all` instruction, for instance, the bindings are already defined outside of `for-all`. This is important for keeping designator bindings intact when the plan is transformed. A third example of code restructuring is to replace a `seq` or `par` command by a `partial-order` instruction, in the first case by representing the complete sequencing order in the partial order constraints, in the second by omitting all partial ordering constraints. Although this doesn't change the behavior of the plan, the `partial-order` command provides more flexibility in removing or adding constraints on the execution order. This means that subsequent transformations can transform a fixed sequence to one where some of the steps can be executed in parallel.

Reordering of Plan Steps does affect the behavior of a plan, but is usually another introductory step for other transformations. In many cases, the order of plan steps doesn't change the overall result or performance of the plan. For instance, the order of steps is irrelevant when bringing a cup and a plate to the table. Therefore, in many cases, the reordering of plan steps is very similar to a syntactical transformation, the visible robot behavior being only modified slightly and the final state being equal.

The rule named T_2 in Figure 5.9 converts a `for-all` command containing n substeps to n `for-all` commands containing one substep each. For example, when the plan contains the instruction to achieve the steps (1) bringing a plate to the table, and (2) bringing a cup to the table, both to be performed for four people, the plan can be converted to the two instructions (1) to bring a plate to the table for each of the four people and (2) to bring a cup to the table for each person. The overall result of the plan is not changed by this transformation. However, it enables subsequent stacking of the plates and usage of both arms or the tray to bring the cups to the table. When performing this transformation, it is important to take care of the designator bindings. If the bindings were declared inside the original `for-all`, the resulting plan might be flawed, because not all designators are bound correctly in the new `for-all` instructions. Therefore, the syntactical transformation moving the designator bindings outside (T_1) should have been applied before this rule.

Another instance of reordering plan steps is simply to swap the execution of two steps. This is an instance of the more general rule to change, add, or delete constraints of a `partial-order` instruction. As we said before, a sequential or parallel execution can be transformed to a partially ordered one. The combination of the two rules is a powerful tool to modify the execution order.

In some cases, it is reasonable to order plan steps by a certain execution criterion, for example the order in which the robot acts in different locations. By grouping the steps where the robot works at one location, navigation steps can be removed or the execution can be enhanced by using external or robot resources, for example by using a tray.

Using External Resources like containers can significantly enhance a robot's performing when executing a plan, mainly by reducing the number of navigation tasks. One example is the stacking of objects. In this case, the robot uses the bottom object of the stack as a tool to move the other objects. The stacking rule comes in two variants, depending on the syntactical structure of the plan (Rule T_3 and T_4). If the moving commands for the objects are combined in a `seq` instruction, rule T_3 can be used, whereas T_4 takes care of commands inside a `for-all` command. When using transformations, all plans could be transformed by resolving the `for-all` to a `seq` instruction and then using rule T_3. However, as mentioned before, this restricts the plan application to a certain number of objects or people. Leaving the `for-all` in place results in a more general plan.

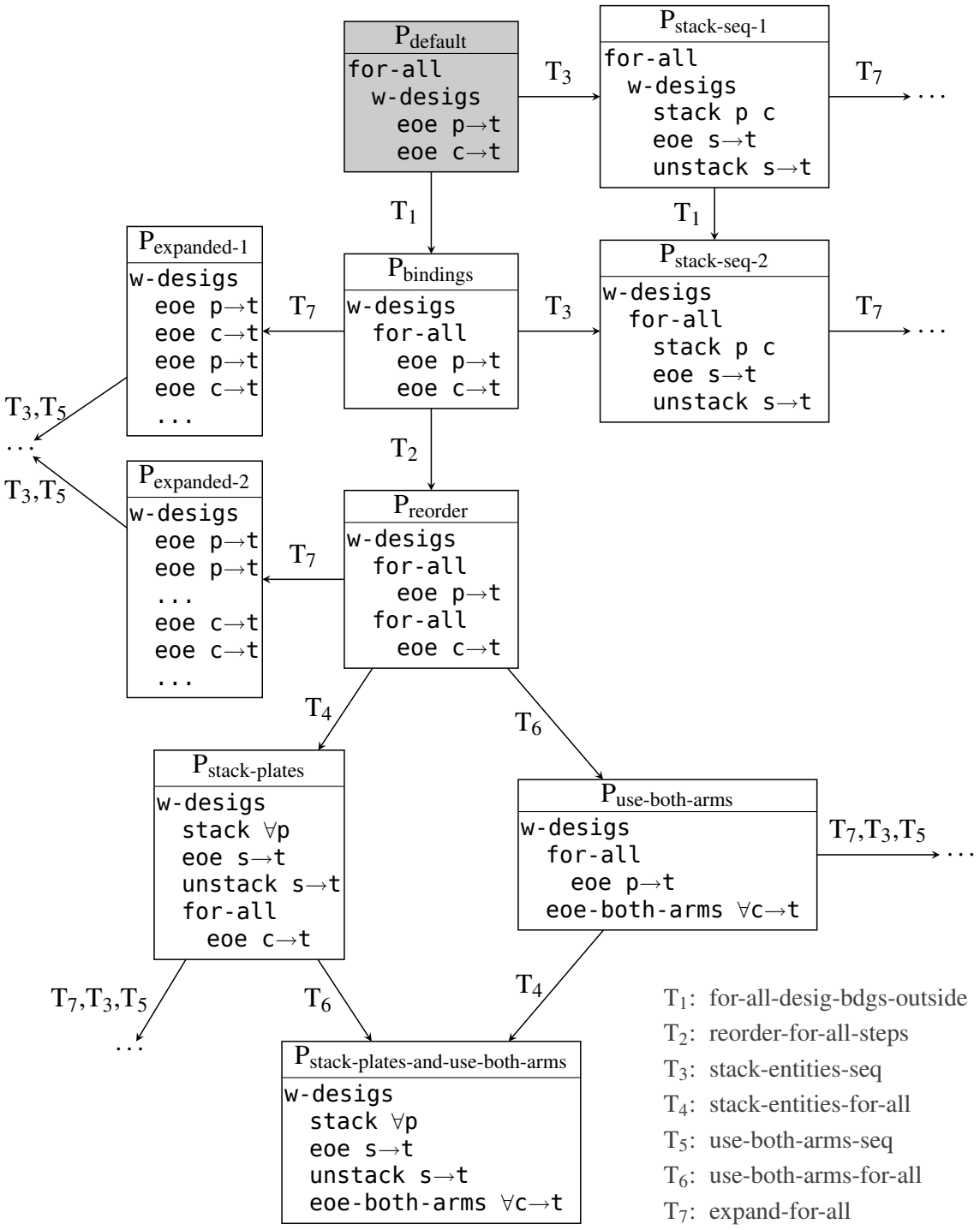
The stacking of objects can be regarded as an instance of the more general rule of using containers. The container can be a tray, a bucket, a pot, or a plate. When the objects inside the container are of the same kind as the container itself — like plates on a plate — the objects are stacked. An even more general formulation than using containers is using tools to fulfill a given task. Instead of opening a package of milk with its grippers and running the risk of spilling the milk, the robot should rather use a pair of scissors.

Using Robot Resources is another way of optimizing the behavior of a plan. Robot resources include its arms and the integrated tray on top of the robot. Similar to the two rules for stacking objects depending on whether the plan steps are grouped sequentially or by a `for-all` command, there are two rules for using both arms when carrying objects. Rule T_5 works on a sequence of carrying instructions, whereas rule T_6 modifies the moving of all objects in a `for-all`. Another option is to use the robot's built-in tray in the same way as an external container to move several objects at a time.

Modification or Introduction of Auxiliary Goals is very important for making the overall robot behavior successful. As we explained in Section 4.3 on page 66, auxiliary goals are indispensable for showing reasonable long-term behavior. However, it is extremely difficult to include auxiliary goals at the appropriate places in a plan. Plan transformations are a good way to discard unnecessary auxiliary goals by summarizing some of them or to introduce auxiliary goals at appropriate places in the plan.

One instance of removing superfluous auxiliary plan steps is the closing of cupboards as already explained in Sections 5.2.1 and 5.2.2 (page 79ff). As a default, the robot closes a cupboard as soon as it has taken an object out of it. In the overall context of the plan, this can end in the robot opening and closing a cupboard several times subsequently. The transformation rule first removes the closing actions and then makes the robot notice which cupboards it opens during the plan execution. At the end of the main task, all open cupboards are closed. A similar transformation is necessary to handle the retraction of the slidable boards inside the cupboards.

Figure 5.9 Transformation graph of optimizing the default plan for setting the table.



Depending on the context of the whole plan, auxiliary plan steps might be unnecessary altogether. Usually when the robot stirs the content in a pot, it lays down the spoon afterwards. When several stirring actions are necessary and the robot doesn't need its hand resource for other actions, the spoon should be kept in the gripper. On the other hand, sometimes auxiliary goals are not taken care of by the subplans, but should be added when these plans are executed in a wider context. Again, when the robot has finished stirring, it might be appropriate to clean up the spoon or to put it at some place where it is needed afterwards.

Optimization of Idle or Waiting Time works on a high level of abstraction. It optimizes high-level activities so that they can be executed in parallel like setting the table while preparing pasta. These transformations need a lot of knowledge about the execution of the plans involved, especially the duration of plan steps. The steps can then be intertwined by reordering transformations.

The transformations presented here are general in that they can be applied to a variety of kitchen tasks. In the following sections, instances of all classes of transformation rules are explained in more detail in the context of our running example of setting the table.

5.4.3 Using Containers

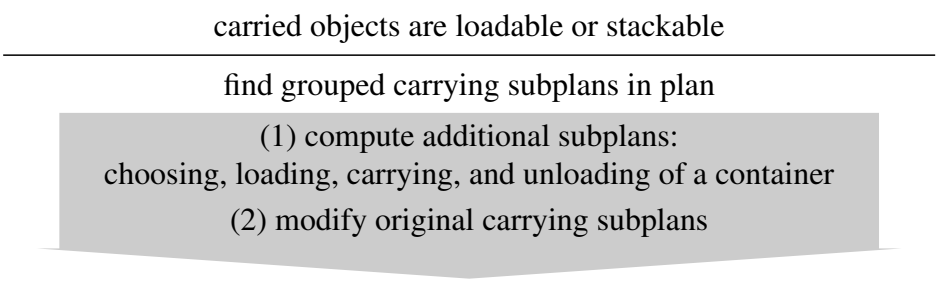
Often the robot has to carry objects during its daily operation. Using containers is usually useful when many objects must be transported to similar locations, carrying steps are performed consecutively in plans, or when objects can easily be lost like in the case of cutlery. The type of container to be used can be very diverse. For example, the robot can use a tray, a pot or even one of the objects it has to carry. The transformation rules have to add new plan steps for loading the container, thereby stacking the individual objects if possible, and unloading it. In this section we concentrate on the transformation rule class for using containers, which is a subclass of using external resources. In particular, the case where the container is the bottom object of a stack is discussed.

Figure 5.10 shows a general description of how instances of the transformation rule class for using containers are implemented. The input matching has to find subplans for carrying groups of objects and the applicability checks if the carried objects can be stacked or loaded on a container. The transformation part computes additional steps for choosing an appropriate container, loading, carrying, and unloading the container. The new output plan contains the additional and adapted carrying subplans.

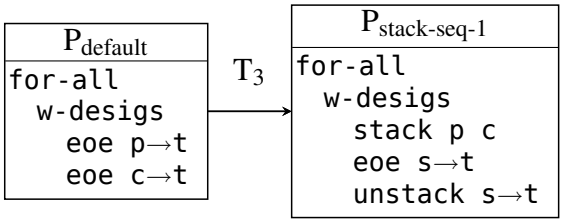
The next two paragraphs present two instances of this general description (Rules T_3 and T_4) where the new plans stack objects before carrying and unstacking them. One question here is where to place the stack before it is unloaded, a good choice being the final location of the bottom object. After the transformation the robot needs to navigate

less than in the original plan, or in the best case not at all. For stacking and unstacking the higher-level goals `entities-stacked` and `entities-unstacked` are used as described in Section 4.2.3.

Figure 5.10 General description of how instances of the transformation rule class for using containers are implemented.



T_3 stack-entities-seq. The default plan[6] for setting the table places a plate and a cup on the table for each person. A possible transformation is to stack the cup on the plate, place the stack (i.e. the plate at the bottom) on the table (Figure 5.11(a)) and unstack it by placing the cup on the table. The most interesting part of this transformation rule is the input schema shown in Listing 5.4. The transformation and output plan parts are straightforward.



The input schema consists of three `match-plan` commands (lines 1, 31 and 34), the first is the most complicated, particularly the condition part. The `rematch-p` predicate (line 4, see page 88) is needed because when the plan is rematched after transforming subplans the transformation rule would fail to find a sequence of `entity-on-entity` goals. This sequence is matched by the inner `match-plan` called by `set-of` (lines 6–17). Here the specification of the path with `:at` (line 9) is crucial. The outer `match-plan` (line 1) already performs a search over all paths of the plan and binds the current path in every search node to the variable `?path`. The inner `match-plan` performs another search, but this time the search is not over the whole plan. Instead, only all direct subplans of the path bound in `?path` are checked whether they fulfill the unification condition

[6]From now on, every time a transformation rule is explained, an excerpt depicts the part of the transformation graph of Figure 5.9 described in the paragraph. The left part shows the input plan and the right part the transformed output plan.

`(achieve (entity-on-entity ...))`. The path of each direct subplan is bound to the variable `?eoe-path`. The unified variables, `?eoe-path` and `?eoe-plan` are collected as tuples in the variable `?eoe-all-tuples`. Finally the condition part of the first `match-plan` (lines 18–20) extracts all found subpaths using the predicate `unify-tuples->lists`[7] and

Listing 5.4 Input schema of transformation rule `stack-entities-seq`.

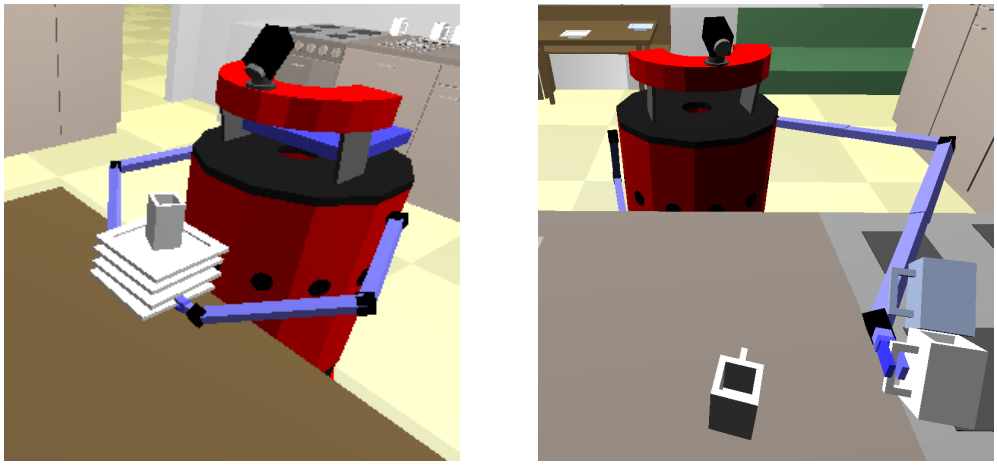
```
1  ( ((match-plan
2      :at ?path
3      :plan ?plan
4      :cond (or (rematch-p)
5                (and
6                  (set-of (?eoe-path ?eoe-plan ?top-entity
7                           ?bottom-entity ?location)
8                          (match-plan
9                            :at ( !?path ?eoe-path )
10                           :plan ?eoe-plan
11                           :cond (unify (achieve
12                                          (entity-on-entity
13                                            ?top-entity
14                                            ?bottom-entity)
15                                          !?location)
16                                        ?eoe-plan))
17                         ?eoe-all-tuples)
18                 (unify-tuples->lists
19                   ?eoe-all-tuples (?eoe-all-paths !?_))
20                 (is-sequence ?eoe-all-paths))))
21    :branch (:generate (power-set ?eoe-all-tuples)
22             :unify ?eoe-tuples
23             :cond (and (lisp-pred > (length ?eoe-tuples) 1)
24                        (unify-tuples->lists
25                          ?eoe-tuples
26                          (?eoe-paths ?eoe-plans ?top-entities
27                           ?bottom-entities ?locations))
28                        (unify
29                          (?eoe-paths-first !?eoe-paths-rest)
30                          ?eoe-paths))))
31   ((match-plan
32      :at ( !?path ?eoe-paths-first )
33      :plan ?eoe-first-plan))
34   ((match-plan :at ( !?path ?eoe-path-no-op ))
35    :for-each ?eoe-paths-rest :unify ?eoe-path-no-op) )
```

[7] The predicate `unify-tuples->lists` has two arguments. If the first one has the structure ((a_1 b_1 ... x_1) (a_2 b_2 ... x_2) ... (a_n b_n ... x_n)) then the second argument has to be of the form ((a_1 a_2 ... a_n) (b_1 b_2 ... b_n) ... (x_1 x_2 ... x_n)).

checks if the paths are a sequence.

The `:branch` command (lines 21–30) defines how many new output plans should be generated. The `power-set` function returns all possibilities of combining the elements of the sequence. The condition part of the `:branch` command checks if the sequence has at least a length of two and binds variables needed for the last two `match-plans`. The second `match-plan` (lines 31–33) chooses the first element of the generated set of paths and during transformation the new steps for stacking and unstacking are generated using the variables bound in the `:branch` command. The last `match-plan` (lines 34–35) iterates over the rest of the found paths and the transformation part will replace these subplans with `(no-op)`, which code beautifier transformations later remove completely.

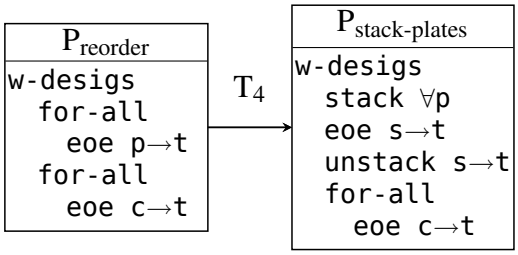
Figure 5.11 Examples of plan executions after applying stacking transformation rules.



(a) Plates and one cup stacked.

(b) Stacking of cups fails.

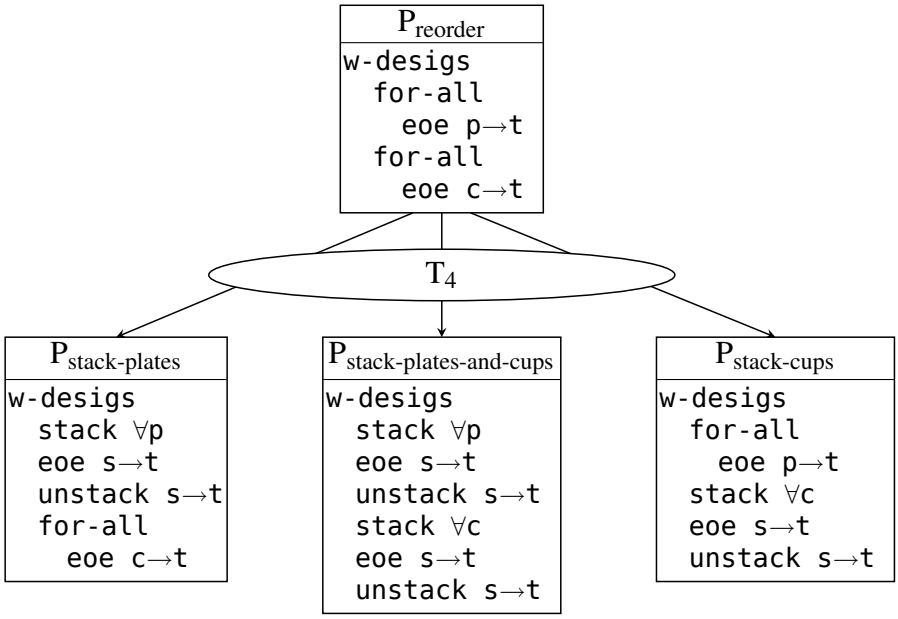
T_4 stack-entities-for-all. The previous transformation rule searched for the consecutive occurrence of the goal `entity-on-entity` as a sequence in a plan. If the goal is the single step of a `for-all` then the `for-all` can be resolved by first stacking all top entities, placing the stack at a suitable location and unstacking it. The necessary input plan $P_{reorder}$ for the transformation is constructed from the default plan by applying the two transformation rules for moving designator bindings to a higher level and reordering plan steps as explained in the next section.



The input schema searches for all occurrences where a `for-all` contains a single `entity-on-entity` step. The plan $P_{reorder}$ contains two matches. Since it is not clear if both matches or only one should be transformed, a `:branch` command is defined in the input schema generating a power set of the possible matches (see Listing 5.2). For $P_{reorder}$ there are three[8] possible output plans (see Figure 5.12), but only one plan is successful when the plans are tested. The successful plan only stacks plates, the other two try to stack the cups which fails in simulation (Figure 5.11(b)) and so the plans are discarded.

Actually the input schema shown in Listing 5.4 for transformation rule T_3 is more complicated. If a plan contains more than one sequence of `entity-on-entity` goals, then the same branching as in this transformation rule must also be performed in T_3. This means that two nested `set-of`s are necessary and the number of generated output plans is much higher.

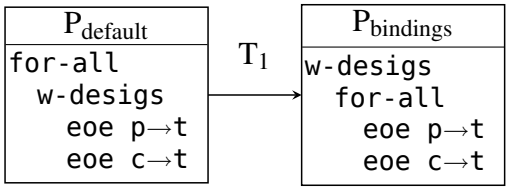
Figure 5.12 Applying transformation rule T_4 on plan $P_{reorder}$ results in three output plans. The two plans on the right fail, when tested in simulation



5.4.4 Syntactical Transformations and Plan Reordering

In the previous section transformation rule T_4 relied on input plan $P_{reorder}$. To get to this plan from $P_{default}$ the two transformation rules T_1 and T_2 must be applied.

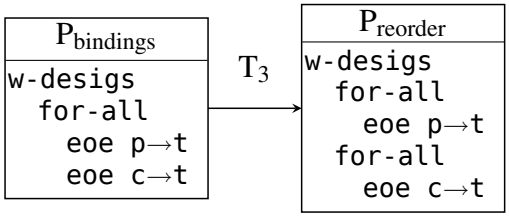
[8] If n is the number of matches, then $2^n - 1$ output plans are generated.



T_1 for-all-desig-bdgs-outside. T_1 belongs to the class of code restructuring rules. A precondition for the reordering performed by rule T_2 is that the `for-all` steps don't depend on local variables defined inside the `for-all` command. To be more precise, local variables are only a problem if they themselves depend on the arguments of the `for-all` lambda definition. In the running example shown in Listing 5.3 on page 88 the only lambda argument is `person` (line 3) and the designators `seating-location`, `plate` and `cup` depend on this argument. The effect of T_1 will be to swap the order of the designator definitions and the `for-all` command.

The problem is how to move designator binding definitions outside of the `for-all` command, although the definitions depend on variables only available inside the `for-all` command. To this end we extended the designators to support the specification of unbound variables. We call designators with unbound variables *partial designators*. The unbounded variables are constrained later when the local definition is known. Partial designators ensure that they are unique when constrained to the same values. Unbound variables are indicated by a dollar sign as a prefix.

The transformation part of T_1 first searches for designators depending on arguments of the `for-all` lambda definition. The lambda arguments are replaced with unbound variables inside the designator specification (e.g. `person` becomes `$person`). Inside the `for-all` command all references to now partial designators are replaced with a call to constrain partial designators with the values of the lambda variables.



T_2 reorder-for-all-steps. After having moved the designator bindings outside of the `for-all` command this transformation rule is very simple. T_2 reorders the substeps of a `for-all` command by regrouping them. There are many possibilities of how to reorder the substeps. However, T_2 doesn't change the order of the substeps, but a reordering is achieved by grouping the substeps and adding a `for-all` step around each group. For example if there are five substeps then possible groupings are [(1) (2) (3) (4) (5)], [(1 2 3) (4) (5)], or [(1 2) (3) (4 5)], but not allowed is [(2 5) (1 3 4)]. If there are n substeps, the number of possible groups[9], is $2^{n-1}-1$. In our running example n is 2, so the only possible output plan is the one shown. Now first all plates and then all cups are placed on the table. When the default plan is extended to place cutlery (knifes, forks, and spoons) on the table, this rule plays

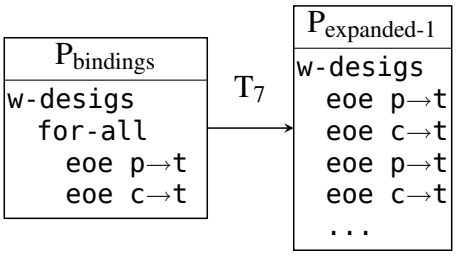
[9] If n is the number of steps, there are $n-1$ possibilities where a separator can be inserted. The power set of these $n-1$ separators except the empty set ($2^{n-1}-1$) gives all possibilities how the steps can be partitioned.

an important role for finding general plans where, for example, plates are stacked and the cups and the cutlery are transported using both arms.

The input schema of T_2 searches for every occurrence of a `for-all` command containing only a sequence of `achieve` steps. Again the power set function is used to generate branches for every possibility of which combination should be transformed. The power set function is combined with the function for generating all possible groups. In the end we get $(2^n - 1) * (2^{n-1} - 1)$ output plans. The transformation and the output plan part are straightforward.

Another solution to get $P_{reorder}$ from $P_{default}$ could have been to skip T_1 completely and to copy the designator definitions for every newly created `for-all` command. This has some disadvantages, like that designators are created more than once, that equally constrained designators are not unique any more, and that it is more complicated to recognize the designators used in the plan and to transform the plan further.

T_7 expand-for-all. So far the transformed plans reordered the plan steps, stacked one cup on one plate or stacked all plates. Another possibility to stack the objects is to stack all plates and one cup on top of it (Figure 5.11(a)). To achieve this the `for-all` command has to be expanded by unrolling the loop to a sequence. The length of the sequence depends on the minimum value of lengths of the `for-all` input arguments. This means, for example, that the resulting plan for three people attending a meal looks different from one for four people, although the original plan is the same for both cases, containing the number of people in a variable. The resulting plans are now very specific and work only with the correct argument length, whereas the other presented transformations cope with every length. This may seem to be a disadvantage, but only after applying this transformation further adapted plans can be generated and evaluated.

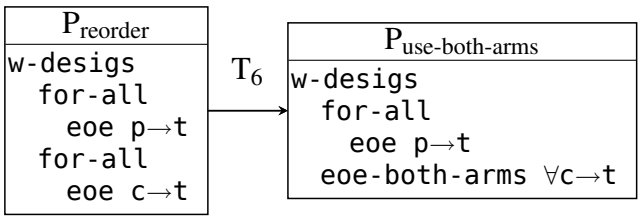


As shown in Figure 5.9 on page 93 T_7 can be applied to many of the already transformed plans. The resulting plans are then transformed further, particularly with rule T_3 for stacking objects and rule T_5 for using both of the robot's arms as presented in the next section. With this combination of transformation rules we got two interesting plans setting the table for four persons. The first plan places one cup on top of four stacked plates and carries the remaining three cups by first placing two cups on the table and then the last one. The second plan only stacks two plates and one cup and places the objects on the table. The remaining two plates and one cup are stacked again and placed on the table. Finally the last two cups are carried to the table. Which of these two plans performs better is not clear a priori and can only be evaluated by executing the plans in simulation (see Chapter 7 for results).

5.4.5 Using Robot Resources

For the plates a more efficient way of bringing them to the table is to stack them, but for the cups this fails. One possibility to be more efficient could be to use a tray, either as an external resource or if available as a robot's resource if a tray is mounted on top of the robot (Figure 5.13(a) on the following page). Another alternative for cups is to carry two cups at the same time, since for carrying cups only one arm is required (Figure 5.13(b)). The behavior of the last alternative is produced by the transformation rules T_5 and T_6. Like T_3 and T_4 for stacking objects they differ in the way the input plan has to look like.

T_6 use-both-arms-for-all. Like T_4 this transformation rule searches for the goal `entity-on-entity` occurring as a single step of a `for-all` and replaces the `for-all` with steps carrying two objects using both arms. In the figure on the right and on page 93 `eoe-both-arms` represents these steps. If the table is to be set for an uneven number of persons, then in the last step only one cup is picked up and placed on the table.



Again, if more than one `for-all` is found in the plan a branch using the power set function is generated. The figure on the right shows the only valid plan, since two plates can only be carried in a stack and not by holding one plate in each hand. The robot needs both hands to grip and carry one plate.

T_5 use-both-arms-seq. Again this transformation rule searches for the goal `entity-on-entity` occurring consecutively as a sequence in the plan. The input schema and the branching is the same as shown in Listing 5.4 and described in Section 5.4.3. Only the transformation and the output plan parts differ, but here like in T_6 the original plan steps are replaced with steps for carrying two objects at a time using both arms.

We have now explained all the transformation rules of the example in Figure 5.9. Note that $P_{\text{stack-plates-and-use-both-arms}}$ can be obtained from P_{reorder} by either applying rule T_4 and then T_6 or by applying first T_6 and then T_4. Similarly, the plan $P_{\text{stack-seq-2}}$ results from the default plan by applying any order of the rules T_1 and T_3. The following rules can enhance the example plan further, but are ignored in the figure for the sake of a clearer presentation.

Using the Built-in Tray. When the robot is equipped with a tray on top of its base interesting new alternatives arise for setting the table. For example when setting the table for three persons, two cups can be placed on the tray and one cup on the stack of plates,

then the robot must only navigate to the table once. The transformation rule for using the tray searches for the occurrence of an `entity-on-entity` goal in a sequence of steps, generates plan alternatives where placing the object on the tray is performed at all possible previous steps in the sequence. The original call to the `entity-on-entity` goal is kept, because when the goal is executed the object is taken from the tray and placed at the desired location.

Figure 5.13(a) shows an instance of where the robot uses its built-in tray. This rule yields especially impressive performance gains when used with cutlery, because the danger of losing such objects is reduced.

Figure 5.13 Robot using its own resources more effectively.

(a) Robot using its built-in tray.

(b) Robot using both arms to carry objects.

5.4.6 Further Transformation Rules

We have described in detail transformation rules operating on our running example. The following transformation rules present some more rules operating on the plans from the plan library.

Modification or Introduction of Auxiliary Goals

The issue of auxiliary goals has been discussed in several places of this work (Sections 2.2.2, 3.4.2, 4.3). One instance of this problem is the opening and closing of doors (see Figure 5.6 on page 79, Listing 5.1 on page 83 and their explanations). Our default plans use the simple heuristic of closing cupboard doors every time an object has been taken out or has been placed in the cupboard. This leads to suboptimal behavior. Our

transformation rule transforms the plan so that the cupboard doors are opened as in the original plan, but the closing is postponed as the last step of the plan.

This rule is very general. It can be applied to every intermediate plan resulting from a transformation in the running example of Figure 5.9. A similar rule takes care that the boards inside the cupboards are retracted only in the end.

Optimize Locations

In Section 3.4.2 we have introduced the construct `at-location`, which takes care that the robot stays at a certain location while performing another task. This construct allows to identify subplans where the robot performs actions at the same place. By grouping these subplans, the robot navigates to the location only once and then stays there to perform all the tasks of the subplans. This is another rule of the class "Reordering of Plan Steps".

Optimizing Free Time

The last rule to be presented here transforms two plans so that they can be executed concurrently by using free time in one plan to perform steps of the other. As input plan this rule accepts any combination with `seq`, `par` or `partial-order` of two activity or higher-level manipulation plans. The output is a partial ordering of the plan steps contained in these two plans.

This rule differs slightly from the other rules in our library. First, because it is less general in that it works only on activities or higher-level manipulation plans. Second, it is more complex as it uses a two-step approach for transformation.

When the combination of plans is to be transformed, a first transformation modifies the combined plan so that special execution traces are monitored during plan execution. This traces contain information about the duration of all direct subplans and idle times contained in both plans. This transformed plan is then executed in simulation. Its behavior has not changed, therefore it must be transformed like the original plan. But this time, the monitoring is already included in the plan, so that a second transformation rule is chosen.

This rule breaks up the separation of the two plans and produces a new plan containing the direct subplans of the two original plans. The partial ordering of these new plan steps is determined by a scheduling algorithm based on the data observed during the simulated plan execution, that is the duration of each plan step and the available free time. This second transformation rule is special in that it uses dynamically acquired data for the transformation.

5.5 Related Work on Transformational Planning

TRANER can be viewed as a modern version of Sussman's *Hacker* (1973; 1977). Like *Hacker*, TRANER aims at learning plan libraries by debugging the flaws of default plans. Unlike *Hacker*, which worked in the idealized blocks world domain, TRANER applies to real-world robot control.

Other transformational planners are *Chef* (Hammond 1990) and *Gordious* (Simmons 1988). The main difference between these systems and TRANER is that TRANER reasons about concurrent robot control programs while *Chef* and *Gordious* reason about plans that are sequences of plan steps. Another difference is that they try to produce correct plans while TRANER adapts plans to specific environments and corrects failures during execution. Bothelho and Alami (2000) show how robots can enhance plans cooperatively by merging partially ordered plans using social rules.

TRANER is most closely related to more recent variants of transformational planning techniques. Most notably, to McDermott's (1992b) *XFRM* planner that performs improving transformations on an idealized grid world agent. Beetz (1997; 2000; 2001; 2002a) successfully applies transformational planning mechanisms to autonomous robot control, in particular office delivery tasks for a robot without manipulators. Transformational planning, however, is particularly promising and challenging if the robot's tasks are the manipulation of objects. This is what TRANER does. Still, several of TRANER's methods for plan representation and specifying transformation rules are inspired by the work of McDermott and Beetz and have been further extended (for details see pages 80 and 85).

5.6 Summary

The chapter starts by motivating and introducing the syntax of transformation rules and their processing. In contrast to other transformational planners TRANER's plan language is very rich and represents all levels of abstraction, not just the most abstract one. This makes the matching of plans to transformation rules a lot more difficult. The syntax for the rules takes this complexity into account, while still providing an easy-to-understand representation.

Furthermore, the rules are structured clearly into syntactical and semantical application criteria. They make the transformation steps explicit and plainly show what the resulting plan will look like.

After specifying the structure of the transformation rules, we have demonstrated how the procedure of transforming plans takes place. When a transformation rule matches a plan in several ways, there can be several output plans, depending on the specification in the rule. In complex transformations several such branches can occur.

After that, we have described in detail our transformation rule library with a running

example showing which transformations applied to the default plan lead to which resulting plans. The transformation rules can be classified into syntactical transformations, ones that reorder plan steps, use external or robot resources, integrate or modify auxiliary goals, and ones optimizing the concurrent execution of plans. All our rules are general in the sense that they are applicable to a wide range of different plans, which is also shown in Figure 5.9 where some transformation rules apply to multiple plans.

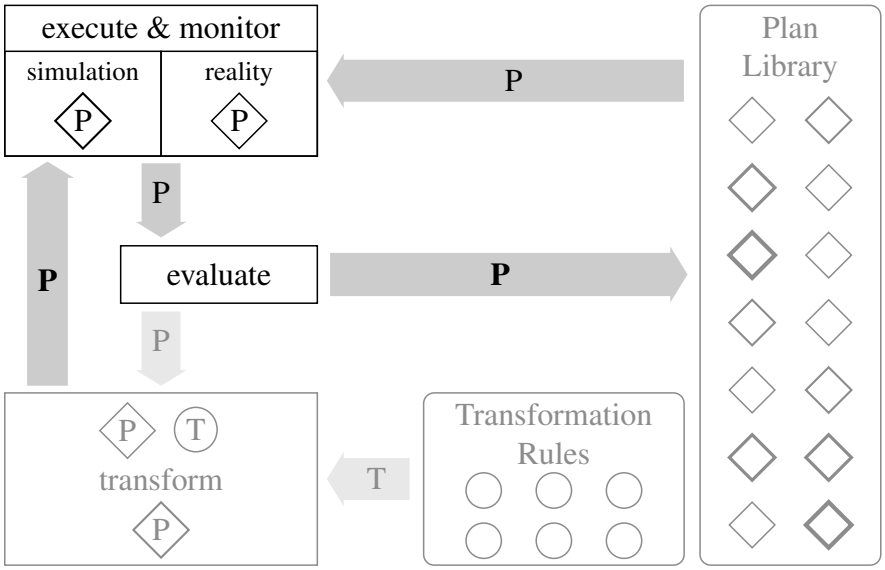
Chapter 6

Plan Execution and Evaluation

In the previous chapters, we have presented how plans are represented and organized in a plan library and how they are enhanced by plan transformations. It is now time to explain how these components work together in the TRANER system (as shown in Figure 6.1). Therefore, in this chapter we show how the plan execution and monitoring works and how TRANER decides if a plan is superior to another.

First we present our execution environment and argue why a realistic simulation of the robot is necessary for the TRANER system to work. Then we show how we monitor the plan execution and how the monitoring results can be combined in an evaluation function. Finally, we discuss the general applicability of this procedure to arbitrary plans.

Figure 6.1 Part of TRANER described in this chapter.

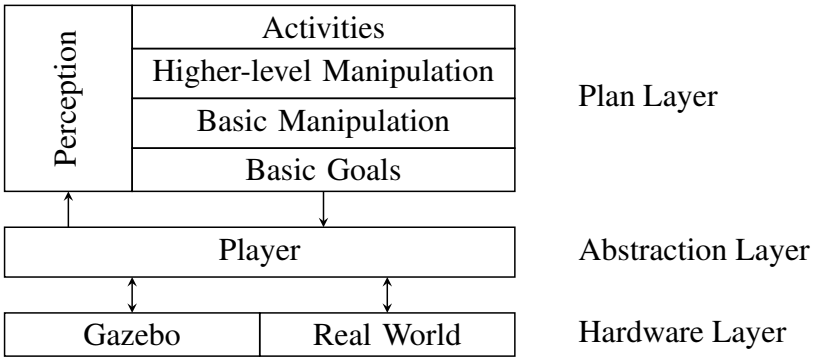


6.1 Plan Execution

It is now time to provide more detail on our simulated environment, which we have already described briefly in Chapter 2 and close the gap between the basic goals described in Chapter 4 and the low-level functionality provided by our environment.

Figure 6.2 shows our software architecture. The higher levels consist of the plan library from Chapter 4, using perceptual data. The perception and plan layer communicates with a program called Player. This layer provides percept data and receives control commands. Player abstracts away from the underlying hardware. This means that the control program doesn't know if it is running on a real robot or a simulation. In our case, the hardware layer consists of a simulator implemented with the Gazebo simulation environment, which in its turn is based on the physics engine provided by ODE.

Figure 6.2 Software architecture of our system.



6.1.1 The Environment

Our simulated environment (see Figure 2.1 on page 16) consists of a kitchen with several pieces of furniture such as cupboards, a table and a work top and solid objects such as plates, cups, pots, sieves, and cutlery. Our robot is a simulated B21 with two arms and a camera. We have already argued in Chapter 2 that this simulation is very realistic. In the following we describe some intricacies and details of the simulation.

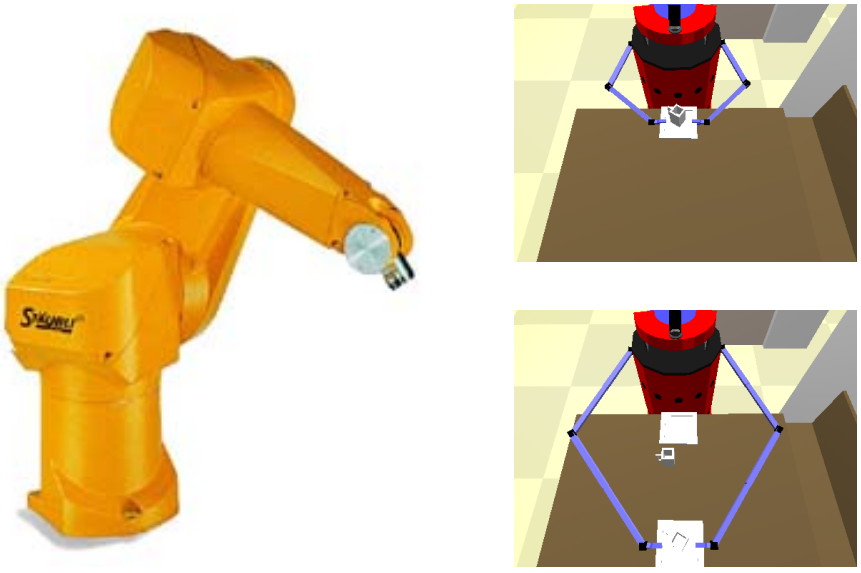
Beside the solid objects, an important ingredient for kitchen work is water. Unfortunately, ODE only simulates solid objects and their interaction. Therefore, we implemented a separate Gazebo process, which handles (solid) water objects and hides or shows them when necessary. This means that when the robot opens the tap, water objects fall out of it. If no container is provided to catch them, the water cubes disappear in the sink, or if the tap is opened over the worktop the water cubes stay there. For boiling pasta not every

noodle is modeled, instead pasta is modeled as solid cuboids with a certain weight, even after boiling.

The manipulation tasks in a kitchen can be quite intricate. For example, for opening cupboards, the robot must grasp the handle very accurately and provide a sophisticated interaction between arm movement and robot navigation (because the robot must move out of the way of the door). Also the grasping of objects inside cupboards is not easy, because the free space is very restricted. In our work, however, we are not interested in precise arm control. Therefore, we equipped the simulated kitchen with devices that are usually not found in an average kitchen, but could easily be built in any normal kitchen without too much effort. Such devices include automatically opening and closing doors, a remote controlled tap, a remote controlled cooker, and automatically extending and retracting boards inside the cupboards to make grasping of objects easier. The robot can control these devices by wireless communication.

Our robot is built after a B21, which originally comes without arms. We added two arms, which are constructed along the Stäubli RX90 (Figure 6.3) robot arm[1], used in industrial environments. To make the arms more agile and expand their operating region, we added four more joints: two joints in the shoulder and two slider joints in the upper and lower arms. The additional shoulder joints make the arms more agile and enable a more natural movement. The slider joints make the arms extendable, so that the robot's work space gets bigger as shown in Figure 6.3.

Figure 6.3 The original Stäubli RX90 (left) and our robot's extendable arms (right).



[1]It is a successor of the Unimation PUMA 560

The robot is equipped with a pan-tilt-zoom camera mounted on top of the robot. With this camera, the robot perceives all objects in its field of view. Actually we have no vision algorithms running for detecting objects, instead we take the ground truth positions from Gazebo and implemented the possibility to add noise to the data for getting a realistic simulation. In the future it is planned to learn the accuracy of the state estimation in order to get a more realistic simulation.

6.1.2 Basic Functionality

In the last section we have described the hardware layer of Figure 6.2. Now we give a brief overview of the basic control functionality we implemented in the abstraction layer (the Player interface). This comprises navigation, arm control, and the camera.

For navigation we implemented two P-controllers, one for the translational velocity and one for the rotational velocity, which are assumed to be independent. Only when the robot's angle towards its target point is too large, it first turns on the spot and then switches to the P-controllers. For small distances (below 1 m), the robot has the option to move backward instead of forward. The choice is based on how much turning is involved in the forward and backward trajectories.

We designed the arms along the Stäubli RX90 robot, so that we could use its inverse kinematics. The four joints we added are not taken into account in this calculation. They are controlled by approximation methods. This means that when the arm is to move to a position, a heuristic determines the positions of the four additional angles, for example to extend the forearm. This changes the parameters of the Stäubli RX90 arm, by setting the length of the forearm to the new length for instance. Then the normal Stäubli RX90 inverse kinematic is applied. The opening position of the gripper is controlled by a P-controller.

The camera's pan and tilt are also adjusted by P-controllers. Moreover, there is the option to follow an object with the camera (the robot cannot perceive objects outside its field of view). This is particularly useful when the robot is picking up or putting down objects. This functionality can be used with the programming construct `(with-camera-tracking body)` inside the plans.

6.1.3 The Need of Simulation

The architecture in Figure 6.2 doesn't address the question whether to work in simulation or on a real robot. However, for the whole process of plan transformation to work, a simulation is always necessary. In our case, the execution environment and the simulation are identical. When working with a real robot, an additional simulation of this environment is needed. There are three arguments supporting this claim: safety, efficiency, and reliability.

Although a kitchen is not the most dangerous place, there are still myriads of ways to perform plans with unwanted effects like braking objects or setting the house on fire. For the sake of safety, plans should always be tested in simulation before being tried in the real household. By using simulation even syntactically invalid plans can be tested (the controller aborts and the robot doesn't do anything afterwards) and can therefore be evaluated. This makes the design of plan transformation more feasible. Specifying good transformations only ensures that not too many garbage plans are generated.

Even if we were bold enough to let the robot try its newly transformed plans in the real kitchen directly, for some activities only a very small number of plans could be tested. For example, when the robot has transformed the plan for making a cake. In simulation, it can test this plan several times, but who would need dozens of cakes (or the remains of unsuccessful tries) only because the robot is practicing a new plan.

Another aspect to be considered is that newly transformed plans should be tested several times, not only once. Consider the behavior of plans in the light of failures. Not every failure causes the whole plan to fail and some failures are intrinsic in the lower-level plans, not in the plan to be tested. For assessing the importance of failures that have occurred during testing, it is necessary to test a plan several times and then decide if a failure is intrinsic in the plan (if it occurs several times) or has been observed as an instance of the uncertainty in the environment (if it is observed infrequently). In all, when plans are transformed, each transformed plan must be tested several times against several versions of other transformed plans achieving the same goal. This can only be done in simulation.

Our architecture provides the possibility to use the real robot and a simulation at the same time. For obtaining a good simulation of an environment, a Gazebo model of the environment can be defined. This can be further refined by learned models of the world's dynamics. The simulation environment has to be realistic but not perfect. Successfully validated plans in simulation are also validated when executed in the real world, to see if the new plans really have a performance gain. This ensures that differences between the real and the simulated world are detected and can be learned.

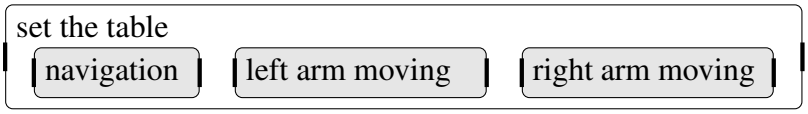
6.2 Monitoring

For evaluating a plan's performance we need data about its execution, for example on resource usage. An important requirement for monitoring a plan that might have been transformed or still is to be transformed is that the monitoring is defined and performed outside the plan. We simply don't want any special monitoring code inside the program, although this would be the most straightforward way to get data out of a program. But if we added the monitoring code to the plan, the transformations would be a lot more complicated.

For monitoring plan execution from the outside we use RoLL (Robot Learning Lan-

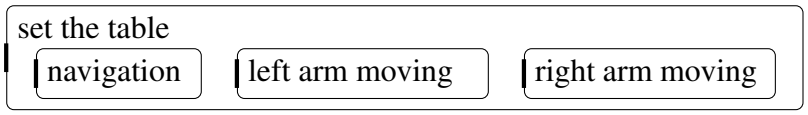
guage) (Kirsch and Beetz 2007), a language extension of RPL that embeds learning into control programs. RoLL allows a simple definition of experiences for learning, which we use here for acquiring our data. Without explaining the details of RoLL, we show how we can define the data we want with the example shown in Figure 6.4.

Figure 6.4 Raw and abstract experience definitions for monitoring the table setting plan.



set the table/navigate/left arm moving/right arm moving $_{\text{begin}}$ →*timestep*
set the table/navigate/left arm moving/right arm moving $_{\text{end}}$ →*timestep*
navigate $_{\text{interval}}$ →*robot position*
left arm moving $_{\text{interval}}$ →*left arm position*
right arm moving $_{\text{interval}}$ →*right arm position*

(a) Raw experience definition.



set the table $_{\text{begin}}$ →*duration*
navigate/left arm moving/right arm moving $_{\text{begin}}$ →*duration, count, distance covered*

(b) Abstract experience definition.

The definition of monitoring data (or as it is called in RoLL, an *experience*), is based on the concept of hybrid automata. A hybrid automaton is a tool for modeling discrete as well as continuous behavior of a system. By defining an automaton we tell the system when to be attentive to interesting data. RoLL supports a hierarchical nesting of hybrid automata, so that a fine granularity of data can be defined. Figure 6.4(a) shows an automaton for acquiring data while executing the set the table plan. The highest-level automaton is the one that corresponds to the plan "set the table". It contains three subautomata, which are active when the subplans for navigation, moving the right arm and moving the left arm respectively are called. There is no ordering constraint on when the subautomata can be active. More importantly, the definition doesn't specify how often a subautomaton is active. So the navigation plan is registered every time it is active in the course of executing one table setting plan.

By only specifying interesting time periods in the form of hybrid automata, we haven't said anything about the data we are interested in. For each automaton we can specify data to be acquired at the beginning and end of each automaton activity or data to be recorded

continually while the automaton is active. The data can originate from global variables or local variables in any RPL task. In the example in Figure 6.4(a) we record the time step of the beginning and end of the high-level automaton as well as its subautomata. Besides, for the subautomata we want to know more about the robot's movements. Therefore, when the navigation plan is active, the robot's position is recorded continuously. Similarly, the positions of the robot's arms are logged.

With the automaton definition of Figure 6.4(a) the system automatically monitors the data we have specified. But the information gathered is not quite what we want. We aren't really interested at what exact time a navigation plan started and when it ended, but rather how long the plan execution lasted. RoLL offers the concept of *abstract experience*, which allows to transform a raw experience (the data acquired with a definition like the one in Figure 6.4(a)) to more abstract concepts.

The definition of abstract experiences is analogous to raw experiences. Figure 6.4(b) depicts the automaton definition of the abstract data extracted from the raw experience. The automaton hierarchy is the same. Only the data associated with it is calculated from the data of the raw experience. For all plans we calculate their duration. For the subplans for navigation and arm movement, we additionally store the number of occurrences of the respective subplan and the distance traveled, for the navigation plan this is the distance the robot has moved, for the arm movement the trajectory of the arm.

With the definitions of the raw and abstract experiences RoLL automatically collects the desired data each time the table setting plan is performed. The plan itself doesn't contain any traces of the data gathering process. Some results of monitoring the plan are presented in Figure 6.5. The table shows the data of five executions of the table setting plan (these plans aren't identical, some of them have been transformed from the original plan). It tells us that the first (the original) plan needs 216.41 s for completing its task. During the plan execution, the robot navigated 8 times covering a total distance of 16.36 m in a total time of 95.04 s. We can further see that the right arm was used much more often than the left one.

Figure 6.5 Monitoring data for table setting plans.

	duration	navigation			left arm			right arm		
1	216.41 s	95.04 s	8	16.36 m	44.39 s	26	5.10 m	86.26 s	50	11.36 m
2	219.02 s	96.27 s	8	16.38 m	43.29 s	26	4.99 m	86.51 s	50	11.19 m
3	194.06 s	86.11 s	8	13.67 m	34.80 s	22	3.50 m	76.78 s	46	9.54 m
4	210.28 s	94.28 s	8	14.11 m	69.39 s	39	8.87 m	63.76 s	37	7.60 m
5	186.06 s	86.30 s	8	11.40 m	58.84 s	34	7.13 m	55.39 s	34	6.03 m

6.3 Evaluating Plans

The data acquired in the monitoring step (Figure 6.5 and 2.3 on page 24) is then used to calculate a performance value for comparing different plans. The criteria used in this function can vary depending on the application and on the plan. The plans, whose data is depicted in Figure 6.5 can be compared along several lines. One criterion is the overall time needed. Here Plan 5 is the clear winner. But if we prefer the robot to reduce arm movement, Plan 3 pips Plan 5 both in the total time the arms are moving (111.58 s compared to 114.23 s of Plan 5) and the distance covered by the arms (13.04 m compared to 13.16 m). Such criteria can be combined arbitrarily.

In our current approach we define the monitored data and evaluation function manually for each plan. However, this is only feasible for a restricted number of plans and cannot be used as a general solution. Besides, when plans are transformed, other data might become available and the evaluation criteria might change. For example, in the original table setting plan all plates are carried one by one. One transformation is to stack the plates. The average number of stacked plates can be used as an evaluation criterion, whereas this doesn't make sense in the original plan where the stacking idea isn't used at all.

For a general approach, common evaluation criteria for all plans must be defined. Since they are independent of the individual plan, they can only take into account aspects of the robot and the environment. Such evaluation aspects are the duration of the plan, the distance traveled, the usage of the arms, the number of objects moved, number of failures or user satisfaction.

6.4 Summary

The execution and evaluation part of TRANER assumes that a simulation of the world the robot is acting in is present. In our case the simulation and the real world are identical. We use a simulated B21 robot, which is supplemented by two arms designed after the Stäubli RX90 robot, with four additional joints. Using the Gazebo simulator we have a sophisticated tool for physical simulation and a 3D graphical display.

The Player interface provides an abstraction layer, which enables the use of different robots with the same program. Besides, low-level routines for navigation and manipulation can be implemented efficiently in Player.

For plan evaluation, the execution of the plans must be observed. In order to get hold of all relevant information without changing the control program itself, we use the data recording facilities of RoLL, a robot control language that integrates learning into control programs. The RoLL constructs allow an abstract description of the data needed and can record all local variables within the program.

The evaluation takes place by applying a performance measure to the recorded data

and comparing the computed value to some reference value. In this work we use the duration of activities as the main performance measure. Other possibilities are also discussed in Section 7.1.1 on page 120.

Chapter 7

Evaluation

We have explained in great detail our approach of transforming default plans to adapt to specific environments. In this chapter we proof empirically the necessity of transforming plans instead of hand coding them for every situation.

In Chapter 5 we have already demonstrated that our transformation rules are general in the sense that they apply to different plans. Thus, the number of plans and transformation rules that have to be specified is less than the number of plans that are generated and used by the robot.

In the following we demonstrate not only that the robot performance improves significantly with the transformed plans, but also that for different situations and environments different plans are to be preferred. This second point is very important. It shows that it is impossible to implement one set of plans that works perfectly in all environments. However, we can develop default plans that work in all situations, but suboptimally. Only by adapting the behavior using empirical evidence about which plan is to be preferred, the robot can show the desired performance.

The default plans need to be robust, general, transformable and enable cognitive behavior as argued in Section 3.1. During the experiments the execution was very robust. The plans monitor eight kind of failures and the robot always recognized if a failure occurred. 86 % of the failures could be recovered from, otherwise the robot could at least explain the type of failure. Flexibility and reliability also require synchronized concurrent activity: on average 10–15 threads of activity are executed concurrently. During one run of the table setting plan approximately 700 conditions (perceptual changes, failures) are tested.

For the different experimental settings the default plans worked without modifications. This was ensured by the plan library presented in Chapter 4 which consists of 2 activities, 22 higher-level manipulations plans and 12 basic-manipulation plans using 27 different types of parameter designators. Together with the concept of object designators the plans showed to be general. The hierarchy of both activities is 7–9 levels deep when executed.

The experiments will show, that the performance of the plans can be improved by transformations up to 45 % in the best case.

We have conducted several experiments in the context of table setting and one where both activities, setting the table and boiling pasta, are involved.

7.1 Setting the Table

In the following experiments, the task is always to set the table. There are four people (Alvin, Theodore, Simon, Dave). The table is to be set for all four or only some of the persons. The combinations we used in our experiments are shown in Figure 7.1. We only used nine out of the 15 possible combinations of setting the table for one to four people, where all persons have their fixed seat. We selected these cases, because they cover all interesting combinations of how the robot positions itself at the table.

Figure 7.1 The combination of persons for which the robot had to set the table in our experiments. The letters (first letter of a name) below a table indicate the mnemonic used in the results tables later.

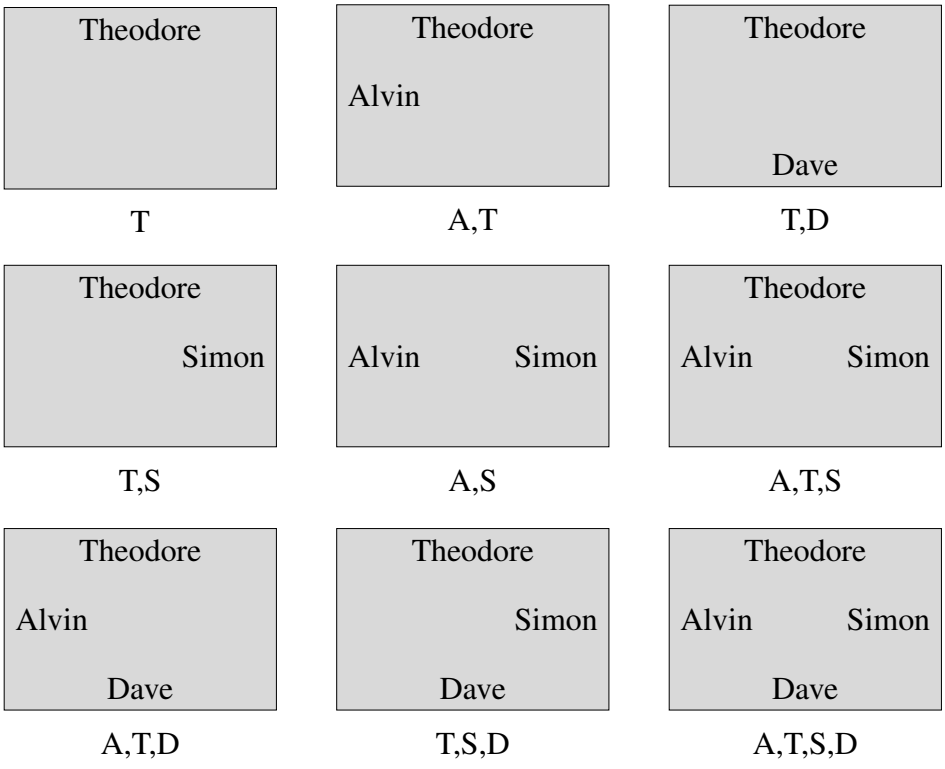
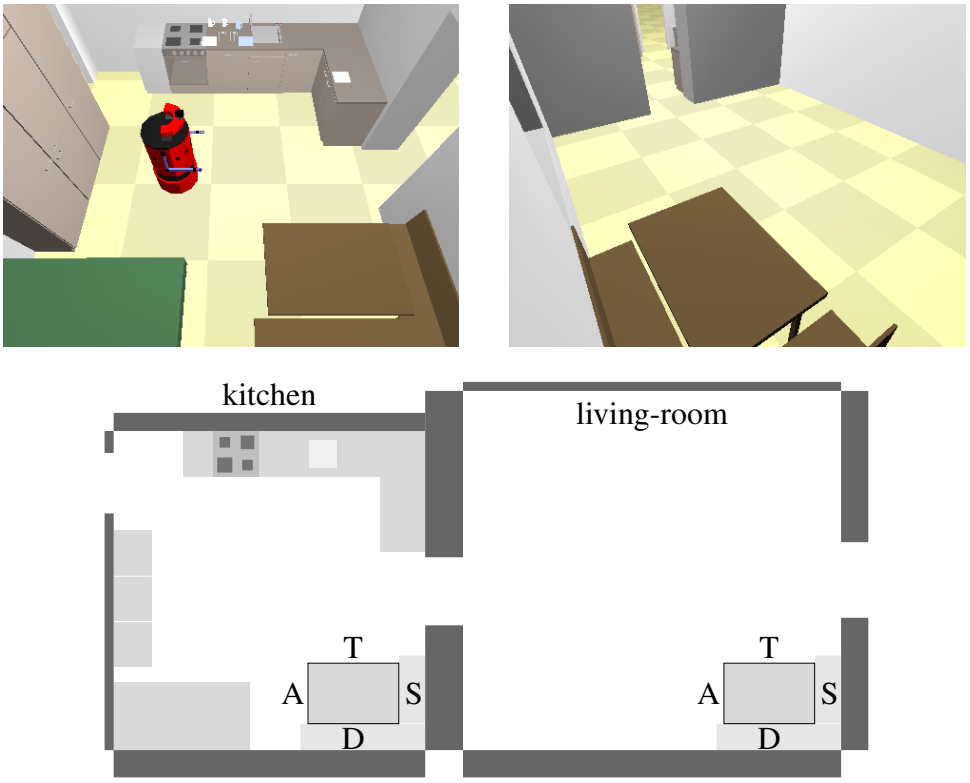
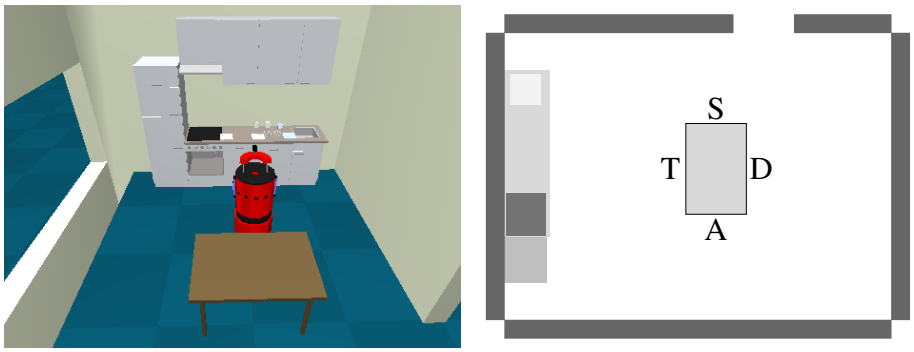


Figure 7.2 Views of our two households. A, T, S and D denote the preferred seating locations at the tables of Alvin, Theodore, Simon and Dave respectively.



(a) Household A with one table in the kitchen and one table in the living-room.



(b) Household B with one table in the kitchen.

In our simulation we have two different households. Household A (see Figure 7.2(a)) is composed of a kitchen in which people can eat and a living room, where the meals can also be served. In household B (see Figure 7.2(b)) there is only a kitchen table. Most of the experiments use household A.

Not only the environment, also the robot's abilities can change. The basic version of the robot can only transport objects with the grippers. An advanced feature, which can be installed in addition, is a tray on top of the robot. Most experiments use the basic version, but we also examine the plans used when the tray is present in experiment 4.

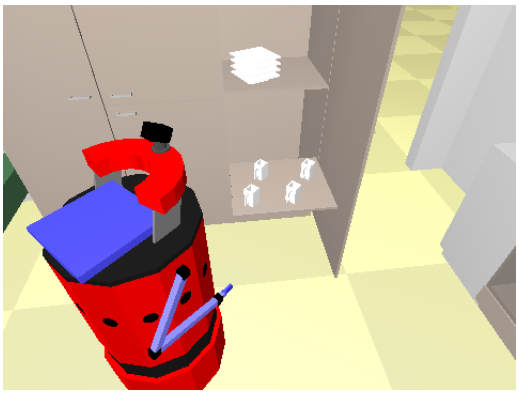
Moreover, all experiments compare the same set of plans (except the one with the tray, which allows additional transformations). Most of these plans were explained at length in Chapter 5 and occur in Figure 5.9 on page 93. Table 7.1 summarizes all the plans that are used in the following evaluation with a textual description of what they do. For better readability the plans are numbered and have an additional symbolic description in all the tables of the experiments.

With the different combinations of persons, rooms and plans each experiment examines between 168 and 336 runs of plan execution. Depending on the experiment setting, an average plan execution run needs between 5.4 min and 6.5 min. The total time of executing plans for the experiments was approximately five days.

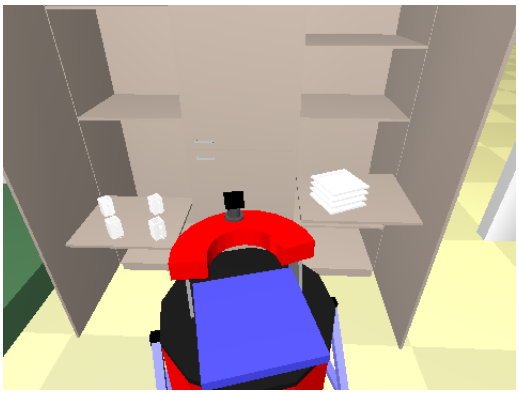
7.1.1 Experiment 1: Plates and Cups Inside one Cupboard

In our first experiment the plates and cups are stored away in one cupboard. The plates are stacked on one board, the cups are arranged next to one another (see Figure 7.3(a)). The robot had to set the table for each combination of persons shown in Figure 7.1.

Figure 7.3 Initial settings of experiments 1 and 2. The plates are stacked on one board and the cups are on another board, either in the same cupboard or in a second one.



(a) Experiment 1.



(b) Experiment 2.

Table 7.1 Plans used in setting the table experiments. The names of plans as they appear in Figure 5.9 on page 93 are given in brackets. The mnemonic for each plan shows if plates are stacked (several lines) or carried separately (single line), if cups (filled rectangles) are stacked on a plate, carried with both arms (symbolized by two rectangles) or carried one by one. A number in brackets specifies if the plan is adapted to a specific number of persons to lay the table for.

Plan		Description
Plans introduced in Section 5.4 on page 88		
P_0	_▪	Default plan. Carry all plates and cups one by one. [P_{default}]
P_1	▮	For each pair of plate and cup stack the cup on the plate and carry them together. [$P_{\text{stack-seq-1}}$]
P_2	≡▪	Stack all plates and move the whole stack, carry cups one by one. [$P_{\text{stack-plates}}$]
P_3	_▪▪	Move plates one by one, carry cups two at a time, one in each hand. [$P_{\text{use-both-arms}}$]
P_4	≡▪▪	Stack plates and use both arms. [$P_{\text{stack-plates-use-both-arms}}$]
Plans for a fixed number of persons generated by applying T_7 and then T_3 and T_5		
P_5	≡▪[2]	The table is to be set for two persons. Stack both plates and one cup on top of the plates and carry the last cup separately.
P_6	≡▪[3]	The table is to be set for three persons. Stack all plates and one cup on top of the plates and carry the last two cups one by one.
P_7	≡▪▪[3]	The table is to be set for three persons. Stack all plates and one cup on top of the plates and carry the last two cups together by using both arms.
P_8	≡▪[4]	The table is to be set for four persons. Stack all plates and one cup on top of the plates and carry the last three cups one by one.
P_9	≡▪▪[4]	The table is to be set for four persons. Stack all plates and one cup on top of the plates. Then carry two cups together by using both arms and the last one separately.
P_{10}	≡≡▪[4]	The table is to be set for four persons. Stack two plates and one cup and carry them to the table. Then stack the remaining two plates and one cup. Carry the last two cups separately.
P_{11}	≡≡▪▪[4]	The table is to be set for four persons. Stack two plates and one cup and carry them to the table. Then stack the remaining two plates and one cup. Carry the last two cups together by using both arms.

Beside the plans listed in Table 7.1, this set-up allows the additional transformations of optimizing the auxiliary goals for opening and closing the cupboard doors and extending and retracting the boards inside the cupboards. As these transformations can be applied to any of the plans of Table 7.1, they are denoted with superscripts. Plan P_n^d is obtained by applying the door optimizing transformation to plan P_n. Similarly, Plan P_n^b results from the board optimizing transformation applied to P_n^d (optimizing boards only makes sense when the doors aren't closed after every gripping action).

Performance Measures

For all combinations of person set-ups and plans we recorded several possible performance indicators. Table 7.2 shows an excerpt for the case where the table is to be set

Table 7.2 Experiment 1: Data to be used for different performance measures when setting the table for Theodore and Dave. The main aspects are the duration to complete the plan, the usage of navigation and each of the arms, and the usage of the cupboards. For the robot resources (navigation, left arm, right arm) we give the total time where these resources are used, the number of times they are applied, and the total distance traveled (either by the robot or the arm). The same parameters are given for the cupboard doors and boards, except the distance.

	plan	duration	navigation			left arm			right arm			doors		boards	
1	P_4^b ≡▪▪	196.7 s	52 s	8	6 m	60 s	34	8 m	62 s	34	9 m	9 s	2	11 s	4
2	P_4^d ≡▪▪	208.3 s	54 s	8	6 m	59 s	34	8 m	62 s	32	9 m	10 s	2	19 s	6
3	P_2^b ≡▪	227.0 s	73 s	8	9 m	60 s	34	8 m	62 s	34	9 m	10 s	2	13 s	4
4	P_4 ≡▪▪	227.4 s	45 s	8	6 m	54 s	32	7 m	62 s	34	9 m	42 s	6	19 s	6
5	P_5^b ≜▪[2]	234.6 s	52 s	11	6 m	74 s	42	9 m	60 s	36	6 m	10 s	2	11 s	4
6	P_2^d ≡▪	238.7 s	76 s	8	9 m	59 s	34	8 m	60 s	34	9 m	10 s	2	19 s	6
7	P_3^b _▪▪	241.9 s	78 s	8	9 m	75 s	41	11 m	78 s	41	11 m	10 s	2	11 s	4
8	P_5^d ≜▪[2]	244.7 s	56 s	10	6 m	76 s	42	9 m	51 s	31	6 m	10 s	2	24 s	8
9	P_2 ≡▪	253.3 s	64 s	8	9 m	58 s	34	8 m	55 s	32	8 m	41 s	6	18 s	6
10	P_3^d _▪▪	260.7 s	81 s	8	9 m	73 s	41	11 m	75 s	40	11 m	10 s	2	25 s	8
11	P_1^b ▪	269.0 s	53 s	13	6 m	79 s	46	10 m	92 s	48	10 m	9 s	2	11 s	4
12	P_5 ≜▪[2]	273.2 s	47 s	10	6 m	72 s	43	11 m	50 s	30	6 m	55 s	8	26 s	8
13	P_0^b _▪	281.5 s	98 s	9	13 m	106 s	58	15 m	54 s	28	7 m	9 s	2	11 s	4
14	P_0^d _▪	288.6 s	99 s	8	13 m	73 s	41	11 m	75 s	40	11 m	10 s	2	25 s	8
15	P_3 _▪▪	288.9 s	67 s	8	9 m	64 s	39	10 m	73 s	41	11 m	56 s	8	25 s	8
16	P_1^d ▪	322.1 s	57 s	14	6 m	67 s	38	9 m	111 s	64	12 m	10 s	2	38 s	12
17	P_0 _▪	324.4 s	86 s	9	13 m	97 s	56	14 m	52 s	30	8 m	56 s	8	25 s	8
18	P_1 ▪	377.7 s	46 s	14	6 m	73 s	41	11 m	114 s	63	13 m	82 s	12	36 s	12

for Theodore and Dave. The plans are ordered by the time it took to complete them. This means when the duration is used as performance measure, the plan where plates are stacked and cups are carried in both hands (P_4^b, line 1) is the best. Interestingly, the worst plan is not the default plan, but the one where pairs of cups and plates are carried together (P_1, line 18), because the cupboard doors and boards are manipulated excessively often. However, with optimized boards P_1^b (line 11) shows a better performance than P_0^b (line 13).

Although we use the duration of the plan execution as our performance measure throughout the rest of the experiments, we should point out that other performance measures can result in the preference of other plans. Table 7.2 indicates that the duration is closely linked to the time needed for navigation, unless the manipulation tasks, especially the opening and closing of cupboards requires an undue amount of time. The way covered by the robot relates very closely to the time needed, whereas the number of navigation tasks doesn't provide much information.

Usually, the most expensive part of the robot are its arms. Therefore, one goal might be not to overuse the arms. The added duration of using both arms is minimal for the plan in line 4 with only 116 s compared to 187 s with the plan in line 18. Or it might be desirable to use both arms equally instead of overusing one and sparing the other. Here the plan in line 5 wins with a difference of 1 s in arm usage versus a maximum of 52 s when using the plan in line 13. Depending on the robot hardware, when the arms are of different value and dexterity, the opposite criterion might be interesting. For example, if the left arm is to be used as rarely as possible, the plan in line 4 would prevail.

We could find more aspects for evaluating the different plans including parameters outside the robot like the frequency of manipulating cupboard doors. Moreover, the different criteria can be weighted and combined to more complex performance measures. The question of finding a perfect performance measure is outside the scope of this work. However, with regard to the examples we have given, we observe that the duration is not a bad choice as a performance measure. No matter which criterion we apply, the best plan is always among the top 5 plans when sorted by duration and the worst plans can be found at the bottom of the list. Besides, in our simulation, there is no need to preserve specific parts of the robot hardware.

Performance Enhancement

Table 7.3 shows the duration of all plans when setting the table for three persons (the three combinations contained in our set-ups of Figure 7.1) either in the kitchen or in the living room of household A. The plans where boards are optimized are not listed, because the robot loses the cup after it collides with a board (see Figure 7.4(a) on page 126) and the plans fail[1].

[1]The collisions only occur for three and four persons, because the robot has to reach deeper into the cupboard for gripping the cups.

Table 7.3 Experiment 1: Comparison of durations of all successful plans for different experiment set-ups.

	kitchen table			living-room table		
plan	A,T,S	A,T,D	T,S,D	A,T,S	A,T,D	T,S,D
P_0	489.5 s	502.9 s	473.6 s	625.7 s	632.7 s	634.5 s
P_0^d	447.4 s	458.2 s	424.3 s	570.0 s	580.7 s	583.8 s
P_1	573.8 s	568.5 s	544.0 s	625.3 s	621.9 s	611.9 s
P_1^d	463.7 s	448.1 s	476.8 s	530.4 s	528.8 s	516.6 s
P_2	402.3 s	395.2 s	360.6 s	482.5 s	475.9 s	468.2 s
P_2^d	378.4 s	370.5 s	335.8 s	455.3 s	448.5 s	441.1 s
P_3	500.1 s	499.5 s	451.3 s	584.3 s	566.9 s	573.5 s
P_3^d	434.8 s	440.2 s	400.4 s	530.4 s	519.4 s	517.9 s
P_4	410.8 s	381.9 s	336.7 s	440.9 s	413.1 s	405.8 s
P_4^d	369.6 s	356.1 s	309.7 s	408.1 s	382.6 s	377.2 s
P_6 [3]	442.6 s	411.5 s	385.2 s	478.7 s	462.6 s	455.5 s
P_6^d [3]	382.9 s	366.6 s	344.9 s	441.7 s	418.1 s	409.1 s
P_7 [3]	419.5 s	399.6 s	371.4 s	457.4 s	437.7 s	414.0 s
P_7^d [3]	372.5 s	354.6 s	327.2 s	411.3 s	404.6 s	365.4 s

Table 7.4 Experiment 1: Summary of best plans when using the duration as performance measure for all combinations of persons and both rooms in household A.

	Household A					
persons	kitchen table			living-room table		
T	P_0^b		-2.2 %	P_1^b		-11.4 %
A,T	P_4^b		-23.9 %	P_4^b		-30.1 %
T,D	P_4^b		-39.4 %	P_4^b		-45.3 %
T,S	P_4^b		-30.2 %	P_4^b		-36.9 %
A,S	P_4^b		-31.5 %	P_4^b		-33.4 %
A,T,S	P_4^d		-24.5 %	P_4^d		-34.8 %
A,T,D	P_7^d [3]		-29.5 %	P_4^d		-39.5 %
T,S,D	P_4^d		-34.6 %	P_7^d [3]		-42.4 %
A,T,S,D	P_4^d		-32.0 %	P_4^d		-42.7 %

The table clearly shows that the difference in performance varies greatly for different plans. For example, the best plan for serving Theodore, Simon and Dave in the living room only requires 365.4 s (P_7^d) compared to the default plan needing 634.5 s, which corresponds to an enhancement of 42.4 %.

Although the difference is not always that striking, the difference between the best and worst plan for a set-up is in most cases more than 20 %. In Table 7.4 for each combination of persons and each room in household A the best plan is listed together with the relative change compared to the default plan.

Selecting the Best Plan

The third point we want to emphasize in this experiment is that — irrespective of the performance measure — the best plan for a task depends on the parameterization of the plan.

The shaded fields in Table 7.3 indicate the best value for each combination of persons and rooms. The best plan is either the plan where the plates are stacked and the cups are carried with both arms, which works regardless of the number of persons to attend the meal, or the plan tailored to three persons, where first all plates are carried with one cup on top and the last two cups are carried using both arms.

The preference of which plan to use varies both in the combination of persons and in the room where the meal is served. So when Alvin, Theodore and Dave are to have the meal in the kitchen a different plan should be chosen as to when they decide to eat in the living room. In the same way, the plan for setting the table in the kitchen varies when Simon attends the meal instead of Alvin.

Table 7.4 shows more instances of different plans being preferable in different situations. When setting the table for only one person the stacking of plates doesn't help. Therefore, these cases require different plans, either the default plan or the one where the cup is stacked on the plate.

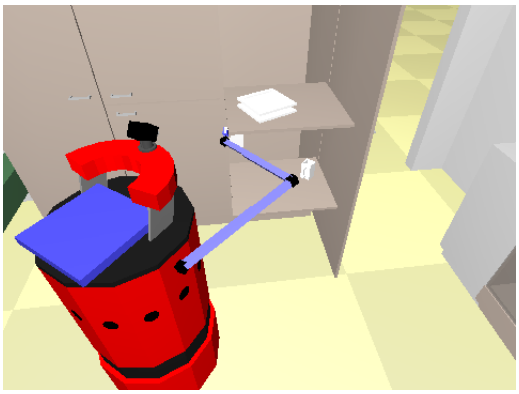
7.1.2 Experiment 2: Plates and Cups Inside two Cupboards

We now regard a slight variation of experiment 1, where the plates are kept in one cupboard and the cups in another as shown in Figure 7.3(b). The results for plan improvement and that different situations require different plans are comparable to the ones of the last experiment. Here we want to emphasize the robot's behavior in the light of failures.

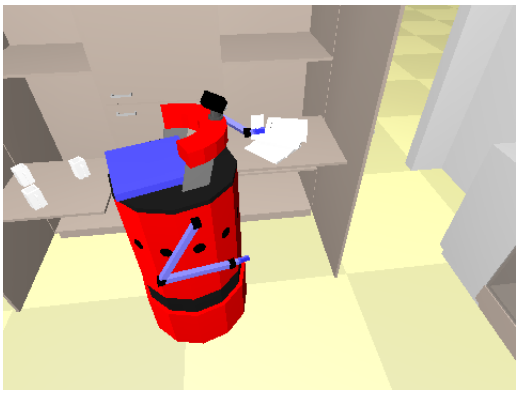
In the previous experiment we mentioned that the plans where the handling of the boards is optimized fail for three or four persons. In this case, where the plates and cups are stored in two adjacent cupboards, the optimization of the boards always leads to a failure (see Figure 7.4(b)).

The recognition of failures works very reliably in TRANER. To achieve reliability and flexibility our plans monitor eight types of failures. None of the plans failed without the robot knowing which failure occurred. In many cases, the failures can be repaired by the robot. In 86 % of the cases where a failure was detected, the top-level plan could be accomplished nevertheless. This level of reliability is very impressive in the robotics domain. Nonrecoverable failures include ones where an object falls out of the robot's gripper to a position where the robot cannot retrieve it. An instance of this problem is when plates are lying upside-down. The robot's grippers aren't dexterous enough to grip the plates from these positions.

Figure 7.4 Failures occurring when boards are kept slided out.



(a) Cup gripped in right arm collides with board and falls down.



(b) Robot knocks over the stack of plates while turning.

7.1.3 Experiment 3: Dishwasher

As a further variation we considered the case when the dishes aren't stored in the cupboards, but have to be taken out of the dishwasher. Simulating a fully equipped dishwasher is complicated, because the rack for keeping the dishes would have to be modeled to great detail, a work we haven't done. The set-up of our experiment as shown in Figure 7.5(a) on page 128 models the dishwasher scenario by placing the dishes on the worktop without stacking them, because the robot would have to take the plates out of the dishwasher one by one.

Table 7.5 shows the best plan for each combination of persons in households A and B. Let's compare these results to the one in Table 7.4, the analogous table for experiment 1. The first observation is that this time other plans are preferred for the same combination of persons attending the meal in the same room. For example, when setting the living room

table for Alvin and Simon, the robot should stack each cup on a plate and carry the two covers separately. In contrast, when the plates are already stacked in the cupboard, the plan carrying the stack and then the two cups with both arms is to be preferred.

Also the variance in the plans to be chosen is higher in this experiment. Whereas the winning plan in experiment 1 is either P_4^d or P_7^d when more than two persons are involved, the best plan can be any of P_1, P_4, P_5, P_7, P_9 or P_{10}. Moreover, plan P_1 once even showed worse performance than P_0 in experiment 1 (see Figure 7.2, line 18), but shows the best performance in several cases in this experiment. This means that there is no best plan for all situations and that it is hard to tell from an analytical point of view which plan to use.

The second observation when compared to the results of experiment 1 is the smaller gain in performance by the plan transformations. For the living room table we still get performance enhancements of more than 30 %, but there are other cases, where even the default plan is chosen as in the case of setting the table for one person in kitchen B. The reason is that the plates are already stacked in experiment 1, whereas the stacking takes time in this experiment. In the case of laying the kitchen table in household B for Alvin, Theodore and Simon it is even best to carry the plates one by one. Besides, the potential for optimization is especially high when auxiliary goals are involved as in the case of leaving the cupboard doors open.

In sum, this experiment confirms our observations of experiment 1 that plan transformations improve the performance significantly and that the choice of the best plan depends on the specific task and environment. We have further shown that the state the environment is in (the plates being in the cupboard or in the dishwasher) greatly influences the performance of plans and leads to different preferences of plans.

Table 7.5 Experiment 3: Summary of best plans for all combinations of persons and rooms in households A and B.

	Household A						Household B		
persons	kitchen table			living-room table			kitchen table		
T	P_1		-11.0 %	P_1		-26.7 %	P_0		-0.0 %
A,T	P_5	[2]	-6.2 %	P_5	[2]	-23.3 %	P_5	[2]	-6.8 %
T,D	P_4		-17.7 %	P_4		-30.6 %	P_4		-13.6 %
T,S	P_4		-14.6 %	P_1		-28.8 %	P_4		-14.6 %
A,S	P_4		-12.3 %	P_1		-25.7 %	P_4		-13.2 %
A,T,S	P_1		-11.4 %	P_7	[3]	-29.3 %	P_3		-3.3 %
A,T,D	P_4		-12.7 %	P_7	[3]	-27.5 %	P_4		-9.3 %
T,S,D	P_7	[3]	-12.9 %	P_7	[3]	-34.2 %	P_4		-13.2 %
A,T,S,D	P_9	[4]	-10.5 %	P_9	[4]	-30.7 %	P_{10}	[4]	-12.3 %

7.1.4 Experiment 4: Using the Built-in Tray

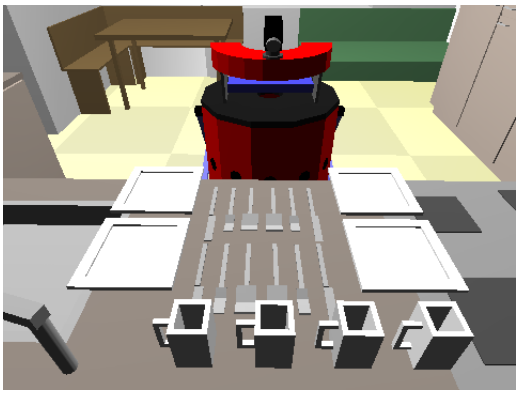
We repeated the last experiment with a robot that has been equipped with an additional tray on top of it (see Figure 7.5(b). This new resource makes other plan transformations possible, producing more plans to choose from. Table 7.6 shows a list of additional plans and their descriptions.

Although one might expect to open new ways of gaining performance, the impact of using a tray is relatively small. In the kitchen of household A the fastest plan never makes use of the tray. Table 7.7 shows the result for all combinations of two people to be served in the living room. Here the plan P_{18} performs better than the plans without tray in two cases. Still, without the tray, the performance would only be worse by 12.6 s and 6.2 s respectively.

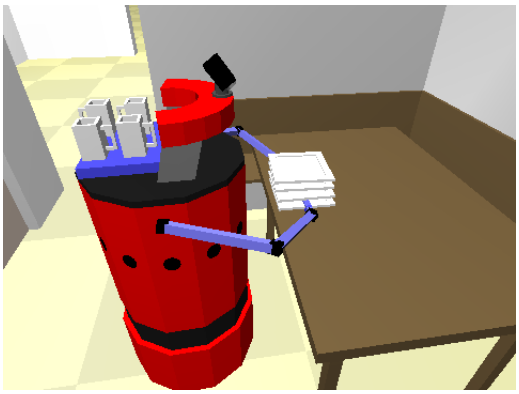
The explanation of this phenomenon has already been hinted at in the last experiment, where the performance gain was less striking than in experiment 1, because of the time needed for the stacking operation. In the current state of the art, robot navigation has been examined in much more detail than robot manipulation. Therefore, manipulation tasks are much slower compared to human manipulation than the navigation. Thus, the only cases where a tray should be used are ones where the navigation paths are very long, the ones into the living room. When operating in the kitchen, the robot is faster when navigating frequently, but avoiding complex manipulation tasks.

The effect of this imbalance can be lessened by changing the performance measure in a way that frequent navigation is punished. This might be necessary when many people are present in the environment and the robot bothers people by driving around all the time. Besides, it is probable that robot manipulation will be more efficient in the future,

Figure 7.5 Robot setting the table.



(a) Initial setting of experiments 3 and 4.



(b) Robot additionally using its tray for carrying cups in experiment 4.

Table 7.6 Additional plans obtained by plan transformations that take into account the presence of the built-in tray. The mnemonic for each plan is the same as in Listing 7.1. Additionally a rectangle containing cups specifies how many cups are carried on the tray.

Plan		Description
P_{12}	≡▪▫[2,3,4]	Carry the plates in a stack and at most one cup on the tray, carry the other cups one by one.
P_{13}	≡▪▪▫[3,4]	Carry the plates in a stack and at most one cup on the tray, carry the other cups using both arms.
P_{14}	≡▪▫▫[2,3,4]	Carry the plates in a stack and at most two cups on the tray, carry the other cups one by one.
P_{15}	≡▪▪▫▫[4]	Carry the plates in a stack and at most two cups on the tray, carry the other cups using both arms.
P_{16}	≡▪⁘[3,4]	Carry the plates in a stack and at most three cups on the tray. If there is a cup left, carry it in the gripper.
P_{17}	≡⁘[4]	Carry the plates in a stack and all four cups on the tray.
P_{18}	≜▫[2]	The table is to be set for two persons. Carry a stack composed of the plates and one cup and transport the second cup on the tray.
P_{19}	≜▪▫[3]	The table is to be set for three persons. Carry a stack composed of the plates and one cup, transport the second cup on the tray and come back for the last cup to be carried in the gripper.
P_{20}	≜▫▫[3]	The table is to be set for three persons. Carry a stack composed of the plates and one cup and transport the other two cups on the tray.
P_{21}	≜▪▫[4]	The table is to be set for four persons. Carry a stack composed of the plates and one cup, transport the second cup on the tray and come back to carry the last two cups one by one.
P_{22}	≜▪▪▫[4]	The table is to be set for four persons. Carry a stack composed of the plates and one cup, transport the second cup on the tray and come back to carry the last two cups using both arms.
P_{23}	≜▪▫▫[4]	The table is to be set for four persons. Carry a stack composed of the plates and one cup, transport two cups on the tray and come back for the last cup to be carried in the gripper.
P_{24}	≜⁘[4]	The table is to be set for four persons. Carry a stack composed of the plates and one cup, transport three cups on the tray.

getting nearer to the human proportion of manipulation versus navigation performance. When this happens and our robot is equipped with new manipulation routines, there will be no need to alter its control program, because it can transform its plans to match the new conditions.

Table 7.7 Experiment 4: Summary of best plans for all combinations of two persons in the living room of household A.

	Household A	
persons	living-room table	
A,T	P_5 [2]	-23.3 %
T,D	P_{18} [2]	-34.2 %
T,S	P_1	-28.8 %
A,S	P_{18} [2]	-27.5 %

7.2 Coordinating Activities

In Chapter 5 on page 103 we have presented a special two-step transformation rule for interleaving plan steps, so that two plans can be executed concurrently in an efficient manner. We used this rule to combine the table setting plan and the one for boiling pasta as presented in Figure 4.7 on page 65.

As a result, the transformation rule found that the time between putting the pot on the stove and putting the pasta into the water, that is the time until the water boils, can be used for bringing the plates to the table. Similarly, during the time needed until the pasta is done the robot should carry the cups to the table.

This plan transformation leads to a significant enhancement in the plan execution. In our experiment, the plan for boiling pasta took 694 s, and the table setting 213 s. The sequential execution thus needed 907 s. When the plans are combined in the way just described, the resulting plan was completed in only 708 s. This means that the execution of both plans only needs 12 s more than boiling pasta without setting the table. Although the table setting is fully completed while boiling pasta, the different navigation paths make the optimized plan a little slower than the boiling pasta plan alone.

7.3 Summary

The evaluation of our approach is mainly performed in the context of setting the table for one to four persons, the dishes being stored in different places in the kitchen. We

demonstrated that the choice of the performance measure has an influence on which plan is preferred. On the other hand, the best plan using one performance measure is usually also one of the best plans when applying other performance measures.

We could further demonstrate that our plans are very robust with respect to failures. Both failure detection and recovery are extremely reliable, the recovery being restricted to the robot dexterity in finding and retrieving lost objects. Moreover, the experiments show that state-of-the-art robots perform much better when navigating as compared to manipulating. Therefore, the saving of manipulation times outweighs the gain in fewer navigation tasks.

Most importantly, our experiments prove that the performance of robot plans can be greatly enhanced by plan transformations. Even more significant is the observation that different situations and environments require different plans for the robot to show the best performance. This supports our claims in Chapter 2 that the best plan for a robot acting in real-world domains cannot be determined analytically and that an adaptation to the circumstances at hand is vital.

Chapter 8

Conclusion

We conclude this work by summarizing our approach on transformational planning and pointing out the benefits of our TRANER system. After that we give directives on possible future work.

8.1 Transformational Planning in Everyday Environments

This work has presented in detail our approach of transforming plans in the context of everyday environments. We will now briefly recall the intricacies of household scenarios from Chapter 2, followed by a discussion of how our approach handles these challenges.

8.1.1 The Household Scenario

Plan-based control and transformational planning for a robot performing household chores is an interesting and challenging task. This is the case, because the problem includes (1) complex object manipulations (2) in a human, semi-structured environment and examines (3) planning of everyday activities in an area where humans perform especially well, because this environment requires adaptability, reliability, foresight, and handling of unforeseen events.

First of all, a household robot is confronted with a wide range of diverse tasks. On the one hand this requires the robot to be equipped with complex, powerful manipulators with a large number of DOFs. On the other hand, a large set of plans is necessary to fulfill the tasks. Therefore, the development effort for plans must be kept as small as possible, allowing the robot itself to perform optimizing adaptations.

Beside the complexity of the manipulation tasks, the robot must deal with surroundings that are adapted to human needs, not to that of the robot. One problem is the state

estimation. Not just the image processing and laser analysis, but especially the object descriptions on a higher level must be able to differentiate between necessary object properties for a task and those that are irrelevant. For example, when a plan tries to find a particular blue cup and cannot find it, the robot should find a suitable substitution. Depending on the situation, a red cup might be a good choice (when the cup is to be used as a container). In contrast, a blue plate is probably not a good choice for substitution, although it also shares the property of being blue with the original object.

Since a household robot is exposed to the presence of humans, it must show the kind of behavior people would expect it to show. This feature is often referred to as cognitive capabilities. It doesn't suffice to perform actions efficiently, but to act in a way that humans recognize as reasonable. This doesn't mean that a robot must necessarily imitate humans, but a robot navigating in an unforeseeable manner makes people feel uncomfortable, because they never know where the robot will move next. A robot showing cognitive behavior should also be able to explain what it is doing and why. This makes it easier for humans to find flaws in the robot's behavior and to comprehend its course of action. Another important aspect of cognitive behavior is the execution of auxiliary goals. A robot that leaves all the cupboard doors open and starts cleaning the dishes only when they are needed (and the leftovers of the last meal are dried) is not what a human would call an intelligent home help. Another challenge lies in the presence of failures. A household robot must anticipate and deal with failures just as humans do.

8.1.2 Summary of the Approach

We propose a transformational planning approach with reactive plans for tackling the challenges just discussed. In our TRANER system predefined default plans are adapted to a specific environment by plan transformations and plan evaluation on the basis of a simulation.

For implementing the default plans we propose an extended version of the reactive plan language RPL. The extensions comprise *parameter* and *object designators* as symbolical descriptions of objects in the world and parameters in the control program, as well as *semantic annotations* of plans for highlighting code pieces that perform failure recovery, make the robot perform actions at a certain location, specify designators, and declare plan steps as auxiliary goals. Besides, the default plans all follow a similar hierarchical plan structure using the programming constructs for semantic annotations. This approach leads to general, robust default plans that exhibit cognitive capabilities and can easily be transformed.

The plans are organized in a plan library. Here we distinguish four classes of plans or goals: *basic goals* for low-level control like navigation and simple arm movements, *basic manipulation* goals including gripping and simple object handling, *higher-level manipulation* goals for more sophisticated object manipulation, and *activities* representing

user commands like setting the table or boiling pasta. The categories group plans sharing the same characteristics with respect to transformability, failure handling, constraining and instantiating designators and auxiliary goals. Usually plans call other plans from the same or a lower level. In the context of interacting plans we have identified two kinds of challenges. The first question is when to instantiate designators in a hierarchical plan structure. The second deals with the accommodation of auxiliary plans in sequential plan interactions. Both issues can be dealt with by plan transformations.

Plan execution in TRANER requires an efficient *monitoring* mechanism observing the execution without modifying the plan code. When a plan is chosen from the library certain performance criteria are recorded, which are afterwards evaluated with a performance measure. Depending on the outcome of this evaluation, the plan is transformed or kept in the plan library as it is. After a plan transformation the same process of execution and evaluation is required. To make the whole procedure feasible and unperilous, a *realistic simulation* of the environment is necessary.

The core of TRANER are plan transformations, which create new plans out of a plan and a matching *transformation rule*. The structure of our rules allows the specification of complex, yet general, transformation rules in the rich language we use for representing our plans. We have presented a detailed example showing the evolution of a default plan by applying several transformation rules from our transformation rule library. The rules include syntactical transformations preparing the plan for subsequent transformations, reordering of plan steps, the usage of external and robot resources, the handling of auxiliary goals, and the optimization of free time by parallel execution of several plans. The rules are general enough to be applied to a wide range of plans.

8.1.3 Discussion

We have evaluated our approach in a realistically simulated kitchen with a B21 robot equipped with two arms. Using two activities — setting the table and preparing pasta — we could show the feasibility of our approach and prove the necessity of plan adaptation to specific environments by plan transformation.

Our experiments have clearly shown that plan performance is improved significantly by plan transformations. While the same result for a specific high-level plan in a specific environment could have been achieved by manually programming all the necessary plans, our approach only needs a small number of unoptimized, but robust default plans and a set of transformation rules. Considering the number of activities required from a household robot, TRANER reduces the development effort significantly and thus enables an economical development of more sophisticated robots.

The results show that TRANER needs an expressive and extendable plan representation for concurrent reactive and failure aware robot control together with a powerful transformation language.

Contrarily to most approaches in robot planning and control, we assume that the particulars of a specific environment and the circumstances at hand greatly influence the overall performance. We could show that this assumption holds in domains as complex as kitchens. The best plan can't be determined analytically, rather it has to be decided which plan performs better by prediction or simulation. The preferred plan for setting the table greatly differs depending on the number of persons to attend the meal, the room in which the meal is to be served, and the household the robot is working in.

Most other works on planning assume that the behavior of the robot can be deducted directly from the plan. We have shown, however, that this assumption doesn't hold in complex everyday environments. Therefore, our approach examines the actual behavior the plan produces by running it in an accurate simulation in order to ensure robust and flexible performance.

8.2 Prospects on Future Work

This work presents a fully functional system for a simulated robot. In future research activities it should be extended to decrease the development effort for plans and transformation rules and to narrow the search for the best plan. Besides, ongoing efforts are aimed at running the system on a real robot and in other application domains.

8.2.1 Plan Generation

In the current system, the default plans are coded by hand. Because household activities are very diverse and the demands on the default plans regarding optimality are low, the plans could also be generated from abstract descriptions of household activities.

Such knowledge can be found in the knowledge base Cyc and on websites like `ehow.com`, where household activities like setting the table for special events and recipes are provided. Such descriptions, however, are often inaccurate and leave out plan steps. For example, a recipe usually doesn't mention that the oven must be turned off. Most of the times, the missing activities are auxiliary goals that are taken for granted by humans, but must be complemented for a fully functional plan description.

8.2.2 Inspiration from Human Behavior

For displaying cognitive capabilities a robot needn't imitate human behavior, but often the observation of people acting in a household can provide a good guideline for a robot how to perform a plan efficiently and to find out if its behavior is acceptable for humans.

First, the observation of people can serve as a performance value for evaluating the robot's plans, giving indications when transformations are reasonable. This can be the duration of an activity, but also the resources used during a course of action.

Second, human behavior can give directions as to which transformation rule to use and thus narrowing the search space for the best plan. For example, when a robot has observed that people are faster in setting the table, it can analyze the difference in the plans, the human stacking plates whereas the robot carries objects one by one. In the next step, the robot can apply a transformation rule that makes its own behavior more similar to that of its human role model.

8.2.3 Application on Real Robots

Ongoing research aims at aligning the simulation used in this work with a real B21 robot. Figure 8.1(a) shows the robot in a kitchen, which corresponds to household B in our evaluation (cp. Figure 7.2 on page 119) To facilitate the state estimation, the kitchen is equipped with sensors like cameras and lasers. The cupboards can provide information about them being open or closed and what objects they contain by tagging objects with RFID chips and the cupboards with adequate readers. Also the kitchen devices provide additional sensors. An example is the knife shown in Figure 8.1(b), which is equipped with a pressure sensor.

The full functionality of a robot in a real kitchen still offers many challenging questions. First of all there is the state estimation. Although the environment can provide some information, the robot still has to perform sophisticated image processing. Besides, when humans are present, an additional complexity is the prediction of the movement of the people in the kitchen. Second, robot control in a real environment is harder than in simulation. Although our simulation mimics non-determinism, it is very reliable and the execution of activities can be repeated frequently. The development process on real robots is considerably slower. Finally, the hardware for robot arms is still very complex and expensive. Our current robot uses two Amtec PowerCube arms with grippers. Reaching and manipulating objects with these simple tools is extremely challenging.

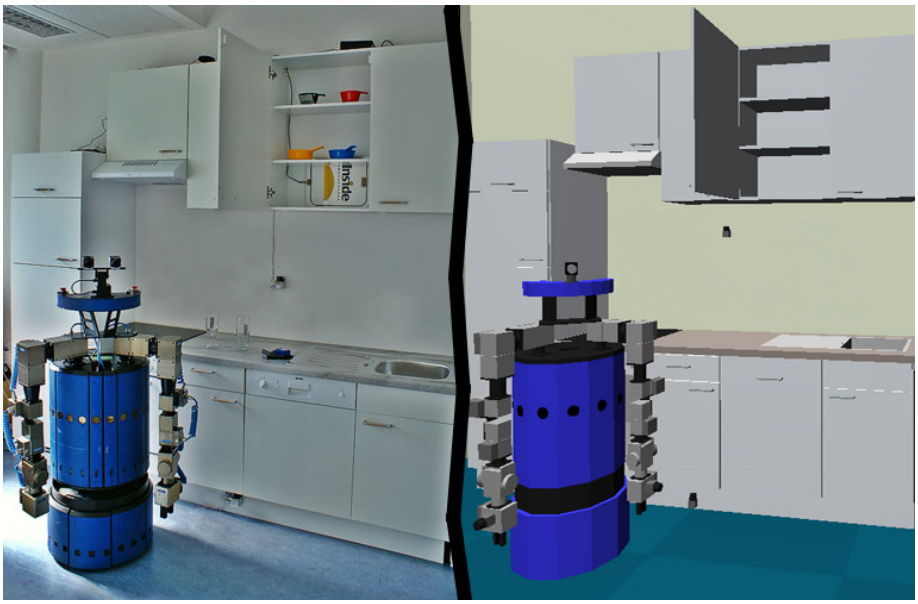
8.2.4 Extension of the Kitchen Scenario and Other Applications

Parallely to carrying over the current abilities of our simulated kitchen robot to a real one, we will extend the repertoire of tasks the robot can perform. This includes the ability to prepare a wider variety of meals, cleaning tasks, and the interaction with humans (Beetz et al. 2007; Beetz, Buss, and Wollherr 2007).

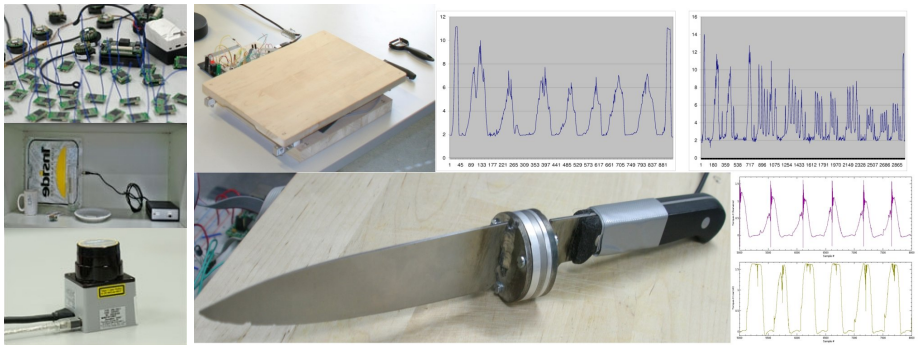
We further intend to demonstrate the generality of our approach and how it enhances the development of competent robots by using it for a completely different application. Currently a cognitive factory is under development, where all devices have information about their state, capacity and utilization. Using this information and the plan transformation mechanisms provided by TRANER, the processes in the factory can be enhanced by dynamically adapting the production process (Zäh et al. 2007).

The demonstration scenario is a flexible manufacturing system consisting of two CNC machines, an assembly robot, a material handling robot, an automatic storage unit, a computer controllable conveyor belt, and a quality measuring unit, which is further enhanced with sensor networks and high performance 3D laser sensors. Cognitive mechanisms enable the factory system to learn situation-specific models of production steps and to estimate the state of manufacturing processes. TRANER will allow to adapt individual production steps to react to perceived disturbances and respond to dynamically changed objective functions.

Figure 8.1 Real robot and kitchen environment with sensors.



(a) Real (left) and simulated (right) robot and kitchen.



(b) Excerpt of sensors used in the kitchen.

List of Figures

Bibliography

Agre, Philip E. (1988). The dynamic structure of everyday life. Technical report, Massachusetts Institute of Technology, Artificial Intelligence Lab.

Agre, Philip E. and David Chapman (1987). Pengi: An implementation of a theory of activity. In *National Conference on Artificial Intelligence (AAAI)*, pp. 268–272.

Agre, Philip E. and Ian Horswill (1997). Lifeworld analysis. *Journal of Artificial Intelligence Research* 6: 111–145.

Alami, R., R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand (1998). An architecture for autonomy. *International Journal of Robotics Research* 17(4).

Beetz, M. and D. McDermott (1997). Expressing transformations of structured reactive plans. In *Recent Advances in AI Planning. Proceedings of the 1997 European Conference on Planning*, pp. 64–76. Springer Publishers.

Beetz, Michael (2000). *Concurrent Reactive Plans: Anticipating and Forestalling Execution Failures*, Vol. LNAI 1772 of *Lecture Notes in Artificial Intelligence*. Springer Publishers.

Beetz, Michael (2001). Structured Reactive Controllers. *Journal of Autonomous Agents and Multi-Agent Systems. Special Issue: Best Papers of the International Conference on Autonomous Agents '99* 4: 25–55.

Beetz, Michael (2002a). *Plan-based Control of Robotic Agents*, Vol. LNAI 2554 of *Lecture Notes in Artificial Intelligence*. Springer Publishers.

Beetz, Michael (2002b). Plan representation for robotic agents. In *Proceedings of the Sixth International Conference on AI Planning and Scheduling*, pp. 223–232, Menlo Park, CA. AAAI Press.

Beetz, Michael, Jan Bandouch, Alexandra Kirsch, Alexis Maldonado, Armin Müller, and Radu Bogdan Rusu (2007). The assistive kitchen — a demonstration scenario for cognitive technical systems. In *Proceedings of the 4th COE Workshop on Human Adaptive Mechatronics (HAM)*.

Beetz, Michael, Martin Buss, and Dirk Wollherr (2007). Cognitive technical systems — what is the role of artificial intelligence? In Hertzberg, J., M. Beetz, and R. Englert, editors, *Proceedings of the 30th German Conference on Artificial Intelligence (KI-2007)*, pp. 19–42. Invited paper.

Beetz, Michael, Alexandra Kirsch, and Armin Müller (2004). RPL-LEARN: Extending an autonomous robot control language to perform experience-based learning. In *3rd International Joint Conference on Autonomous Agents & Multi Agent Systems (AAMAS)*.

Bonasso, P., J. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack (1997). Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Intelligence* 9(1).

Bonasso, R. P. and D. Kortenkamp (1995). Characterizing an architecture for intelligent, reactive agents. In *Working Notes: 1995 AAAI Spring Symposium on Lessons Learned from Implemented Software Architectures for Physical Agents*.

Bothelho, S. and Rachid Alami (2000). Robots that cooperatively enhance their plans. In Parker, Lynne E., George A. Bekey, and Jacob Barhen, editors, *5th International Symposium on Distributed Autonomous Robotic Systems (DARS)*, pp. 55–68, Knoxville, Tennessee, USA. Springer.

Brachman, Ronald (2002). Systems that know what they're doing. *IEEE Intelligent Systems* pp. 67 – 71.

Fikes, R. O. and N. J. Nilsson (1971). STRIPS: A new approach to the application of theorem proving to problem solving. Technical report 43r, AI Center, SRI International.

Firby, J. (1989). Adaptive Execution in Complex Dynamic Worlds. Technical report 672, Yale University, Department of Computer Science.

Firby, R., P. Prokopowicz, M. Swain, R. Kahn, and D. Franklin (1996). Programming CHIP for the IJCAI-95 robot competition. *AI Magazine* 17(1): 71–81.

Firby, R. James (1987). An investigation into reactive planning in complex domains. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

Firby, R. James, Peter N. Prokipowicz, and Michael J. Swain (1995). Plan representation for picking up trash. In *Seventh International Conference on Tools with Artificial Intelligence*.

Firby, Robert James (1989). Adaptive Execution in Complex Dynamic Worlds. Ph.D. diss., Yale University. Technical Report YALEU/CSD/RR #672.

Fox, Maria and Derek Long (2006). Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research* 27: 235–297.

Gat, Erann (1997). On three-layer architectures. *Artificial Intelligence And Mobile Robots* .

Georgeff, M. P. and A. L. Lansky (1987). Reactive reasoning and planning. In *AAAI-87 Proceedings*, pp. 677–682. American Association of Artificial Intelligence.

Georgeff, Michael P. and François F. Ingrand (1989). Monitoring and control of spacecraft systems using procedural reasoning. In *Proceedings of the Space Operations Automation and Robotics Workshop*.

Gerkey, Brian, Richard T. Vaughan, and Andrew Howard (2003). The Player/Stage Project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th International Conference on Advanced Robotics (ICAR)*, pp. 317–323.

Hammond, K. (1990). Explaining and repairing plans that fail. *Artificial Intelligence* 45(1): 173–228.

Hammond, Kristian J., Timothy M. Converse, and Joshua W. Grass (1995). The stabilization of environments. *Artificial Intelligence* 72(1-2): 305–327.

Ingrand, F., M. Georgeff, and A. Rao (1992). An architecture for real-time reasoning and system control. *IEEE Expert* 7(6).

Ingrand, François F., Michael P. Georgeff, and Anand S. Rao (1990). An architecture for real-time reasoning and system control. In *Proceedings of DARPA Workshop on Innovative Approaches to Planning*, San Diego, CA.

Ingrand, François Félix, Raja Chatila, Rachid Alami, and Frédérick Robert (1996). PRS: A high level supervision and control language for autonomous mobile robots. In *Proc. of the IEEE Int. Conf. on Robotics and Automation*, pp. 43–49, Minneapolis.

Kirsch, Alexandra and Michael Beetz (2007). Training on the job — collecting experience with hierarchical hybrid automata. In Hertzberg, J., M. Beetz, and R. Englert, editors, *Proceedings of the 30th German Conference on Artificial Intelligence (KI-2007)*, pp. 473–476.

McDermott, D. (1990). Planning reactive behavior: A progress report. In Sycara, K., editor, *Innovative Approaches to Planning, Scheduling and Control*, pp. 450–458, San Mateo, CA. Kaufmann.

McDermott, D. (1991). A Reactive Plan Language. Research Report YALEU/DCS/RR-864, Yale University.

McDermott, D. (1992a). Robot planning. *AI Magazine* 13(2): 55–79.

McDermott, D. (1992b). Transformational planning of reactive behavior. Research Report YALEU/DCS/RR-941, Yale University.

McDermott, Drew (1993). A reactive plan language. Technical report, Yale University, Computer Science Dept.

McDermott, Drew (2000). The 1998 AI planning systems competition. *AI Magazine* 21(2): 35–55.

McDermott, D.V., W.E. Cheetham, and B.D. Pomeroy (1991). Cockpit emergency response: The problem of reactive plan projection. In *IEEE International Conference on Systems, Man, and Cybernetics*.

Müller, Armin and Michael Beetz (2006). Designing and implementing a plan library for a simulated household robot. In Beetz, Michael, Kanna Rajan, Michael Thielscher, and Radu Bogdan Rusu, editors, *Cognitive Robotics: Papers from the AAAI Workshop*, Technical Report WS-06-03, pp. 119–128, Menlo Park, California. American Association for Artificial Intelligence.

Müller, Armin and Michael Beetz (2007). Towards a plan library for household robots. In *Proceedings of the ICAPS'07 Workshop on Planning and Plan Execution for Real-World Systems: Principles and Practices for Planning in Execution*, Providence, USA.

Müller, Armin, Alexandra Kirsch, and Michael Beetz (2004). Object-oriented model-based extensions of robot control languages. In *27th German Conference on Artificial Intelligence*.

Müller, Armin, Alexandra Kirsch, and Michael Beetz (2007). Transformational planning for everyday activity. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS'07)*, pp. 248–255, Providence, USA.

Myers, K. L. (1997). User guide for the procedural reasoning system. Technical report, Artificial Intelligence Center, SRI International, Menlo Park, CA.

Payton, David (1986). An architecture for reflexive autonomous vehicle control. In *Proceedings of the International Conference on Robotics and Automation (ICRA)*.

Pell, B., D. Bernard, S. Chien, E. Gat, N. Muscettola, P. Nayak, M. Wagner, and B. Williams (1997). An autonomous spacecraft agent prototype. In Johnson, L. and B. Hayes-Roth, editors, *Proceedings of the First International Conference on Autonomous Agents (Agents'97)*, pp. 253–261, New York. ACM Press.

Pollack, M. and J. Horty (1999). There's more to life than making plans: Plan management in dynamic, multi-agent environments. *AI Magazine* 20(4): 71–84.

Schoppers, M. J. (1987). Universal plans for reactive robots in unpredictable environments. In McDermott, John, editor, *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, pp. 1039–1046, Milan, Italy. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA.

Selman, Bart, Rodney A. Brooks, Thomas Dean, Eric Horvitz, Tom M. Mitchell, and Nils J. Nilsson (1996). Challenge problems for artificial intelligence. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pp. 1340–1345, Menlo Park, California. American Association for Artificial Intelligence.

Simmons, R. (1988). A theory of debugging plans and interpretations. In *Proc. of AAAI-88*, pp. 94–99.

Simmons, Reid and David Apfelbaum (1998). A task description language for robot control. In *Proceedings of the Conference on Intelligent Robots and Systems (IROS)*, Victoria, Canada.

Simon, Herbert A. (1955). A behavioral model of rational choice. *Quarterly Journal of Economics* 69: 99–118.

Simon, Herbert A. (1996). *The Sciences of the Artificial - 3rd Edition*. The MIT Press.

Stulp, Freek and Michael Beetz (2005). Optimized execution of action chains using learned performance models of abstract actions. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI)*.

Sussman, G. (1977). *A Computer Model of Skill Acquisition*, Vol. 1 of *Aritficial Intelligence Series*. American Elsevier, New York, NY.

Sussman, Gerald Jay (1973). A computational model of skill acquisition. Ph.D. diss., Massachusetts Institute of Technology.

Williams, Brian C., M. Ingham, S. H. Chung, and Paul H. Elliott (2003). Model-based programming of intelligent embedded systems and robotic space explorers. *Proceedings of the IEEE: Special Issue on Modeling and Design of Embedded Software* 9(1): 212–237.

Zäh, M. F., C. Lau, M. Wiesbeck, M. Ostgathe, and W. Vogl (2007). Towards the cognitive factory. In *Proceedings of the 2nd International Conference on Changeable, Agile, Reconfigurable and Virtual Production (CARV 2007)*.

Zöllner, Raoul, Michael Pardowitz, Steffen Knoop, and Rüdiger Dillmann (2005). Towards cognitive robots: Building hierarchical task representations of manipulations from human demonstration. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1535–1540, Barcelona, Spain.